



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

Introduction

PART I—Introduction to Windows 2000 Programming

CHAPTER 1—The Windows 2000 Environment

The Hardware Environment

Machine and RAM Requirements

Hard Drive Space Requirements

File Systems: FAT16 versus FAT32 versus NTFS

Differences between Windows 2000, Windows 98, Windows 95, and Windows NT

Windows NT

Windows 2000 (NT5)

Programming Cross-System Applications

Dual-Boot System Operations

Summary

CHAPTER 2—Application Programming for Windows 2000

WinHello: An Introductory Windows Program

The Windows.H Header File

The WinMain Procedure

Messages and Event-Driven Programming

The WndProc Procedure

Message Loop Alternatives

The .DEF (Define) File

The Template Program: A Template for Application Development

The Template.C Source Code

[The Template.I File](#)

[The Template.H Header File](#)

[The Template.RC Script](#)

[Windows Conventions and Data Types](#)

[Variable Names and Hungarian Notation](#)

[Data Types](#)

[Data Structures](#)

[Handle Identifiers](#)

[Summary](#)

CHAPTER 3—The Windows NT File System (NTFS)

[Characteristics of NTFS](#)

[NTFS Filenames](#)

[Limitations on NTFS Volumes](#)

[Non-NTFS File Systems](#)

[The FAT File System](#)

[The Protected-Mode FAT File System \(VFAT\)](#)

[NTFS File Attributes](#)

[Querying File/Drive Information](#)

[Querying File Systems](#)

[Universal Long Filenames](#)

[Multiple Data Streams in NTFS](#)

[The *NTFS_Streams* Demo](#)

[Linking Files under NTFS](#)

[Summary](#)

CHAPTER 4—Windows 2000—Specific Services

[Multiple Monitor Support](#)

[How Multiple Monitors Are Used](#)

[NTFS Changes](#)

[The Change Journal](#)

[Reparse Points](#)

[Sparse Files](#)

[Encryption](#)

[Volume Quotas](#)

[Native Structured Storage](#)

[Volume Mount Points](#)

[Power Management](#)

[Creating Power-Aware Applications](#)

[Querying Power Status](#)

[The Windows 2000 Kernel](#)

[Job Objects and Functions](#)

[The Windows 2000 User Interface/GDI](#)

[Message-Only Windows](#)

[Alphablend](#)

[GetGuiResources](#)
[SystemParametersInfo](#)
[Summary](#)

PART II—Application Design

CHAPTER 5—Pop-Ups: Tip Windows and Menus

[Pop-Up Tip Windows](#)
[The *Popups* Demo Program](#)
[Pop-Up Message Windows](#)
[Pop-Up \(Floating\) Menus](#)
[Pop-Ups in Dialogs](#)
[Summary](#)

CHAPTER 6—Creating and Synchronizing Multiple Threads

[Thread Concepts](#)
[When to Create Threads and Processes](#)
[Thread Objects](#)
[Scheduling and Synchronizing Threads](#)
[About Win32 Object Handles](#)
[Thread-Related Commands](#)
[Making and Modifying Threads](#)
[Synchronizing Threads](#)
[An Example with Multiple Threads: The *Threads* Program](#)
[Initialization Procedures](#)
[Window and Message Handler Procedures](#)
[Modification Procedures](#)
[Thread Procedures](#)
[The *MultiThreads* Program](#)
[Summary](#)

CHAPTER 7—Creating Processes and Pipes

[Process Concepts: Division of Labor](#)
[Inheritance](#)
[Life Cycle of a Process](#)
[Pipe Concepts: Communication between Processes](#)
[The Life Cycle of a Pipe](#)
[Varieties of Pipes](#)
[Process-Related Commands](#)
[Making and Modifying Processes](#)
[Getting Information about a Process](#)
[Changing Process Priority](#)
[Synchronizing Processes](#)
[Sharing Handles](#)
[Ending a Process](#)

Pipe-Related Commands

[Creating Anonymous Pipes](#)

[Creating Named Pipes](#)

[Connecting to an Existing Pipe](#)

[Modifying an Existing Pipe](#)

[Getting Information about an Existing Pipe](#)

[Reading and Writing through a Pipe](#)

[Synchronizing Connections](#)

[Making Transactions](#)

[Disguising the Server](#)

[Destroying a Pipe](#)

Processes Communicating through a Pipe

[One-Way Communication: The *Anonymous Pipe* Version](#)

[Multiple Communications: The *Named Pipe* Version](#)

Summary

CHAPTER 8—Using the Registry

Welcome to the System Registry

[The Registry Editor](#)

[The Registry Structure](#)

[Registry Access Methods](#)

The *Regs_Ops* Demo: Registry Operation

[Preparing for the *Reg_Ops* Demo](#)

[Opening a Key](#)

[Querying Registry Keys and Values](#)

[Setting Registry Keys and Values](#)

[Interpreting System Error Messages](#)

[Running the *Reg_Ops* Demo](#)

Summary

CHAPTER 9—Exception Handling

Traps and the Trap Handler

Structured Exception Handling

[Provisions for Error Handling](#)

[Filtering Exceptions](#)

[What Is Exceptional?](#)

[Frame-Based Exception Handling](#)

[Order of Execution](#)

[Exception Handling and Debuggers](#)

[Exception-Handling Macros](#)

[Exception Classes](#)

[Plug-and-Play Event Messages](#)

The *Exceptions* Demo: Exception-Handling Examples

[A Simple Exception Handler](#)

[A Nested Exception Handler](#)

[A Failed Catch Example](#)

[A Resource Exception Example](#)

[A User Exception Example](#)

[Summary](#)

CHAPTER 10—Memory Management

[Memory-Management Concepts](#)

[The Virtual Memory Manager \(VMM\)](#)

[The Address Space](#)

[Mapped Files](#)

[Memory Management Commands](#)

[Virtual Memory Commands](#)

[Heap Functions](#)

[Global and Local Memory Commands](#)

[Validation Commands](#)

[C Runtime Equivalents](#)

[The List Demo: Reserving and Committing Memory](#)

[File-Mapping Objects](#)

[Mapped Files](#)

[File-Mapped Object Sharing](#)

[The Browser Demo: Mapping Memory](#)

[Summary](#)

CHAPTER 11—Security and Cryptography

[Comparing Windows NT/2000 with Windows 95/98 Security Support](#)

[NT/2000 Security](#)

[Planning for Security](#)

[Users and Objects](#)

[Built-In Security versus Application Security](#)

[Security Is for Servers](#)

[Structures and Terminology](#)

[Security Attributes](#)

[The Four Major NT/2000 Security Structures](#)

[Client Impersonation](#)

[Checking Access Rights](#)

[NT/2000's Security API](#)

[Checking and Updating SDs: The *FileUser* Program](#)

[Retrieving the SID](#)

[Getting the File's SD](#)

[Working with the DACL](#)

[Adding ACEs](#)

[Setting the New SD](#)

[Encryption Technology Concepts](#)

[Secure Communications and the Government](#)

[Do You Need Encryption?](#)

[Encryption Methods](#)

[Crypto API and Cryptographic Service Providers \(CSPs\)](#)

[The Crypto API Functions](#)

[Key Functions](#)

[Encryption Functions](#)

[Hashing Functions](#)

[Adding Encryption Support to Applications](#)

[Getting Started](#)

[Required Project Settings](#)

[Creating a Default CSP](#)

[Encrypting a File](#)

[Decrypting a File](#)

[Inadvertent Compromising of Security](#)

[Summary](#)

PART III—Windows 2000 Graphics and Multimedia

CHAPTER 12—The Windows Graphics Device Interface

[The Device Context](#)

[Device-Context Handles](#)

[Why Query a Device Context?](#)

[Accessing the Device Context](#)

[Acquiring an Information \(Device\) Context](#)

[Device-Context Information](#)

[GDI Identifier Constants](#)

[The *DC* Demo: Reporting Device Capabilities](#)

[Mapping Modes](#)

[Windows Mapping Modes](#)

[Getting and Setting Mapping Modes](#)

[Coordinate Translations: Viewport versus Window](#)

[The *Modes* Demo: Mapping Mode Effects](#)

[Summary](#)

CHAPTER 13—DirectX and OpenGL Graphics—An Overview

[DirectX](#)

[Playing with DirectX](#)

[OpenGL](#)

[Fahrenheit: The Next Generation for Graphics?](#)

[Summary](#)

CHAPTER 14—Multimedia Operations

[Windows Multimedia Support](#)

[Multimedia Devices](#)

[Multimedia Services](#)

[Sound Data Formats](#)

[Three Easy Ways to Play Sounds](#)
[Media Control Interface \(MCI\) Operations](#)
[Multimedia File I/O Functions](#)
[RIFF Files](#)
[The Multimedia File I/O Functions](#)
[The ShowWave Demo: A Sound Recorder Clone](#)
[The ShowWave Header Files and Resource Script](#)
[The MCI Module](#)
[The MMIO Module](#)
[The GraphWin Module](#)
[Some Improvements for the ShowWave Program](#)
[Summary](#)

CHAPTER 15—Multimedia MMX for the Pentium III

[Intel and MMX](#)
[Support for SIMD](#)
[SIMD Floating-Point Instructions](#)
[The SIMD Floating-Point Operations](#)
[SIMD Integer Instructions](#)
[The SIMD Integer Instructions](#)
[SIMD Cacheability Control](#)
[Streaming Store](#)
[Move Masked Byte\(s\) to Memory](#)
[Prefetch](#)
[Store Fence](#)
[SIMD State Management](#)
[SIMD Control Status Register](#)
[Load and Store MXCSR Register](#)
[State Save and Restore](#)
[Resources for the SIMD Extensions](#)
[Summary](#)

PART IV—Database Programming

CHAPTER 16—Universal Data Access Architecture

[COM-Based Universal Data Access](#)
[OLE DB](#)
[ActiveX Data Objects](#)
[Internet Advantages with OLE DB and ADO](#)
[Cross-Platform Support with UDA](#)
[Data Access in the Internet Age](#)
[Summary](#)

CHAPTER 17—ODBC and SQL Database Access

[Basic Database Organization](#)

- [Database Tables](#)
- [Structured Query Language \(SQL\)](#)
- [SQL Statements and MFC](#)
- [Restricting Record Selections](#)
- [Sorting Records](#)
- [Registering a Database with ODBC](#)
- [Programming a Database Application](#)
- [Accessing Data: The CTitleSet Class](#)
- [Displaying Data: The CTitleForm Class](#)
- [Adding Additional Table Access](#)
- [Modifying Classes after Creation](#)
- [Modifying the New Classes](#)
- [Updating and Adding Records](#)
- [CRecordset Update Transactions](#)
- [Editing an Existing Record](#)
- [Appending a New Record](#)
- [Checking the Legality of Operations](#)
- [Record Locking](#)
- [Transactions](#)
- [The *BookList* Example](#)
- [More on RFX_\(\) Function Calls](#)
- [A Sequence of Operations](#)
- [Adding Edit Operations](#)
- [Modifying the Display Form](#)
- [Resetting Edit Fields](#)
- [Resetting Controls](#)
- [Adding a New Record](#)
- [Customizing the Add Entry Form](#)
- [Canceling Edit or Add Record Operations](#)
- [Summary](#)

[CHAPTER 18—ADO Database Access](#)

- [OLE DB](#)

- [ADO](#)

- [Support for a Transient Environment](#)

- [Data Providers versus Data Consumers](#)

- [Data Sources](#)

- [The ADO Object Model](#)

- [The Connection Object](#)

- [The Command Object](#)

- [The Recordset Object](#)

- [The Fields Collection](#)

- [The Errors Collection](#)

- [The Parameters Collection](#)

[The Properties Collection](#)
[Shortcomings in ADO](#)
[Programming with ADO](#)
[Creating an ADO Database Program](#)
[Opening a Connection](#)
[Cleaning Up](#)
[Building an Item List](#)
[Presenting the Details](#)
[Updating a Record](#)
[Appending a New Record](#)
[Deleting a Record](#)
[More on Supports Testing](#)
[Summary](#)

[PART V—Internet and Network Programming](#)

[CHAPTER 19—Internet Support](#)

[Four Years' Worth of Progress at Microsoft](#)
[Internet-Related Concepts](#)
[IP Addresses](#)
[Internet Protocols](#)
[Sockets and Winsock](#)
[The Winsock 2 API Functions](#)
[Socket Functions](#)
[Data-Conversion Functions](#)
[Database-Lookup Functions](#)
[Asynchronous Versions of the Winsock Functions](#)
[The *FindAddr* Demo: An IP Address Lookup Example](#)
[The *SendMail* Demo: Sending E-Mail](#)
[Winsock Applications](#)
[The Internet API Functions](#)
[Connect and Disconnect Functions](#)
[File Locating, Reading, and Writing Functions](#)
[FTP-, Gopher-, and HTTP-Specific Functions](#)
[Cookie Functions](#)
[URL-Related Functions](#)
[Time/Date Conversion Functions](#)
[Error and Status Functions](#)
[MFC Wrapper Classes](#)
[The *FtpGet* Demo: Simple FTP Transfer](#)
[Internet Applications](#)
[ActiveX Web Controls](#)
[The *WebView* Demo: A Simple Web Browser](#)
[Summary](#)

CHAPTER 20—Network Programming

The NetBIOS Functions

The WNet API

WNet and Network Resources

The *WNetDemo* Demo: Mapping a Drive

The LANMan Functions

Winsock 2.0 for Network Programming

New Features of Winsock 2

The *MacSock* Demo: Winsock 2 and AppleTalk

Named Pipes and Mailslots for Network Programming

Remote Procedure Calls

RPC Concepts

Creating an RPC Project

Mathlib.CPP: A Function Library

Mathlib.IDL: The IDL Function Interface Definition

Mathlib.ACF: The Marshaling Interface Handle File

Mathlib.H, Mathlib_c.C, and Mathlib_s.C

The *RPCClient* Demo: A Client Application

The *RPCServ* Demo: A Server Application

RPC Client and Server Testing

Summary

PART VI—COM, COM+, and Active Directory

CHAPTER 21—COM / COM+ Overview

What Is COM?

COM Terminology

COM Pros and Cons

The Benefits of COM

The Limitations of COM

COM Components and Interfaces

What Are Interfaces?

Characteristics of Interfaces

Types of Interfaces

Interface Rules

Interface Design

Implementation Rules for COM Components

The Implement IUnknown Rule

Memory Management Rules

Reference Counting Rules

COM Activation

Types of COM

COM Clients

COM Servers

- [ActiveX Controls](#)
- [COM and Object-Oriented Techniques](#)
 - [Encapsulation](#)
 - [Abstraction](#)
 - [Polymorphism](#)
 - [Inheritance](#)
- [COM+](#)
- [COM+ and Windows DNA](#)
 - [The User Interface Tier Technologies](#)
 - [The Middle Tier Technologies](#)
 - [The Database Tier Technologies](#)
- [The Component Services Snap-In](#)
- [Transactions](#)
- [Queued Components \(QC\)](#)
 - [Limitations of Real-Time Architectures](#)
 - [Transactional Message Queuing](#)
 - [Queued Components Architecture](#)
 - [Runtime Architecture](#)
 - [Queued Components Failure Recovery](#)
 - [QC Security](#)
- [Dynamic Load Balancing](#)
- [Object Pooling](#)
- [Summary](#)

[CHAPTER 22—Building and Using COM Servers in VC++](#)

- [A Look at an IDL File](#)
- [Building Our First COM Server](#)
 - [Defining Custom Interfaces](#)
 - [Implementing IUnknown and Our Custom Interface](#)
 - [Completing the COM Server](#)
 - [Creating a Test Client](#)
- [Building COM Servers with ATL](#)
 - [A Look at ATL](#)
 - [Building an In-Process COM Server with ATL](#)
 - [Building Out-of-Process COM Servers with ATL](#)
- [Threads and COM Servers](#)
 - [Multithreaded Win32 Applications](#)
 - [Threading COM Components](#)
- [Automation and IDispatch](#)
 - [IDispatch Implementation with VC++](#)
 - [Automation with ATL](#)
 - [Automation Data Types](#)
 - [More on Type Libraries](#)
 - [A C++ Automation Client](#)

[CHAPTER 23—New COM Features in Windows 2000](#)

[Synchronization Mechanisms](#)

[The COM Synchronization API](#)

[The COM Synchronization Interfaces](#)

[Asynchronous COM](#)

[Asynchronous Interface Construction](#)

[Asynchronous Interface Calls](#)

[Notes on the Asynchronous Server and Client](#)

[Asynchronous Server Processing](#)

[Call Serialization and Auto Complete](#)

[COM Pipes](#)

[COM Pipe Interfaces](#)

[Asynchronous Pipes and Read-Ahead](#)

[Call Objects and Call Cancellation](#)

[Call Cancellation Requests](#)

[Call Cancellation Handling](#)

[Lightweight Handlers](#)

[Standard LWH](#)

[Custom LWH](#)

[Summary](#)

[CHAPTER 24—Active Directory Operations](#)

[Active Directory](#)

[The Lightweight Directory Access Protocol \(LDAP\)](#)

[Active Directory Services](#)

[Preparing to Use ADSI](#)

[Active Directory Properties](#)

[ADSI Objects](#)

[The IADs Interface](#)

[The *ADsPropertyList* Demo](#)

[The *ADSIAddUser* Demo](#)

[IADsUser Methods](#)

[The *AddGroup* Demo](#)

[ADsPath Specifications](#)

[*AddGroup* Demo Provisions](#)

[Other ADSI Demo Programs](#)

[Summary](#)

[Index](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

ACKNOWLEDGMENTS

While an author must necessarily assume sole responsibility for the contents of the volume bearing his or her name, the truth is always that any book is the end product of a great many contributors, both directly and indirectly.

First and foremost, I am indebted to Ash Rofail and Yasser Shohoud for the material on COM and COM+ that has been excerpted and adapted from their book *Mastering COM and COM+* (published by Sybex). For those who desire more extensive coverage, *Mastering COM and COM+* is heartily recommended.

With this acknowledgment aside, there still remain a great many individuals whose efforts in preparing this volume deserve recognition. To each of these, I offer:

- My thanks to Chad Mack at Sybex for his support, for keeping a myriad of miscellaneous details from becoming an overwhelming mountain of tedium, and not least, simply for putting up with me.
- My thanks to Denise Santoro at Sybex for trying to keep this project on track.
- My thanks to Linda Orlando at Sybex for her incessant nitpicking and unceasing demands for consistency as well as ensuring that my pawky prose was always appropriately polished.
- My thanks and appreciation to Greg Guntle for a very thorough tech edit, for his careful patience, and for his many other assistances as well.
- My thanks to Max Vaughn at Microsoft for providing cogent answers to a few of the mysteries surrounding Active Directory and ADSI.
- My thanks to Prash Shirolkar at Microsoft for helping straighten out a few database problems and for offering several valuable insights.
- And, not to detract from any of the preceding in any fashion or degree, my very sincere thanks go as well to the production team at Sybex, who have also left their imprint on this volume.

INTRODUCTION

Windows 2000 is an extension of Windows NT but also owes a lot—particularly in the details of the desktop and many utilities—to Windows 95/98. Superficial appearances aside, however, Windows 2000 is also a

distinct departure from (or improvement upon) previous versions of Windows.

While some of the examples used in this book are similar to examples used in previous versions of Windows, many are new and demonstrate new features, capabilities, or application possibilities...and some have simply changed in response to changes in the operating system.

We will talk about a lot of the changes, particularly those that affect application development, but we'll also be covering some of the background changes as well, which affect us only indirectly.

With this said, the following is an expanded outline of the topics and subjects covered in this book.

Part I: Introduction to Windows 2000 Programming

Part I of this book offers an introduction to Windows programming—or, more specifically, to Windows 2000 programming. Beginning in Chapter 1, we'll look at the Windows 2000 environment and how it differs from Windows 95/98 and from Windows NT.

In Chapter 2, we'll review the basics of application development for Windows, look at programming and notational conventions, and offer a couple of simple applications. Then, in Chapter 3, we'll look at file systems in general and the NTFS file system in specific. We'll also look at operating and file system compatibilities (and incompatibilities) and discuss some of the choices and trade-offs.

In Chapter 4, the topic is Windows 2000–specific services including support for multiple monitors, changes to the NTFS file system, the power management API, and future proposals for power management. We'll also look at the Windows 2000 kernel, as well as the User Interface and GDI.

Part II: Application Design

Application design is a broad topic, and I won't pretend or suggest that we have comprehensive coverage of the topic here. Even so, we will discuss a number of important elements, beginning in Chapter 5 with extending resource elements to include popup menus, tip windows, and popup lists.

In Chapter 6, the topic becomes multithreading applications, a subject that will become increasingly important as multiprocessor systems become standard. Chapter 7 takes us to using processes and pipes to communicate between threads.

Chapter 8 introduces the registry and registry keys as mechanisms for maintaining application-critical data as well as user preferences.

In Chapter 9, we'll examine exception handling and how to prevent applications from terminating when abnormal conditions are encountered, while Chapter 10 looks at memory management and how processes can share blocks of memory through memory-mapped files.

Finally, in Chapter 11, we'll conclude this section by looking at security and cryptography.

Part III: Windows 2000 Graphics and Multimedia

In Part III, we'll begin with Chapter 12 to look at how graphic systems work and how to identify system capabilities.

In Chapter 13, we'll discuss DirectX and OpenGL for graphics and multimedia, looking further into multimedia applications in Chapter 14. Last, we'll wrap up this topic in Chapter 15 as we examine the newest MMX capabilities.

Part IV: Database Programming

In Part IV, to introduce database programming, we'll start in Chapter 16 by looking at Universal Data Access Architecture. Then, in Chapter 17, on a more practical basis, we'll see what can be done using Access and SQL. Finally, in Chapter 18, we'll look at using Active Data Objects (ADO) as an alternative method of accessing data.

Part V: Internet and Network Programming

Chapters 19 and 20 are almost impossible to separate since Internet and local network operations tend to merge together and the dividing line between the two—with the advent of virtual private networks—is becoming increasingly blurred.

Part VI: COM, COM+, and DCOM

Possibly one of the hottest topics today is COM/COM+. Chapter 21 offers an overview of COM/COM+ operations, while Chapter 22 introduces building COM servers. Chapter 23 takes a look at new COM features in Windows 2000.

Lastly, to wrap this subject up, in Chapter 24 we'll look at Active Directory operations using both ADSI and LDAP.

Supplemental Material

In order to keep this book from becoming excessively large (and potentially inviting lawsuits from people injured by a too-heavy volume), the CD accompanying this volume has 23 supplementary chapters together with additional programming examples.

Windows Basics

Since there are still programmers who are new to Visual C++ (or even to programming in C or perhaps to Windows programming), Supplement 1 offers an introduction to Windows message handling and introduces the Microsoft Foundation Classes. Supplement 2 introduces keyboard and mouse events, Supplement 3 looks at how the mouse is used in Windows, and Supplement 4 looks at child windows and control elements.

Windows 2000 Application Resources

The Windows OS offers a wide variety of ready-to-use tools for the programmer, and in Supplement 5, we'll take a first look at application resources. Then in Supplement 6, we'll concentrate on image resources including bitmaps, icons, cursors, and fonts.

In Supplement 7, we'll look at dialog boxes and the dialog box editor before discussing menu resources in Supplement 8. Finally, in Supplement 9, we'll wrap up application resources with accelerator keys, strings, and header files, before combining all of these resources in Supplement 10 by creating a file browser and viewer application.

Graphics Utilities

For those who are heavily into graphics operations, there are supplements that provide extensive coverage of graphics, beginning with colors and color palettes in Supplement 11 and then moving on to drawing tools in Supplement 12. Next, in Supplement 13, we'll look at brushes and at using bitmap brush textures.

In Supplement 14, we'll look at typefaces and fonts; then we'll move on to graphics utilities and file operations in Supplement 15 and cover other drawing and selection operations in Supplement 16. Finally, in Supplement 17, we'll look at graphics printing operations.

Supplement 18 continues graphics operations by introducing graphics selection operations, and Supplement 19 looks at methods allowing graphic images to respond to mouse events. Then, concluding in Supplement 20, graphics simulations are introduced.

Exchanging Information between Applications

Next, we'll look at some of the old standards for exchanging information between applications, beginning in Supplement 21 with metafile operations and continuing in Supplement 22 with clipboard data transfers. Finally, in Supplement 23, we'll take a look at the basics of OLE client/server applications.

Appendices

Appendix A offers some suggestions and data sources for the Windows 2000 Logo Program. Appendix B, however, may be more immediately relevant to programmers and developers, as we look at the Windows Installer service.

Comments and End Notes

This book began development with the Beta 2 release of Windows 2000 and has extended through Release Candidate 3...and there have been many changes along the way. A lot of these changes, of course, have been

fixes for bugs submitted by the author as well as by other testers, but as of RC1, Windows 2000 became what is known as “feature complete.”

Now, the term “feature complete” means simply that no features will be added. We must emphasize *added* because features that initially appeared in the Beta versions and in RC1 have since vanished (have been dropped) in later Release Candidate versions—and the final “Golden” release may see further omissions.

The point, of course, is that neither I nor any other author can say with absolute certainty prior to the final release which features will appear in Windows 2000 and which will not. Therefore, in writing this book, mention may appear of features or elements that—for a variety of reasons—fail to be included in the retail release of Windows 2000.

Features or elements that do fail to make the cut, however, are likely to be reinstated later either in the form of service packs or, in a few cases, as inclusion in separate products.

And a Last Word from the Author

Although this book covers all of the most important topics related to developing Windows 2000 applications, simple limitations both of time (mine) and space (paper) make it impossible to explore each and every topic of potential interest. Therefore, you should consider this book as a starting point from which you can continue and pursue such special interests as your needs and desires suggest.

Your comments and suggestions, however, are always welcome. You can contact me at ben@ezzell.org or through my Web site at <http://www.ezzell.org>.

This said, I wish you all luck in your endeavors and, by all means, keep on hacking and don’t let the pointy-haired boss get you down.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART I

Introduction to Windows 2000 Programming

- CHAPTER 1: The Windows 2000 Environment
- CHAPTER 2: Application Programming for Windows 2000
- CHAPTER 3: The NTFS File System
- CHAPTER 4: Windows 2000–Specific Services

CHAPTER 1

The Windows 2000 Environment

- Hardware requirements for Windows 2000 programming
- 16-bit, 32-bit, and NTFS disk file systems
- Differences between Windows 98, Windows NT, and Windows 2000
- Dual-boot system installation

If you're reading this book, the reasonable assumption is that you're interested in creating applications for Windows 2000. However, before you can design and implement applications, you need to understand the environment where the application will operate and the requirements necessary to support that environment. Toward this objective, this chapter describes the environment (or environments) created by the Windows 2000 operating system, both from a hardware and a software viewpoint.

Additionally, if you are writing applications for Windows 2000, you should also be aware of the similarities and the differences between Windows 98, Windows NT, and Windows 2000. This chapter compares the three operating systems.

The Hardware Environment

Although previous versions of both MS-DOS and Windows have tried to provide support for all earlier hardware systems, much of this backward compatibility has been achieved at the expense of limiting functional capabilities. The support for the owners of earlier computer models, including 808x and 80286 CPUs, has prevented programmers from making the best use of the potential capacities of newer machines and, of course, placed the end user—knowingly or otherwise—under similar restrictions.

At some future time and in a similar fashion, the restrictions inherent in the Windows 2000 operating system will also be considered archaic, restrictive, and cumbersome, and derided for not providing support for the newest and fastest CPUs and peripherals. When that time comes, Windows 2000 will, in turn, be replaced by a new operating system.

Further, since a 64-bit version of Windows 2000 is already under development—and discussed briefly later in this chapter—the ‘lifespan’ for a 32-bit Windows 2000 operating system (OS) is, rather obviously, limited.

NOTE:

The expectation of a 64-bit OS is not to suggest, of course, that 32-bit applications will instantly become obsolete, only that new horizons will be opening in the not too distant future. You can look for the release of Intel’s Merced chip to provide 64-bit processing, but remember, most applications will still need to support 32-bit platforms.

In any case, for the present, we’re programming for the 32-bit environment and need to meet its requirements. We’ll start with the hardware requirements for the computer system, RAM, and hard drive space.

Machine and RAM Requirements

First, and most important, Windows 2000 will not operate on an 8080, 8086, 80286, 80386, or 80486 system. For a WinTel (Windows/Intel) platform, the Windows 2000 operating system requires a minimum configuration consisting of a 166MHz X586 (or equivalent) with a minimum of 32MB of RAM (or 64MB for Win2000 Server) and a 2GB hard drive with a minimum of 650Mb free space. Recommendations, however, suggest 64MB (or more) of RAM for a workstation and 128MB for a server while the maximum RAM supported is 4GB.

NOTE:

According to recent announcements from Microsoft, the Compaq Alpha CPU will no longer be supported. A joint announcement from Compaq states that support for existing Alpha platforms will be continued but upgrades are expected to replace the Alpha platforms with Intel and Intel-compatible CPUs.

Other requirements include a 3.5” diskette drive, a CD-ROM (12x or faster recommended), VGA or higher video, and a mouse (optional but strongly recommended). Installation can also be performed from a network drive.

TIP:

A full list of compatible hardware can be found on the Win2000 CD in either \Support\Hcl.chm or hcl.txt or, for the latest and most current list, at www.microsoft.com/hwtest/hcl. (Caution: links and locations may change without warning.)

Ideally, you should also expect to replace any 16-bit drivers with 32-bit versions—contact the software vendor for new driver versions if necessary.

For hardware compatibility issues, the Windows 2000 Compatibility Tool can be used to identify existing hardware components and to verify compatibility with Windows 2000. The Compatibility Tool is found in the \support\ntct folder on the Windows 2000 Professional CD or you can download it from www.microsoft.com/windows/common/ntcompto.htm.

WARNING:

Many systems today are being sold with the new WinModem cards. Unfortunately, aside from being very poor modems as a class, these are incompatible with all versions of Windows 2000 due to a lack of software drivers necessary for modem emulation operations by the CPU.

Hard Drive Space Requirements

A second requirement is hard drive space. As you should already be aware, Windows 2000 does not install from floppy diskettes (well, I suppose you could use floppies, but why?) and it requires space to install a multitude of drivers, help files, system files, fonts, and so on (I won't waste space with the entire list).

To give you an idea of the requirements, running a quick assessment on the files and subdirectories on my own system, I found a total running close to a full gigabyte of files under the WinNT directory. Granted, that count includes DLLs (dynamic link libraries) and other files that were not part of the Windows 2000 installation itself but were added by other applications, and it also includes a host of files of questionable origin (which means that I'm not sure where they came from or why). But the main point is that this adds up to a third of a gigabyte higher than the stated minimum of 650MB just in the Windows directory (and subdirectories)!

Multigigabyte drives are nothing unusual today, and they're even relatively inexpensive—and they need to be easy to find and afford, considering the general bloat both in operating systems and applications. So, plan accordingly and figure on allocating at least a gigabyte drive or partition just for the Windows 2000 system itself.

Remember, however, that the suggested size is a minimum recommendation and, if you install all of your applications to the system-suggested default drive—which will be the same drive where Windows 2000 is located—a gigabyte partition can very quickly be exhausted and twice this size would not be inappropriate. (See the notes in “Choosing a File System,” later in this chapter, for information about using “Partition Magic” to resize drives/partitions.)

TIP:

Don't rush to throw that old drive away just because you've bought a newer, higher capacity drive. Most IDE drives can be ganged as master/slave pairs, even if they're from different manufacturers. And SCSI drives are even easier to gang together, with as many as six drives supported by a single SCSI controller.

In a pinch, you may want to use a disk-compression utility to increase the capacity of your existing hard drive. Before installing any compression utility, however, check that it is not only compatible with Windows 2000, but, if you are using a multi-boot system, that it is also compatible with any other operating systems you may be running.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Evolution of a File System

Over the years, the FAT16 system has gone through many evolutionary changes.

Originally, all files were written to a single file list, what we now refer to as the root directory. This single-directory format had many limitations, including a limit on the number of separate files that could be written on a disk. However, since hard drives were essentially unknown at the time and a 5¼-inch diskette held 100KB (that's kilo, not mega), there were far more serious limits than the number of files that could be written. And as limited as the capacity was, few people noticed or complained, since the alternative was recording on a cassette—yes, the kind you listen to music on (or did before the advent of CDs).

Later, as capacities rose (first was the 360KB diskette, then the 1.2MB, and finally the 1.4MB 3½-inch diskette), the need for a new format quickly became obvious. The concept of subdirectories was borrowed from the Unix world. With the new file system, the still-limited root directory entries are used to point to locations—not on the root directory tracks—where additional directory data (called subdirectories) is written. These subdirectory structures are stored as what are essentially special types of files.

Finally, a further enhancement was introduced with Windows 95 to allow long filenames. The original 8.3 format filenames were kept, but an extension to the directory was used to store a long filename. Of course, both OS/2 and Windows NT allowed long filenames before Windows 95 was introduced, but these were supported by properties of the quite different HPFS and NTFS file systems, which are not DOS/FAT-compatible.

Even with all of these advances, however, the FAT file system remained firmly committed to a 16-bit addressing system, which imposed its own limitations. Now, in its latest incarnation, we have the FAT32 file system, which offers the advantage of much higher storage efficiencies.

Originally, when the FAT32 system appeared with Win95 (B) and Win98, the existing NT4 OS simply did not recognize FAT32. Happily, with Windows 2000, both the NTFS and FAT32 file systems (as well as FAT16) are supported.

Of course, you should also remember that NTFS-formatted volumes—on dual-boot systems—will still not be recognized by Win98/95 systems (see “Dual-Boot System Operations,” later in this chapter).

File Systems: FAT16 versus FAT32 versus NTFS

While the venerable DOS FAT16 file system continues to be the *lingua franca* of the PC world—despite several attempts to relegate it to the dinosaur graveyard—Windows 98 introduces a new FAT32 file system with a number of important advantages.

Perhaps the biggest advantage of a FAT32 (32-bit) file system is how the disk drive allocates space. With the 16-bit addressing scheme used by the old FAT system (now called FAT16), there is a limit on the number of clusters that can be created on a single drive (or on a logical partition on a physical drive).

In brief, since the FAT16 can handle only 216 addresses, hard drives larger than 512MB need to allocate file space in 16KB blocks. This means that any fraction of a file less than 16KB still occupies a full 16KB block. Larger files occupy multiple blocks but, on most systems, there tend to be a lot of small files; many less than 100 bytes and even more in the 1KB or 2KB range. For example, in my own Win98 directories, I can find hundreds of files that are smaller than 2KB. On a FAT16 system, each of these would occupy a full 16KB of space.

NOTE:

Before you decide that my setup is unusual and that this doesn't really apply to your system, take a few minutes to check the files on your own hard drive. If you have a convenient utility that lets you sort files by size, you should find a very large number of files, in virtually any directory, that are less than 100 bytes and an even larger number that are less than a kilobyte in size. Also, the chances are that you will find a number of files which are 0 bytes in size, but which still occupy complete clusters.

Of course, a 1GB drive is also limited to 65,536 clusters, where the cluster size becomes 64KB.

The problem with the FAT16 system and the 1GB drive is that any individual file must occupy some integral number of clusters; that is, a file has a minimum size of one cluster. This means that if you write a file that uses only 1KB on a 250MB drive, for example, that 1KB file is still occupying 16KB of space; it is using $1/16$ th of that space while wasting 93.75 percent. But that is only a part of the problem, because that 4.3KB file is occupying 8KB of space on a 256KB drive. Here the waste is smaller—only 46.25 percent—but this is added to the rest of the lost space on your drive. On a 1GB drive, where the cluster sizes are larger, the problem becomes worse.

The upshot is that only very large files (and you have relatively few of these) actually use space efficiently. The odds are very good that the FAT16 file system wastes 20 to 30 percent or more of your drive space.

Switching to the FAT32 file system, which can handle 2^{32} addresses, means that cluster sizes can be as small as 1KB (although 4KB is much more common). This applies even to 1GB or larger drives, without partitioning. The result is that the amount of wasted drive space can easily fall as low as 2 percent or less.

The third alternative appears with Windows NT and the NTFS file system, which, like FAT32, is also capable of managing larger drives but has several advantages over both FAT16 and FAT32 drives. These advantages include:

- More secure files.
- Better disk compression.
- Support for large hard disks, up to 2TB (terabytes).
- Performance using NTFS shows less degradation for larger drives than FAT32 systems.
- The maximum drive size for NTFS is much greater than that for FAT.

TIP:

The NTFS file systems supported by Windows NT version 4 and by Windows 2000 (NT5) are similar but not entirely identical, even though for most purposes, the two function in the same fashion. While an NTFS 4-file system can be converted to an NTFS 5-file system, the reverse conversion is not supported.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Choosing a File System

Given the alternatives in file systems, the choices may appear confusing. The following table lists file systems by operating system compatibility, drive size, and, not least, compatibility for dual-boot operation.

TABLE 1.1 Choosing a File System

Operating System	FAT16	FAT32	NTFS
DOS (any version)	yes	no	no
Win3.x	yes	no	no
Win95	yes	yes ¹	no
Win98	yes	yes	no
WinNT4	yes	no	yes ²
Win2000	yes	yes	yes
Linux ³	yes	yes	read only
Drive Sizes	FAT16	FAT32	NTFS
minimum	N/A	512MB	512MB
maximum	less than 2GB	2TB (terabytes)	2TB (terabytes)
Notes	standard for diskettes		preferred for drives larger than 32GB
Dual Boot	FAT16	FAT32	NTFS
MSDos / any	yes	no	no
Win3.x/Win9x	yes	no	no
Win3.x/WinNT	yes	no	no
Win9x/WinNT	yes	no	no
Win9x/Win2000	yes	yes	no
WinNT/Win2000	yes	no	yes

¹OEM 2.0 and later only.

²Files created on an NTFS system under Windows 2000 may not be accessible under Windows NT 4.0.

³Linux partitions cannot be conveniently read by most other operating systems.

Refer also to the section “Dual-Boot Operating Systems” later in this chapter.

There are several utilities on the market that allow you to convert from FAT16 to FAT32 without losing your existing files or needing to reformat the drives or partitions. My own preference for this task is PowerQuest’s Partition Magic, which not only allows conversions from FAT16 to FAT32 (or the reverse), but will also allow you to change the cluster size and to resize, create, or delete partitions—all without wiping the existing data. Other FAT16 to FAT32 conversion packages offer similar services. (Partition Magic also supports converting FAT (only) partitions to NTFS volumes, but not the reverse.)

Differences between Windows 2000, Windows 98, Windows 95, and Windows NT

I could simply assert that there are no real differences between Windows 2000, Windows 98, Windows 95, and Windows NT 4.0, but that would be an oversimplification. In truth, there are worlds of differences. From a programming standpoint, however, most of the differences are invisible. Most applications written to run under Windows 2000 will also execute under Windows NT, Windows 98, and Windows 95, and vice versa. There are exceptions, but they are just that—exceptions.

Windows NT

When it was first introduced (as Windows NT 3.1), perhaps the biggest advantage of Windows NT over earlier versions of Windows (Windows 3.x) was the change from a 16-bit to a 32-bit operating system. Today, however, with the Windows 95/98 operating system dominating the new computer market, the introduction of Windows NT 4.0 was a less abrupt change and differed from NT 3.1 and 3.51 primarily in the change to the Windows 95/98-style interface.

Like Windows 95 and Windows 98, but unlike earlier versions of Windows, NT is its own operating system and does not rely on the underlying presence of the 16-bit DOS operating system. Unlike Windows 95 and Windows 98, NT 3.1, 3.51, and now 4.0 are all true 32-bit operating systems. Windows 95 and Windows 98, unfortunately, are still hybrids relying on both 16-bit and 32-bit core code.

Windows 2000 (NT5)

Like Windows NT, Windows 2000 (NT5) is also a 32-bit operating system but is expected (in a year or so) to appear as a 64-bit operating system. The biggest difference between Windows 2000 and Windows 95/98 lies in the fact that Windows 2000 now requires 32-bit drivers while the earlier OS versions have continued to accept legacy 16-bit drivers. This does not, of course, mean that the 16-bit applications cannot be used under Windows 2000—just as they have under Windows NT—but there are some applications (both 16- and 32-bit) which are not compatible with the more restrictive (and protected) NT environments.

Windows 2000 is distinctly different from Windows NT in a couple of respects. One, of course, is in the file systems supported, as discussed previously.

Another is found in the support for plug-and-play (PnP) devices. PnP support was introduced—with far more fanfare than it deserved at the time—in Windows 95 and was then greatly improved with Windows 98. However, since Windows NT predated PnP support, Windows 2000 is the first protected workstation/server system to actively (and fairly reliably) provide PnP device recognition.

Plug-and-Play

Windows 2000 does come with an extensive library of Plug-and-Play drivers, so there should be relatively few problems with oddball hardware installations. However, this does not mean that Windows 2000 will automatically recognize all peripherals and add-in cards. The situation has improved, but it is still far from perfect. The opprobrious revision “Plug-and-Pray” is not likely to vanish from the common vocabulary in the near future.

While most programmers will probably simply ignore Plug-and-Play at the present as not immediately relevant to their applications, for hardware and device developers, there is no question that PnP is important. To some it is even essential, since, given a viable alternative, users are unlikely to welcome or even tolerate

any return to the old manual, hit-and-miss configuration requirements. Therefore, in Chapter 9, “Exception Handling,” some mention has been included of the Plug-and-Play API functions.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Universal Serial Bus

The Universal Serial Bus (USB) is a new standard designed to replace the present device dependence on a limited number of hardware interrupts. Where existing systems have seen ever-increasing competition by peripherals—such as modems, mice, and sound cards—for the limited resources represented by the sixteen available hardware interrupts (a number of which were already reserved for the central system itself), the USB is designed to offer a virtually unlimited number of connections for existing and newer devices as well as provide effective throughput speed orders of a magnitude higher than existing channels.

While it still remains to be seen whether USB will be a universal panacea, there is no question that USB devices are appearing on the market in ever-increasing numbers and that support for USB—both in hardware with built-in USB ports and in software and OS support—is the new standard.

As a lagging parallel development, watch also for FireWire (IEEE 1394) to eventually appear as a high-speed port for data-intensive devices such as scanners.

New Technology Support

In addition to the USB peripherals, Windows 2000 includes support for Accelerated Graphics Port (AGP) video cards, MMX processors, and DVD drives. Furthermore, you will also be able to connect multiple displays (multiple monitors using multiple display cards). Likewise, the new Multilink Channel Aggregation (MCA) technology allows you to boost bandwidth by combining two or more communication lines (although few ISPs presently permit the multiple logons necessary to support MCA connections.)

WARNING:

At the time this book is being written (using Beta versions 1-3 and Release Candidate versions 1-2), some questions have appeared about whether multiple displays will be supported in the final release. Experiments with RC1 using multiple monitors were quite satisfactory (and are discussed later) and we hope this feature will continue to be supported but this cannot be stated with certainty.

Four Versions of Windows 2000

As you may be aware, Windows NT actually consists of both a workstation and a network server version. In

like fashion, Windows 2000 is a family of operating systems consisting of Windows 2000 Professional, Windows 2000 Server, Windows 2000 Advanced Server, and Windows 2000 Datacenter.

Windows 2000 Professional is a direct replacement for Windows NT Workstation and is designed for standalone systems as well as network clients. The Professional version contains all of the basic services, supports either one or two CPUs, and has limited Web server capabilities.

Windows 2000 Server provides all the features of the Professional edition but adds an Active Directory controller as well as a full version of Microsoft IIS 5.0.

Windows 2000 Advanced Server extends the Windows Server capabilities with support for as many as four CPUs, up to 64GB of RAM, and load balancing and clustering.

Windows 2000 Datacenter extends services by supporting as many as 32 CPUs together with more advanced clustering.

Protected DLLs

Windows 2000 has introduced a feature known as system file protection that prevents third party applications from replacing any DLLs located in the Windows 2000 system directory.

Originally, DLLs were designed to be extensible such that developers could add new functions and functionality to existing dynamic link libraries as long as the original functionality—and the existing function interface calls—remained unchanged. Likewise, DLLs are supposed to be shared such that any number and variety of applications can call on and use DLL-supplied functions. Therefore, initially, this decision by Microsoft to ‘protect’ system DLLs from change (updating) by third-party applications may appear to be a step backwards.

There is, however, no requirement for third-party applications to install their proprietary DLLs in the system directories (although some do choose to do so), and many DLLs are installed in other directories—such as in the application directory.

Further, since allowing third-party developers to revise and extend system DLLs is a potential hazard in itself, leading to variations and non-standard versions, Microsoft’s decision to impose a blanket protection over the directories becomes more reasonable and understandable.

However, as developers, this restricted access does mean that you should not attempt to install proprietary DLLs in the \WinNT\System32 directory or in any of the Windows subdirectories.

Of course, the corollary imposed by this restriction is that there is no longer a recognized common directory where independent developers can freely install DLLs to be shared among applications. At the same time, older, third-party applications that need to replace or use older system DLLs will not be able to work with Windows 2000, although reports from several testing organizations have not revealed any serious problems.

And there are always workarounds. For example, in Appendix C, where the InstallShield installation services are discussed and demonstrated using the BatteryTest program from Chapter 4, “Windows 2000-Specific Services”, two system DLLs (Mfc40.dll and Msvcrt.dll) are included as part of the program setup package. Since these cannot be written to the \WinNT\System32 directory—even though they may already be present—copies of these two DLLs are written to the BatteryTest directory (by default, x:\Program Files\Reality Engineers\BatteryWatch\).

Granted, this can lead to wasting disk space by duplicating DLLs in different directories and, as developers, it would be wise to add code to setup programs to find out if DLLs already exist in accessible locations before copying duplicates to the application directories. Of course, given the ever-increasing size of disk drives (at ever-lower costs), this is not the critical problem today that it would have been a few years ago.

Programming Cross-System Applications

Although this book is devoted to Windows 2000 programming, you will probably want your applications to run on the other Windows platforms as well as on Windows 2000. As mentioned earlier, most applications written for Windows 2000 will also run on Windows NT, as well as Windows 95/98. I’ll point out the exceptions in later chapters, whenever they apply to the subject at hand. For now, I’ll just provide a general overview of some areas that may be of concern when you are designing cross-system applications.

Sometimes, an application will perform on one system but not on the other because the developers have made use of some tricks that are possible under one operating system but not in the other. In many ways, Windows

98 is still a DOS-based operating system because, even though it is not visible on the surface, extensive support is still provided to ensure that the older DOS and Windows 3.x applications will function under Windows 98. Windows 2000 (and Windows NT), however, avoids the limitations and problems inherent in this schizophrenic approach by simply not supporting the earlier system services.

In other cases, applications may have been written to depend on advanced operating system-specific features. For example Windows NT/2000 supports some features, such as networking APIs, which were not found under earlier operating systems.

The one area where you should be particularly aware of differences is where security and access permissions are used in Windows NT/2000. These permissions are not supported under Windows 98 (or under Windows 3.x or Windows 95).

If you are developing cross-system applications, the only real solution is to periodically test your application on each targeted operating system. Then, if the application functions on one but not on another, that's when the fun begins...and best of luck.

TIP:

The Win32 driver model allows developers to create a single hardware driver that will function on both the Windows 98 and Windows 2000 (NT 5.0) operating systems. Previously, developers were required to write separate and independent drivers for each operating system, making it difficult for both developers and users to upgrade hardware.

Dual-Boot System Operations

To dual-boot or not to dual-boot, that is the question. Whether 'tis nobler to suffer the slings and arrows of a single operating system or...

Okay, with all due apologies to William Shakespeare (the misquote is from Hamlet's soliloquy), the question facing developers more than other people is whether to install a new or developmental OS on a separate machine or to dual-boot two—or more—OS's on a single system.

Of course, if you prefer to keep your platforms 'clean', then you'll have little interest in this topic. On the other hand, if you do not have the luxury of an inexhaustible number of systems (and the space for them), the chances are that you currently are or previously have been interested in a dual-boot or multi-boot arrangement. And, assuming that you are interested in dual-boot, this section will discuss some of the possibilities.

TIP:

Caveat—this is not an exhaustive treatment of multi-boot options; there are simply too many options and possibilities to conveniently discuss all of the choices that are available.

Since this book is about Windows 2000, it seems reasonable to assume that you have or will have the NTLDR (NT loader) as an option for managing a dual-boot system, since NTLDR is distributed with Windows NT and Windows 2000. However, before discussing the supplied boot controller, a brief mention of the alternatives is also appropriate.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Hardware Boot Management

One method of booting multiple operating systems on a single platform is to use two or more removable hard drives, with each hard drive containing a separate OS. Of course, the removable drives can be any size units—as long as they are large enough to contain the OS—while the platform may also contain other fixed drives that are shared among operating systems. If this is your choice of approaches, then you will need to ensure that the shared drive (or drives) is formatted with an appropriate file system compatible with each of the operating systems (see “Choosing a File System,” earlier in this chapter).

BeOS's Multiboot

The Be operating system (BeOS) ships with a multiboot manager which allows you to boot from an operating system installed on any drive or partition on your system. The boot manager offers a simple menu and, regardless of the OS selected, all partitions will be visible after boot...assuming, of course, that the file system on any partition is compatible with the selected OS.

The drawback—speaking from experience—is that the loader is badly documented and difficult to configure. Other than this, it does work but not well enough to earn a strong recommendation.

Power Quest's Boot Magic (a.k.a. Boot Manager)

Power Quest's Boot Magic is an interesting boot manager that functions nicely enough, but that works by 'hiding' one (or more) partitions containing alternate operating systems. Thus, using Boot Magic, each OS must be installed on a separate partition and only the partition containing the selected OS is visible after booting. That is, if DOS, Win98, and WinNT are each installed on separate partitions and DOS is selected at boot time, the partitions containing Win98 and WinNT are simply hidden and completely unavailable.

Of course, any partitions that do not contain operating systems—or, at least, which are not controlled by Boot Magic—remain visible when any OS is booted.

System Commander

System Commander from V Communications is probably the most flexible and a very good choice when a variety of operating systems need to be installed on a single platform. System Commander supports any

combination of PC-compatible operating systems—Windows 3.x/95/98/NT, DOS, OS/2, Linux, etc., but not BeOS—and, unlike Boot Magic, does not hide partitions.

Of course, depending on the OS selected, partitions with incompatible file systems will not be visible. That is, if you have booted to DOS, a Linux or NTFS partition simply will not be recognized.

TIP:

It is possible—although not highly recommended—to have more than one boot loader installed such that one boot loader boots to offer a further choice managed by another loader. Since this is a gray area, careful experimentation may be necessary.

LILLO (Linux Loader)

LILLO is a very simple boot manager that can boot a variety of operating systems. Advantages are that simple script instructions, control choices, and new operating systems can be added conveniently.

NT Loader (NTLDR)

Unlike the preceding loaders, the NT Loader (NTLDR) is not a separate application but is (optionally) installed as part of the Windows 2000 setup and allows you to select between operating systems or between different versions of the same operating system each time the computer boots.

In operation, the NTLDR is controlled by a very simple script in the boot.ini file (in the root directory) that might look something like this:

```
[boot loader]
timeout=10
default=multi(0)disk(0)rdisk(1)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(1)partition(1)\WINNT="Microsoft Windows 2000
Professional" /fastdetect
C:\="Windows 98"
```

Notice that the Windows 2000 installation is identified not by partition, but by the physical disk and partition—rdisk(1)partition(1), which is the first partition on the second physical hard drive—and then by directory. A simple change in the script would change the default boot to Win98, as:

```
default=C:\
```

Also, while not offered by default, additional OS entries could be added to this list.

During installation, if Windows 2000 finds an existing operating system but you choose not to replace the existing system, NTLDR is installed automatically. Dual-boot operations are supported for:

- Windows NT 3.51, Windows NT 4.0
- Windows 95, Windows 98
- Windows 3.1, Windows for Workgroups 3.11
- MS-DOS
- OS/2

To set up a dual-boot configuration, you must use a separate partition for each operating system. During Windows 2000 Setup, you can use the Advanced Setup option to select a folder on an unused partition. Additional information about configuring your system for dual-boot can be found in the Windows 2000 Professional Resource Kit.

WARNING:

Important: It is strongly recommended that you create an Emergency Repair Disk before you install another operating system on your computer.

Setting Up Dual-Boot Using Windows 2000 (NTLDR) Before setting up a dual-boot configuration to make both Windows 2000 and another operating system—such as DOS or Windows 98—available on your

computer, a few precautions are in order:

- Each OS should be installed on a separate drive or separate disk partition.
- Any application software currently installed for an existing OS will need to be reinstalled for Windows 2000. (The reinstallation, of course, may be made to the same directory as the previous installation rather than requiring a completely separate installation.)

WARNING:

If you use a single directory to install an application for two different OSes, you should remember that uninstalling the application from one OS affects both OSes by removing the application from the hard drive but without actually uninstalling the application from the second OS.

- The boot drive for a dual-boot installation should be a FAT or FAT32 configuration.
 - To dual-boot Windows 2000 and Windows 95 or MS-DOS, the primary partition must be a FAT drive.
 - To dual-boot Windows 2000 and Windows 95 OSR2 or Windows 98, the primary partition may be either FAT or FAT32.
- If you install Windows 2000 on a computer that already dual-boots between OS/2 and DOS, the new dual-boot will offer a choice between Windows 2000 and the OS used most recently prior to the installation.
- To arrange a dual-boot between DOS or Windows 95 and Windows 2000, the Windows 2000 installation should be performed last. Otherwise, files required to boot Windows 2000 may be overwritten.
- To dual-boot between Windows 98 and Windows 2000, the operating systems may be installed in any order.
- Because of changes to the NTFS file system, a dual-boot between Windows 2000 and any earlier version of Windows NT is not recommended if only NTFS partitions are used.
- Windows 2000 should not be installed on a compressed drive unless the NTFS file system compression utility was used. While DriveSpace or DoubleSpace volumes can be present on a dual-boot system, these volumes will not be available while running Windows 2000.
- It is possible for Windows 95 or Windows 98 to reconfigure hardware settings upon first use, and these changes may cause problems when dual-booting with Windows 2000.

Other Considerations When Dual-Bootting If you do choose to dual-boot both Windows NT and Windows 2000, each installation of Windows NT Workstation and Windows 2000 Professional must have a different network name.

Often, the biggest conflict between Windows 2000 and Windows 98 or Windows NT is not an incompatibility issue but a simple misunderstanding. On a dual-boot system, an application installed under one operating system may be visible under the other (because all of its files are visible on the hard drive), but it will not run on the second operating system. This is usually not because of incompatibility but because the installation needs to be repeated on the second operating system.

Annoying? Yes, but there's a good reason for this annoyance. When an application is installed under either operating system, the installation does more than simply create a directory and copy some files. Sometimes, the installation also copies .DLL or other special files that are installed in the operating system's \SYSTEM32 directory. But each operating system has its own \SYSTEM32 directory, and, since the contents of different \SYSTEM32 directories are not cross-compatible; they cannot be consolidated. Almost always, during installation, entries are made to the operating system's registry. These are also separate and not cross-compatible.

NOTE:

Some simple applications, which require no setup at all, can be executed on either operating system without installation. In such cases, the applications are usually self-contained and do not rely on DLLs or registry entries or have any necessary DLLs installed in their local directory (see the previous remarks concerning restricted access to system directories).

Fortunately, the solution is relatively simple. If, for example, you are installing Microsoft Visual C++ and you wish to use it for both Windows 2000 and Windows 98 or NT on a dual-boot system, follow these steps:

1. Boot either operating system.
2. Perform the installation normally.
3. Reboot the second operating system.
4. Repeat the installation to the same directory.

A dual-installation does not require duplicate directories and duplicate files, except for any files that are copied to each operating system's \SYSTEM or \SYSTEM32 directory. What is required is to make duplicate registry entries (and you don't want to try to do these by hand, because it just isn't worth the trouble).

After installation, if the application fails to execute on one or the other operating system for some reason, check the application documentation to see if there are some special requirements or system incompatibilities. Then try the installation a second time.

Summary

In this introduction to Windows 2000, we've covered quite a few topics, starting with the hardware requirements for Windows 2000 programming—which, essentially, are simply the platform requirements for running Windows 2000. We've also discussed file systems and potential conflicts between using the FAT16, FAT32, and NTFS file systems, and we've looked at differences between Windows 98, Windows NT, and Windows 2000. Our final topic, in this rather diverse collection of subjects, was dual-boot system installations.

Next, in Chapter 2, we will get down to actual programming, beginning with an overview of Windows programming requirements, Windows messaging, and conventions used in program source code.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

CHAPTER 2

Application Programming for Windows 2000

- “Hello World” in Windows
- Message-mapping functions
- A template for Windows application programs
- Conventions used in Windows programming

For DOS programmers, writing applications for Windows 2000 (or for Windows NT or Windows 95/98) constitutes a distinct and abrupt departure from familiar and accepted practices. In fact, this departure is sometimes so abrupt that many programmers find themselves wondering if they have changed languages as well as operating systems. They are surprised to discover distinctly Pascal-like elements appearing in their C code and to find that previously familiar shortcuts and procedures—such as `main()` or `exit(0)`—are either no longer valid or, in some cases, even fatal to the application or the operating system.

Others, particularly those who have worked with previous versions of Windows, or even with OS/2, may find that these new practices use familiar techniques, although they sometimes bear new names or have somewhat different syntax. Unfortunately, those who have programmed applications for Windows 3.x (either 3.0 or 3.1) may find themselves the most confused of all because many of the differences are slight; yet in many cases, these slight differences are also very critical.

On the other hand, if you are already an experienced programmer and have worked in the Windows 95/98/NT environments, a great deal of programming for Windows 2000 will already be familiar territory.

However, even if you are an experienced Windows application programmer, you should take the time to study programming practices for Windows 2000. Chapter 1, “The Windows 2000 environment,” discussed what was new in Windows 2000, a few aspects of the file systems, and how to set up dual-boot, but this was all background. Now it’s time to look at the components of a Windows program, beginning, as the White King advised Alice, “at the beginning.”

WinHello: An Introductory Windows Program

Traditionally, the introductory C program example has always been a “Hello, World” message, provided by about a half-dozen lines of code. For Windows 2000, however, the equivalent introductory example will be a bit more complex, requiring some 130+ lines of code—roughly 20 times longer. This difference is not so much because Windows is that much more complex than DOS, but more a reflection of the fact that any Windows application, even a rudimentary example, operates in a more complex environment than its DOS counterpart. Accordingly, the Windows application requires some minimal provisions to match this environment.

Still, the flip side of this coin does not mean that all applications will be larger and more complex than their DOS counterparts. More commonly, larger Windows applications will become smaller than their DOS counterparts because many functions and services that are supplied by the application itself under DOS are called externally as system functions under Windows.

NOTE:

Because Windows applications generally include image (bitmapped) resources that their DOS counterparts lack, direct comparisons of .EXE file sizes are generally not valid. The real differences in size are more readily apparent in terms of source code sizes and development times.

Theory aside, however, the *WinHello* source example is considerably larger than its DOS counterpart. Its actual size depends on how the application was created. For example, *WinHello*, a simple application program using no class definitions, consists of one source file; while *WinHello2*, an MFC (Microsoft Foundation Class library) application created using AppWizard, has 20 source files.

The Windows.H Header File

The Windows.H header file is the one essential include file required in all Windows source code. The reason for this is simple: Windows.H contains all of the definitions for Windows messages, constants, flag values, data structures, macros, and other mnemonics that permit the programmer to work without having to memorize thousands of hexadecimal values and their functions.

For earlier versions of Windows, the Windows.H file has been a single, massive text file (about 10,000 lines). With later versions, the Windows.H file has shrunk; it now consists principally of a list of other include files, the most important of which is the WinUser.H include file. This include file is the current counterpart of the older Windows.H file and, like its predecessor, is more than 8,000 lines in length.

NOTE:

The following references to definitions provided by the Windows.H file may most likely be found in either the WinUser.H or WinDef.H files. Any of these references, however, may be located in other sources listed as include files in Windows.H.

If you are using MFC, the Windows.H header is included in the AppWizard-supplied STDAFX.H header as:

```
#define VC_EXTRALEAN          // Exclude rarely used stuff from headers
#include <afxwin.h>           // MFC core and standard components
#include <afxext.h>           // MFC extensions
#include <afxdtctl.h>         // MFC support for IE4 common controls
```

The WinMain Procedure

Just as every DOS C program has a procedure titled `main` at its heart, every Windows program has a similar entry point with the title `WinMain` (and, yes, this title is case-sensitive). Also, just as a DOS C program may include provisions within the `main` procedure declaration to retrieve command-line parameters, the `WinMain` declaration includes a similar provision in the `lpszCmdParam` parameter, even though command-line parameters are rarely used under Windows.

TIP:

When applications are created using the MFC foundation classes and the AppWizard, the `WinMain` procedure will not appear in the skeleton source files created for the various classes. However, the `WinMain` procedure has

not vanished, it is now buried—out of sight and out of mind—inside the precompiled foundation class library. Although invisible, `WinMain` is still present and, if absolutely necessary, can be passed instructions to change the initial style and characteristics of the application.

In contrast to DOS programming, the declarations used for `WinMain` are not optional and must be declared in the exact order and form shown, regardless of whether each specific argument will be used or ignored. Remember, because the application's `WinMain` procedure is being called only indirectly by the user, with the actual calling format supplied by Windows, the calling format flexibility present in a DOS context is absent under Windows.

Also note that the reserved word `PASCAL` is used in all exported function declarations, indicating to the compiler that Pascal rather than C ordering is used for all *arguments* (values) pushed onto the stack. While C commonly uses inverted order, placing the least-significant bytes first on the stack, Windows uses Pascal ordering that, like Unix, places the most significant bytes first.

There is another small but crucial difference. *Exported functions* are functions that will be called from outside the class or—in the case of a DLL— from other applications outside the unit (library). In a Windows application where subroutines are called from Windows itself or where a member function in one class is called from outside the class, even if both belong to the same application, the Pascal calling order is necessary.

On the other hand, all internal function declarations (functions and subprocedures called directly from other procedures within the application) will expect arguments to appear in standard C order and should not be declared using the `PASCAL` specification as this would cause the parameters to be expected in an invalid order.

As far as how argument lists are declared in the procedure definitions, it makes absolutely no difference whether the Pascal or C calling conventions are used. These conventions affect only how the arguments are handled internally—that is, on the stack—and do not in any way affect how the programmer constructs the argument lists.

Of the four calling arguments shown below, the first two are of primary importance. The data type `HANDLE` refers to a 32-bit unsigned value; the `hInstance` and `hPrevInstance` arguments are unique identifiers supplied by the Windows 2000 system. Unlike DOS where only one program is active at a time (TSRs excepted), multitasking systems require unique identification, not only for each application, but also for each instance of an application that may be executing. Therefore, the `hInstance` and `hPrevInstance` parameters are assigned only when an application instance becomes active. They provide the equivalents of the task ID and process ID values common in other multitasking environments.

```
int PASCAL WinMain( HANDLE hInstance,
                   HANDLE hPrevInstance,
                   LPSTR lpszCmdParam,
                   int    nCmdShow )
```

NOTE:

Since the data types used in these declarations may be unfamiliar to DOS programmers, see the introduction to Windows data types later in this chapter for more details. Windows programmers should note that the 16-bit `HANDLE` used in Windows 3.x is now a 32-bit unsigned value, which is a change that affects a number of aspects of Windows 2000 programming.

The `hPrevInstance` (previous instance) identifier is the `hInstance` identifier previously assigned to the most recent instance of an application that is already executing. If there is no previous instance of the application currently running, which is frequently the case, this argument will be `NULL` (0). The reasons for this second process identifier will be demonstrated presently.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The third parameter, `lpzCmdParam`, is a long (FAR) pointer to a null-terminated (ASCIIZ) string containing any command-line parameters passed to the program instance. Although the command-line parameters are provided by Windows rather than by a conventional (DOS) command line, the user can specify these parameters through the Run dialog box invoked from the Start Menu. In general, however, Windows applications rely on dialog boxes for specific input and on .INI entries for default values, rather than expecting command-line parameters.

As a rule, .INI files are obsolete (although they are still used). The fourth calling parameter, `nCmdShow`, is simply an integer argument indicating whether the newly launched application will be displayed as a normal window or initially displayed as an icon. Next, following the procedure declaration itself, a brief list of local variable declarations appears:

```
{
    static char szAppName[] = "WinHello";
    HWND        hwnd;
    MSG         msg;
    WNDCLASS    wc;
```

The `HWND` data type identifies a window handle; `MSG` identifies a message value; and `WNDCLASS` refers to a record structure used to pass a number of values relevant to the application's main window. We'll discuss these data types in more detail later in the chapter.

Registering a Window Class

The first task accomplished within the `WinMain` procedure depends on the `hPrevInstance` argument passed. If a previous instance of this application is already active, there's no need to register the window class a second time. It's more likely, of course, that this is the first instance of the application (`hPrevInstance` is `NULL`) and, therefore, the window class definitions must be assigned and the window class registered.

The `wc` structure is defined in `Windows.H` (which must be included in all Windows applications). Of the `WNDCLASS` record fields, the second (`lpfnWndProc`) and last (`lpzClassName`) are the most important. The remainder of the fields can usually remain unchanged from one application to another. (See the source code example `Template.C` at the end of this chapter for another example.)

The first field is the window-style specification. In this example, it is assigned two style flag values (combined by ORing bitwise). The `CS_` flags are defined in `WinUser.H` as 16-bit constants, each with one flag bit set. Here the `CS_HREDRAW` and `CS_VREDRAW` flags indicate that the window should be redrawn completely anytime the horizontal or vertical size changes. Thus, for the *WinHello* demo, if the window size changes, the window display is completely redrawn, with the “Hello, World” message string recentered in the new display.

```
if( ! hPrevInstance )
{
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;
```

The second field in the `WNDCLASS` structure, `lpfnWndProc`, is a pointer to the exported procedure—`WndProc`, in this example—that will handle all Windows messages for this application. The type prefix `lpfn` identifies this field as a “long pointer to function.” But you should realize that these prefix conventions are provided for the benefit of the programmer. They are not absolutes, nor do these designations place any constraints on the compiler. However, predefined fields and identifiers can be considered an exception. Although you can change these, they are best left as defined, if only for the simple reason that redefinitions could easily result in a cascade of changes and confusion.

The next two record fields are integers, which are reserved to specify extra information about the class or window styles. Commonly, neither is required and, by default, both are initialized as zero (0). (Incidentally, the `cb_` prefix stands for “count of bytes.”)

```
wc.cbClsExtra      = 0;
wc.cbWndExtra      = 0;
wc.hInstance       = hInstance;
```

The next field, `hInstance`, is simply the recipient of the `hInstance` argument passed by Windows when the program is initially called. This field assignment is constant for all applications.

The next three data fields currently assign default values for the application’s icon, cursor, and background color and pattern.

```
wc.hIcon           = LoadIcon( NULL, IDI_APPLICATION );
wc.hCursor          = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground   = GetStockObject( WHITE_BRUSH );
```

The default `IDI_APPLICATION` specification for the icon assigns the predefined image of a white square with a black border. The `IDC_ARROW` cursor assigns the stock cursor graphic of a slanted arrow. In the third assignment, the `hbrBackground` field contains the background color and pattern used for the application’s client region. (The `hbr` stands for “handle to brush,” where *brush* refers to a pixel pattern used to fill or paint an area.)

Next, because this application does not have a menu assigned, the menu name is entered as a null value. The class name (`lpszClassName`) is assigned the null-terminated (ASCIIIZ) string defined previously.

```
wc.lpszMenuName    = NULL;
wc.lpszClassName   = szAppName;
RegisterClass( &wc );
}
```

And, last within this conditional subprocess, the `RegisterClass` function is called with the `wc` structure passed as a parameter (by address) to register this window class definition with the Windows 2000 operating system. As mentioned previously, this registration is required only once. Thereafter, the registration and values assigned are available to any new instances of the application, as long as any instance of the application remains active. Once all instances of the application have been closed, the window class registration is discarded, and any future instance will need to execute the class registration process again.

Creating an Application Window

While the previous step, registering a window class, defined characteristics that are common to all instances of the application, the application window itself still must be created. Unlike the `RegisterClass` function call,

which is called only once, the `CreateWindow` function must be called by every instance of the application to produce the actual window display.

The handle to the application window that is returned by the `CreateWindow` function will be used later as an argument in other function calls; it will be used as a unique identifier for the actual window belonging to the application instance. While many properties of the application class have already been defined, other properties specific to this instance of the application have not; they are passed now as parameters to the `CreateWindow` function.

```
hwnd = CreateWindow(  
    szAppName,           // window class name  
    "Hello, World-Windows 2000 Style",  
                        // window caption
```

The first two parameters passed are the application class name—the same ASCII string that was used when the class was registered—and the application's initial window caption. Of course, the second of these is optional, and, if the window is defined without a caption bar or if no caption is desired, this parameter should be passed as `NULL`.

The third parameter defines the window style and, generically, is passed as `WS_OVERLAPPEDWINDOW`, a value that is a combination of individual flags defined in `Windows.H`.

```
WS_OVERLAPPEDWINDOW,    // window style  
CW_USEDEFAULT,          // initial X position  
CW_USEDEFAULT,          // initial Y position  
CW_USEDEFAULT,          // initial X size  
CW_USEDEFAULT,          // initial Y size
```

The fourth through seventh parameters establish the application window's initial position and size. They can be passed as explicit values or, more often, as `CW_USEDEFAULT`. This parameter instructs Windows to use the default values for an overlapped window, positioning each successive overlapped window at a stepped horizontal and vertical offset from the upper-left corner of the screen.

The next parameter, the eighth, is passed as `NULL` for the simple reason that this application is not associated with a parent window. Alternatively, if this window were to be called as a child process belonging to another application, the parent's window handle would be passed as a parameter here.

```
NULL,                   // parent window handle  
NULL,                   // window menu handle
```

The ninth parameter used in calling the `CreateWindow` function is also passed as `NULL`, directing the application to use the default system menu. Note, however, that the menu in question is the window frame's pull-down menu (upper-left icon on most window frames), not the menu bar (or toolbar), which is defined as an application resource and assigned during the application class registration.

The tenth calling parameter, which can never be passed as `NULL`, is the same instance handle originally supplied by the system.

```
hInstance,              // program instance handle  
NULL );                // creation parameters
```

The final parameter, again `NULL` in this example, may in other cases provide a pointer to additional data for use either by the application window or by some subsequent process. In most examples, however, this will be an empty (`NULL`) argument.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Showing the Window

Now, after CreateWindows has been called, the application window has been created internally in Windows 2000's "world view" but does not yet appear on the actual screen display. Therefore, the next step is to call the ShowWindow function, passing as parameters the hwnd value returned by CreateWindow and the nCmdShow argument supplied when WinMain was initially called.

```
ShowWindow( hwnd, nCmdShow );  
UpdateWindow( hwnd );
```

The ShowWindow function, contrary to what you might assume, does only a portion of the task of creating (painting) the window display. It is principally responsible for creating the window frame, caption bar, menu bar, and minimize/maximize buttons. But what this function does not create is the client window area—the display area specific to the application itself. Therefore, one more function call is necessary before the window display is complete: a call to the UpdateWindow function with the hwnd window handle as an argument. (This call actually posts a WM_PAINT message to the application instructing it to repaint its own window area—a process that will be discussed in a moment.)

This completes the process of registering a window class, defining and creating the window itself, and updating the screen to show the window. Although this is more than a small task, it is only preparation for the application; the real task has not yet begun—but it will momentarily. One last, but very essential, portion of the WinMain function remains: the message-handling loop.

Handling Messages

Windows creates and manages a separate message queue for each active Windows program instance. Thus, when any keyboard or mouse event occurs, Windows translates this event into a message value. This value is placed in the application's message queue, where it waits until it is retrieved by the application instance, which is precisely the purpose of the message-handling loop.

The message-handling loop begins by calling the GetMessage function to retrieve messages from the application instance's message queue. As long as the message retrieved is not a WM_QUIT message (0x0012), GetMessage will return a TRUE (non-zero) result. The actual message value is returned in the msg structure,

which was passed by address.

```
while( GetMessage( &msg, NULL, 0, 0 ) )  
{
```

The syntax for the GetMessage function is defined as:

```
BOOL GetMessage( lpMsg, HWND, wParamFilterMin,  
                wParamFilterMax )
```

In most cases, only the first parameter is actually used to return the message itself. The remaining three parameters are usually passed as NULL or zero.

The initial parameter is a pointer to a message structure to receive the message information retrieved and, subsequently, to pass this data on through to the TranslateMessage and DispatchMessage functions. Without this parameter, there would obviously be little point in calling the GetMessage function at all.

The second parameter is optional but can be used to identify a specific window (belonging to the calling application) and to restrict retrieval to messages that belong to that window. When passed as NULL, as in the present example, GetMessage retrieves all messages addressed to any window belonging to the application placing the call. The GetMessage function does not retrieve messages addressed to windows belonging to any other application.

The third and fourth parameters provide filter capabilities, restricting the message types returned. When both parameters are passed as 0, no filtering occurs. Alternatively, constants such as WM_KEYFIRST and WM_KEYLAST could be passed as filter values to restrict message retrieval to keyboard events or, by using WM_MOUSEFIRST and WM_MOUSELAST, to retrieve only mouse-related messages.

Filters and window selection aside, the GetMessage function (together with the PeekMessage and WaitMessage functions) has another important characteristic.

Conventionally, loop statements monopolize the system until terminated, thus preempting or preventing other operations for the duration of the loop. And, in other circumstances—remember this as a *caution*—even under Windows, loop operations can tie up system resources.

The GetMessage function, however, has the ability to preempt the loop operation to yield control to other applications when no messages are available for the current application, or when WM_PAINT or WM_TIMER messages directed to other tasks are available. Thus, it can give other applications their share of CPU time to execute.

For the present, when the application receives an event message (other than WM_QUIT), the message value is passed. First, it goes to the Windows TranslateMessage function for any keystroke translation that may be specific to the application. Then it is passed to the DispatchMessage handler, where the message information is passed to the next appropriate message-handling procedure (back to Windows, either for immediate handling or, indirectly, for forwarding to the exported WndProc procedure).

```
    TranslateMessage( &msg );  
    DispatchMessage( &msg );  
}
```

Finally, when the message-processing loop terminates, the wParam argument from the final message retrieved is, in turn, returned to the calling application—the system Desktop itself.

```
    return msg.wParam;  
}
```

Messages and Event-Driven Programming

In its simplest form, *message-driven* programming (also known as *event-driven* programming) is a process by which various subprocesses and/or applications communicate. In Windows, messages are the process used by Windows itself to manage a multitasking system and to share keyboard, mouse, and other resources by distributing information to applications, application instances, and processes within an application.

Thus, under Windows, instead of applications receiving information directly from the keyboard or the mouse

driver, the operating system intercepts all input information, packaging this information using the MSG message structure (detailed in the following section) and then forwarding the prepared messages to the appropriate recipients. In turn, the recipient applications use TranslateMessage for application-specific interpretation (particularly accelerator key assignments) before calling DispatchMessage to forward individual traffic items to their appropriate handlers.

Furthermore, the process described is not limited to keyboard and mouse events. Instead, this includes all input devices (including ports), as well as messages generated by application child and subprocesses, Windows timers, or, quite frequently, by Windows itself.

NOTE:

Abstract descriptions of message-driven programming provide only a theoretical outline without really illustrating how these processes function. Therefore, a fuller explanation will be left until subsequent examples in this book can provide both hands-on experience as well as practical illustrations (beginning, of course, with messages processed by the *WinHello* demo).

But before we get to how messages are transmitted, let's take a look at the message record structure and how messages are organized.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Message Record Structure

The MSG (message structure) record type is defined in WinUser.H as:

```
typedef struct tagMSG
{
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
} MSG, *PMSG
```

NOTE:

Both the NEAR and FAR types are essentially obsolete in 32-bit systems. Originally NEAR meant an address within a 16Kb block while FAR indicated an address outside the 'local' 16Kb address space. Today, however, all addresses are NEAR without requiring the specification while specifying FAR is obsolete. Continuing to use FAR, however, has no penalties or conflicts and this term does continue to appear in many struct and function definitions, as discussed previously.

The POINT data type is defined in WinDef.H as:

```
typedef struct tagPOINT
{
    LONG    x;
    LONG    y; } POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
```

The message-event fields defined are used as:

hwnd —The handle of the specific window to which the message is directed.

NOTE:

Note that each application is itself composed of a series of separate windows. These windows include the frame,

the caption bar, the system menu, minimize and maximize buttons, and the application's main display, which is referred to as the *client window* or, occasionally, the *display window*. Normally, only messages directed to the client window will be forwarded by the `DispatchMessage` procedure to the application's `WndProc` procedure. Messages directed to other application windows are generally handled indirectly (by Windows 2000), even though this may result, in turn, in further messages being sent to the client window.

message A 16-bit value identifying the message. Constants corresponding to all message values are provided through `Windows.H` and begin with the `WM_` (which stands for “window message”) prefix. For example, a mouse-button event message might be identified by the constant `WM_LBUTTONDOWN` (left button pressed).

wParam A 32-bit (double word) message parameter. The value, format, and meaning depend on the primary event message type. Various, the `wParam` argument might convey a coordinate point pair, use the low-word value to identify a secondary message type, provide some other type of data, or be ignored entirely. In many cases, the `wParam` value will be treated as two separate word values with different functions.

lParam A 32-bit (long) message parameter. The value and meaning of this parameter depend on the primary event message type. Various, the `lParam` argument might provide a pointer to a string or record structure; break down as a group of word, byte, or flag values; or, quite frequently, be completely unused.

time The double word time value identifies the time the message was placed in the message queue.

pt This field contains the mouse coordinates at the time the message was placed in the message queue (irrespective of the message event type or origin).

Note that these last two fields are not passed to the `WndProc` procedure. Instead, these two fields are used only by Windows, principally to resolve any conflict over the order of events and, of course, to determine where a specific event should be addressed.

The WndProc Procedure

The `WndProc` procedure—by whatever name you prefer—is the point where each application actually begins to function. Remember, the `WndProc` procedure receives messages indirectly from the operating system but determines the application's response to the messages received.

Previously, when the application window class was registered, the address of the `WndProc` subroutine was passed to Windows as:

```
wc.lpfnWndProc = WndProc;
```

And, given this address, Windows calls `WndProc` directly, passing event messages in the form of four parameters, as:

```
long FAR PASCAL WndProc( HWND hwnd,    UINT msg,
                        UINT wParam, LONG lParam )
```

The four calling parameters received correspond to the first four fields of the `MSG` structure described in the previous section, beginning with the `hwnd` parameter identifying the window to which the message is directed. Because most applications have only one client window that will receive the message, this parameter may seem superfluous. This parameter will, however, frequently be needed as an argument for use by other processes.

At the present, the second calling parameter, `msg`, is crucial immediately and identifies the window event message. The third and fourth parameters, `wParam` and `lParam`, provide amplifying information to accompany the window event message.

Typically, the `WndProc` procedure does relatively little or nothing outside of the `switch...case` responding to the `msg` parameter. In the *WinHello* demo, local response is provided for only two event messages: the `WM_PAINT` and `WM_DESTROY` messages. All other event messages are handled by default (by the operating system).

The first of these two, `WM_PAINT`, is a message that is generally not issued directly. It will be issued indirectly anytime an application window is created, moved, resized, restored from an icon, or uncovered by a change in some other application window. It will also be issued indirectly when something occurs (in this or in some other application) to invalidate the client area of the present application.

The DOS equivalent of the *WinHello* program would consist principally of a print statement, possibly with an optional clear screen statement. For the Windows version, however, there are two main reasons for the differences:

- The response to the WM_PAINT message is not a one-time occurrence.
- A bit more is accomplished than simply dumping the text to the screen.

Before anything can be written to the client window, the first requirement is for the application to retrieve a handle (hdc) to the *device context* (the output device or, in this example, the screen). After the screen update is finished, this handle will be released by calling the EndPaint function.

```
switch( msg )
{
    case WM_PAINT:
        hdc = BeginPaint( hwnd, &ampps );
        GetClientRect( hwnd, &amprct );
```

After retrieving the device context handle, the GetClientRect procedure is called to retrieve the rect structure with coordinates describing the client window. The rect structure consists of four fields, which report coordinates for the client window. However, the coordinates reported are relative to the client window itself. Therefore, the *left* and *top* fields are returned as zeros, and the *right* and *bottom* fields return the current width and height of the client window (reported in pixels).

Once the window coordinates have been retrieved, the rect structure can be used as an argument in the next step to specify the region where the actual message will be drawn.

```
DrawText( hdc, "Hello, World!", -1, &amprct,
          DT_SINGLELINE | DT_CENTER | DT_VCENTER );
```

Since print statements, *per se*, cannot be used in Windows (because they are unsuited for a graphics display environment), the DrawText function is used instead. DrawText begins with the hdc argument providing access to the active display, followed by the string (text) to be drawn.

The third parameter, -1, indicates that the string argument is a null-terminated string. Alternatively, this parameter could be a value specifying the string length, with the second parameter an indirect reference to a character array.

The fourth argument is the address of the rect structure, identifying an area where the string will be drawn. The fifth argument is a combination of flags that set alignment and restrict the text drawn to a single display line. Other elements affecting the display (such as font, size, and color) use the system default settings—although these factors are subject to change, as you will see in future demos.

NOTE:

The `sprintf` statement can still be used to format text to a buffer array, but direct screen print statements are disallowed.

Last, the EndPaint function is called, again with the client window handle and the paint structure (ps) as arguments. This function releases the device context and validates the now-restored client area, and incidentally, completes the response to the WM_PAINT message.

```
EndPaint( hwnd, &ampps );
return( 0 );
```

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The second application message requiring a local response is the WM_DESTROY message, which is issued when the application is ready to close. This message can be generated via several channels, as will be shown later, but for our example, it is issued only if or when the system menu Close option is selected.

```
case WM_DESTROY:
    PostQuitMessage(0);
    break;
```

The WM_DESTROY message is issued to give the application an opportunity to do any necessary cleanup before shutting down. Therefore, as circumstances demand, the application response at this point could include provisions for calling a dialog box to request confirmation, for closing or saving files, or for any other final tasks required for a smooth exit.

Finally (unless, of course, termination is to be aborted), the WM_DESTROY response is completed by calling the PostQuitMessage function that, in turn, places a WM_QUIT message in the application's message queue to terminate the message loop in WinMain.

Explicit handling has been provided for only two of the messages that might be sent to this application. As a default case, provisions are also required to return all messages that have not been explicitly handled here to Windows for processing.

```
default:                                // if message unprocessed,
    return(                             // return to Windows
        DefWindowProc( hwnd, msg, wParam, lParam ) );
}
return( NULL );
}
```

This default provision returns the message—precisely as it was originally received—to Windows, and then it also returns the result from DefWindowProc to the Windows calling process. This final provision should be considered standard for all WndProc message-handler procedures.

Message Loop Alternatives

A message loop is integral to all Windows applications, but the message loop demonstrated in the *WinHello* application is only one format, while MFC supports a second format employing message-mapping macros. In virtually all cases, the ClassWizard is used to create message-map entries that, in effect, link message commands to the message-handling functions.

Under conventional C/C++ programming, a similar process was possible using CALLBACK functions. However, MFC's message mapping offers advantages for many functions by customizing the parameter passed to the function handling the response. For example, when the OnDraw function is called in response to a WM_PAINT message, the OnDraw function is passed a pointer to the device context as an argument, making it unnecessary to call the BeginPaint()/EndPaint() functions. Similarly, when the OnHScroll() function is called in response to a scrollbar event, parameters are passed specifying the type of event and the thumbpad position. In conventional coding, this information would still be available but would require decoding before use.

Message-mapped functions will appear in many of the sample programs later in this book, but, for a very brief example, a message map containing an ON_COMMAND macro might look something like this:

```
BEGIN_MESSAGE_MAP( CMyDoc, CDocument )
//{{AFX_MSG_MAP( CMyDoc )
ON_COMMAND( ID_MYCMD, OnMyCommand )
    ON_COMMAND( ID_FILE_OPEN, OnFileOpen )
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    ON_WM_RBUTTONDOWN()
    ON_WM_RBUTTONUP()
    ON_WM_SIZE()
// ... More entries to handle additional commands
//}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

In this example, the ID_MYCMD event message is directed to the OnMyCommand() procedure, which will provide the response action. Other ON_COMMAND macro entries would be used to handle command messages generated by menus, buttons, and accelerator keys.

Likewise, the ID_FILE_OPEN event message is sent to the OnFileOpen procedure. The ON_WM_LBUTTONDOWN event and the following examples are other Windows event messages mapped to specific procedures that have been named and created—as skeletons—by the ClassWizard.

In each of these cases, the programmer must still provide the response to the message—within the identified procedure. But the message maps generated by the ClassWizard are a convenient replacement for the conventional message-handling loops, which can often be quite cumbersome.

WARNING:

While message-map macros are important, there is no need to write these directly. Instead, use the ClassWizard to automatically create message-map entries in the application source files by associating message-handling functions with messages. Always use the ClassWizard to add, edit, or remove message-map entries. You should also be aware that manually editing the message map could quite easily cause your app to fail unceremoniously, offering little or no clue as to the cause of the malfunction. This is not to say that you cannot edit the message map, only that you must do so in the full knowledge of what you are doing and why ... and, of course, you must also check the results very carefully.

The .DEF (Define) File

For a Windows program, the .C source code is only a part of the story. In most cases, the application will also incorporate a .H header file and, almost always, a .RES resource file. These two elements will be discussed in later chapters. For the present, however, there is one more source file that all Windows application sources require, without exception.

When a program is compiled, the compiler processes each .C or .CPP source file (and any included .H header files) to produce an .OBJ (object) file bearing the same name. Subsequently, the linker combines .OBJ and .LIB (library) files to produce the executable program.

For DOS applications, this would be essentially all that's required. For Windows applications, however, the

linker also expects a .DEF (definition) file.

TIP:

Applications or DLLs produced using classes defined with the `AFX_EXT_CLASS` declaration may omit the .DEF definition files. The `AFX_EXT_CLASS` declaration marks the entire class as exported and, therefore, available for linking by other applications.

This definition file consists of simple ASCII text, but it contains an essential series of instructions for the linker.

```
=====;
; WinHello module-definition file ;
; used by LINK.EXE ;
=====;

NAME            WinHello
DESCRIPTION     'Hello, World ... Win2000 Style'
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       5120
```

From the top, the .DEF file begins with an application name and a brief description, both of which are optional and could be omitted (however, including a name and description is recommended for clarity and to alleviate future confusion). The third line, `EXETYPE`, is essentially a binary flag stating either that this is intended to be a Windows executable (`WINDOWS`) or a dynamic link library (`DLL`).

The fourth line, `STUB`, specifies the inclusion, during link, of the `WINSTUB.EXE` file. The stub program is simply a brief executable that, if the compiled application is called from DOS, displays a warning message stating that the application can be run only from Windows. Or you might design your own stub program to be referenced as `'MYSTUB.EXE'`.

The fifth and sixth lines provide flags identifying how the code and data segments should be treated during execution. Customarily, both the code and data are defined as `PRELOAD` (load immediately) and `MOVEABLE` (relocatable in memory). Also as a default, the code segment is normally defined as `DISCARDABLE`, permitting the memory used by the code segment to be overwritten when memory resources become limited. (Of course, discarding the code segment also means that the application will need to be reloaded from disk before execution can continue.) The `MULTIPLE` specification for the data segment permits separate data for each active instance of an application. Alternatively, using a `SINGLE` data specification forces data to be shared between multiple instances.

Next, the `HEAPSIZE` and `STACKSIZE` specifications provide default heap and stack memory requirements. The values shown are the suggested defaults and should serve most applications.

The preceding statements can be considered defaults suitable for virtually all applications, and they can be used as a template for future use. The final statements, however, are far more application-specific and, equally, far more important to the application's compiling, linking, and executing correctly.

Last, but certainly important—and perhaps also the least familiar element in the .DEF file—is the `EXPORTS` statement, together with its subsequent list of export labels.

```
EXPORTS
    WndProc
```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Unlike DOS applications, where the application itself is responsible for all calls to internal procedures, Windows applications depend on exporting principal subprocedures and placing these entry points under the control of the operating system (such as Windows 98), where each will respond to control messages sent by the operating system. For this indirect control to function, the link process must ensure that the addresses of these procedures are known and available; ergo, each must be explicitly exported, which is accomplished by references in the .DEF file.

In this example, only one procedure is exported, WndProc, which provides the principal control structure for the application and forms the absolute minimum required by any Windows application (although the name used may be any valid label desired). However, it is not unusual for applications to have several, dozens, or in very complex situations, even hundreds of exported procedures.

The WinHello.DEF and WinHello.C files provide the minimum necessary to compile an executable Windows program.

TIP:

Using the MFC, the .DEF define files can be omitted entirely, as will be shown in later examples.

For future applications, this chapter provides a second program, which uses a slightly different approach. The next example is provided as a generic template for your own application development and also to demonstrate a second style for application development.

The *Template* Program: A Template for Application Development

The *Template* example, like *WinHello*, is a simple but functional Windows application. But unlike *WinHello*, *Template* has been designed specifically for use as a template for your own application development. The *Template.I* include file has also been used as the foundation for a number of the demo applications appearing in this book.

TIP:

If you are using the MFC and the AppWizard provided with the Developers Studio, you will not need nor want the *Template.I* include file, which is the core of the *Template* demo. The *Template* demo is included simply for

Unlike familiar DOS programs, which generally have little, if any, source code in common (other than the main procedure), most Windows applications have a number of elements in common:

- The same WinMain procedure can be used by hundreds of programs, remaining unchanged except for a few string labels identifying the application by name.
- Essentially the same WndProc procedure can be used over and over. In this area, the sample provided by the *Template* application will need to be expanded and altered to provide the specific needs of each separate application.
- Although the About dialog box used in the *Template* application may not be satisfactory for all purposes, it does provide an example for use in constructing and programming generic dialog boxes.

The Template.C Source Code

The Template.C source code begins with two include statements, referencing the Windows.H header first and then the Template.H header.

```
#include <windows.h>
#include "template.h"
```

Notice that the template.h include statement uses quotation marks rather than angle brackets (<>). This identifies the include file as one located locally, as part of the application source code, rather than a system file in the Developers Studio include directory.

```
#define APP_MENU    "TemplateMenu"
#define APP_ICON    "Template"

HANDLE hInst;
char    szAppTitle[] = "Application Template";
char    szAppName[]  = "Template";
```

In the *WinHello* program, only one string identifier was declared—in the WinMain procedure for the szAppClass—and all the other string references were entered directly as required.

For the *Template* application, two strings and two defines are declared, global to the entire program. This format was chosen for two reasons:

- The instructions referencing these strings are contained in an include file, not in the main source file.
- This provides a convenient means to change these lines to match other applications without needing to search through the entire program for their occurrence.

In some cases, the menu and icon names could also be declared as string variables. This format uses #define statements because there are circumstances where a NULL argument may be needed instead of strings.

In later examples, additional references will appear, similar to one of these two declaration styles, and generally for the same reasons.

The balance of the Template.C source code is quite brief and contains only three functions: WinMain, WndProc, and AboutProc. The latter two are exported procedures and are declared as such in Template.DEF.

The first of these, WinMain, is much briefer than its equivalent in the *WinHello* demo, even though both accomplish the same tasks. In *Template*'s version of the WinMain procedure, however, the provisions required to initialize the application class and to initialize application instances have been transferred as independent subprocedures to the Template.I include file. The second procedure, WndProc in this example, provides a skeletal structure for application message handling. In this example, only a few message-response provisions are included: the WM_COMMAND and IDM_ABOUT subcommand messages, and the WM_DESTROY message.

The third procedure, AboutProc, parallels the WndProc procedure in many respects but provides message handling for the About dialog box.

The Template.I File

The Template.I include file contains the two subprocedures mentioned earlier. These are the InitApplication

procedure, which initializes the application class (if this task has not already been accomplished), and the `InitInstance` procedure, which initializes each instance of the application. The operations and functions provided by both of the functions are essentially identical to the operations described earlier for the *WinHello* program.

NOTE:

This `Template.I` include file will be referenced by many of the application examples found in this book.

The `Template.H` Header File

The *WinHello* demo program did not require a `.H` header file, but the *Template* demo (and every other demo in this book) does require a header. In this case, this requirement is dictated by the need to define a new, application-specific message value. Remember, the `Windows.H` (or the `WinDef.H`) header supplies the stock definitions used, but values for any messages that are not already provided must be defined in the application's header file, where these definitions can be accessed by both the Resource Editor(s) and the C compiler.

```
//=====//
//  Template.H header file  //
//=====//
```

```
#define IDM_ABOUT 100
```

In this case, only one message value is required: `IDM_ABOUT`. However, in later examples, much longer lists of defines will be quite common. Also, in most cases, there is no need to worry about value conflicts between values defined in the application header and values defined in the `Windows.H` header because these will be used in different contexts.

It may be useful, however, to avoid values less than 100. Using higher values will avoid confusion with some common standard button IDs and will help you to group values as much as is practical for your own clarity.

Also, many programmers prefer to include forward function declarations in the application header, as shown in the following code. However, including these declarations is optional, and unless required by your own organization and function ordering, they may be omitted entirely.

```
//=====//
// forward function declarations (optional) //
//=====//
```

```
BOOL InitApplication( HANDLE );
BOOL InitInstance( HANDLE, int );
long FAR PASCAL WndProc( HWND, UINT, UINT, LONG );
BOOL FAR PASCAL AboutProc( HWND, UINT, UINT, LONG );
```

The `Template.RC` Script

The `Template.RC` script provides the basic elements required for the *Template* application. We'll discuss application resources, resource files, and resource scripts in detail in later chapters.

The *Template* application requires three resources: a menu bar (`TemplateMenu`), a dialog box (`AboutDlg`), and an icon image (`template.ico`). Notice also that the `Template.RC` script includes a reference to the `Template.H` header file.

NOTE:

In the following resource script, note the Pascal-like elements alluded to at the beginning of this chapter—particularly the `BEGIN` and `END` statements.

```
//=====//
//  Template.RC Resource Script  //
```

//=====//

```
#include "windows.h"
#include "template.h"
```

```
TemplateMenu MENU
BEGIN
    POPUP "&Help"
    BEGIN
        MENUITEM "&About Template...", IDM_ABOUT
    END
END
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

This section of the Template.RC script creates a simple menu bar with one pull-down menu titled Help (see Figure 2.1) with a single item, About Template..., which returns the command message value defined by IDM_ABOUT. Of course, most application menus are considerably more complex. We'll get to these complexities in later chapters.

The IDM_ABOUT command message returned from the menu calls a dialog box that is also described in the resource script, as shown in the following lines:

```
AboutDlg DIALOG 22, 17, 144, 75
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About Template"
BEGIN
    CONTROL "The Template application provides", -1,
        "STATIC",
        SS_CENTER | WS_CHILD | WS_VISIBLE | WS_GROUP,
        14, 7, 115, 8
    CONTROL "a generic template for designing", -1,
        "STATIC",
        SS_CENTER | WS_CHILD | WS_VISIBLE | WS_GROUP,
        14, 18, 115, 8
    CONTROL "Win2000 applications.", -1, "STATIC",
        SS_CENTER | WS_CHILD | WS_VISIBLE | WS_GROUP,
        14, 29, 115, 8
    CONTROL "OK", IDOK, "BUTTON", WS_GROUP, 56, 50, 32, 14
END
```

The AboutDlg dialog box described also appears in Figure 2.1.



FIGURE 2.1 The About Template dialog box

Last, the resource script includes a reference to an .ICO image file, which contains the application icon.

Template ICON “template.ico”

Because binary images are not readily adaptable to the present format (the printed page), you will need to use an icon editor to create a sample icon image with the Template.ICO file name or select an appropriate image from any of the many available sources.

The complete *Template* program, including the Template.C, Template.DEF, Template.H, Template.BAT, and Template.RC files, is on the CD that accompanies this book.

Windows Conventions and Data Types

The following sections describe some Win2000 conventions for naming, as well as some of the Windows data types, data structures, and handle identifiers.

Variable Names and Hungarian Notation

As programs have become more complex both in terms of size and of the proliferation of data types, many programmers have adopted a variable-naming convention, which is commonly referred to as *Hungarian notation* (apocryphally named in honor of a Microsoft programmer, Charles Simonyi).

NOTE:

Over the past several years, several “standard” versions of Hungarian notation have been proposed and/or published. The version given here is dictated in part by personal preferences and in part by conventions established by Windows in naming constants, variable, and data structure definitions. Because all of these standards are intended to be mnemonic for your convenience, you may follow or alter them as desired.

Using Hungarian notation, variable names begin with one or more lowercase letters that denote the variable type, thus providing an inherent identification. For example, the prefix *h* is used to identify a handle, as in *hWnd* or *hDlg*, referring to window and dialog box handles, respectively. In like fashion, the prefix *lpsz* identifies a long pointer to a null-terminated (ASCIIZ) string. Table 2.1 summarizes the Hungarian notation conventions.

TABLE 2.1: Hungarian Notation Conventions

Prefix	Data Type
b	Boolean
by	byte or unsigned char
c	char
cx / cy	short, used as size
dw	DWORD, double word or unsigned long
fn	function
H	handle
i	int (integer)
l	long
n	short int
p	a pointer variable containing the address of a variable
s	string
sz	ASCIIZ null-terminated string
w	WORD, unsigned int
x, y	short, used as coordinates

Predefined Constants

Windows also uses an extensive list of predefined constants that are used as messages, flag values, and other operational parameters. These constant values are always full uppercase and most include a two- or

three-letter prefix set off by an underscore. Here are some examples:

CS_HREDRAW	CS_VREDRAW	CW_USERDEFAULT
DT_CENTER	DT_SINGLELINE	DT_VCENTER
IDC_ARROW	IDI_APPLICATION	WM_DESTROY
WM_PAINT	WS_OVERLAPPEDWINDOW	

In the case of constant identifiers, the prefixes indicate the general category of the constant. Table 2.2 shows the meanings of the prefixes in the examples shown here.

TABLE 2.2: A Few Constant Prefixes

Prefix	Category
CS	Class style
CW	Create window
DT	Draw text
IDC	Cursor ID
IDI	Icon ID
WM	Window message
WS	Window style

Data Types

Windows also uses a wide variety of new data types and type identifiers, most of which are defined in either the WinDef.H or WinUser.H header files. Table 2.3 lists a few of the more common data types.

TABLE 2.3: A Few Windows Data Types

Data type Meaning	
FAR	Same as far. Identifies an address that originally used the segment:offset addressing schema. Now FAR simply identifies a (default) 32-bit address but may be omitted entirely in many cases.
PASCAL	Same as pascal. The pascal convention demanded by Windows defines the order in which arguments are found in the stack when passed as calling parameters.
WORD	Unsigned integer (16 bits).
UINT	Unsigned integer, same as WORD.
DWORD	Double word, unsigned long int (32 bits).
LONG	Signed long integer (32 bits).
LPSTR	Long (far) pointer to character string.
NEAR	Obsolete, previously identified an address value within a 16Kb memory block.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Data Structures

Similarly, Windows adds a variety of new data structures. Again, most are defined in either WinDef.H or WinUser.H. The five examples shown in Table 2.4 appear in the *WinHello.C* program.

TABLE 2.4: Five Windows Structures

Structure	Example	Meaning
MSG	msg	Message structure
PAINTSTRUCT	ps	Paint structure
PT	pt	Point structure (mouse position)
RECT	rect	Rectangle structure, two coordinate pairs
WNDCLASS	wc	Window class structure

Handle Identifiers

In like fashion, a variety of handles are defined for use with different Windows elements. Like constants, the handle types use all uppercase identifiers. Table 2.5 shows a few examples.

TABLE 2.5: Five Handle Identifiers

Handle type	Examples	Meaning
HANDLE	hnd or hdl	Generic handle
HWND	hwnd or hWnd	Window handle
HDC	hdc or hDC	Device context handle (CRT)
HBRUSH	hbr or hBrush	Paint brush handle
HPEN	hpen or hPen	Drawing pen handle

You'll see these conventions used throughout this book. All of the examples demonstrated, including the *WinHello* and *Template* programs, are found on the CD accompanying this book.

Summary

While this chapter concentrating on the basics of Windows programming may be familiar territory to many, experience has shown that there are always those who need a refresher or who are new to the requirements imposed by a multitasking environment. Thus, in this chapter, we've used the traditional "Hello, World" application, in two different forms, to introduce the basics required by any application.

We've also discussed Windows messages and message-mapping functions at some length, and offered a template for Windows applications for those who prefer not to use the AppWizard-generated templates. And, finally but not least, we've looked at the notational conventions that are commonly used in Windows programming.

Next, in Chapter 3, we'll take a further look at the NTFS file system and examine some of the possibilities and options provided by NTFS, as well as its limitations and restrictions.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 3

The Windows NT File System (NTFS)

- Characteristics of NTFS
- Non-NTFS file systems (FAT and VFAT)
- NTFS file attributes and querying file/drive information
- Multiple data streams in NTFS and the *NTFS_Streams* demo

Over the years, a number of different file systems have been in use on various computer platforms. For a very long time, for PCs, the standard (and default) file system was the DOS file allocation table (FAT) system. When OS/2 was introduced back in '89, the FAT file system was supplanted by the high-performance file system (HPFS) and, with the introduction of Windows NT 3.1 in the early 90s, a second new file system appeared as the New Technology file system (NTFS).

NOTE:

For a long time, Microsoft denied that the NT in Windows NT actually stood for anything in particular and speculation (fanciful and otherwise) assigned a variety of meanings to the initials, including: NoThing, New Thing, Not Telling, Not True, and other less complimentary terminology. More recently, however, the words New Technology have appeared as a seemingly tacit admission that this is the actual source of the NT acronym.

More recently, the FAT32 file system—supported by Windows 95B and Windows 98—has appeared, but the differences between FAT (FAT16) and FAT32 are related mostly to storage processes while remaining pretty much transparent to applications. NTFS is also generally transparent to applications but, unlike the FAT16/FAT32 systems, offers several extensions or features that were not found in previous file systems.

While these features may or may not be relevant to your application (many are security- and access-related), knowledge of what features are available is still valuable, whether or not you choose to employ them.

Thus, we will begin this chapter by looking at the characteristics and features supported by NTFS, the limitations under NTFS, and finally, at a few of the functions provided—both Windows- and NT-specific

functions—that provide access to NTFS files.

Characteristics of NTFS

NTFS provides support for object-oriented applications by treating files as objects possessing both user-defined and system-defined attributes. At the same time, NTFS offers all of the capabilities found in the FAT and HPFS file systems but avoids many of the limitations of the earlier file systems.

In like fashion, NTFS is designed to be a fully recoverable file system. That is, NTFS is designed to recover from system crashes, CPU failures, or I/O errors by restoring consistency to the drive volume without resorting to the AutoChk, ChkDsk, or ScanDisk commands, each of which can require lengthy operations while recovering after a system failure.

TIP:

Both the AutoChk, ChkDsk, and ScanDisk utilities are provided under Windows NT with support for NTFS volumes and these may be used if the normal recovery fails or if corruption outside the control of the file system occurs.

NTFS provides a variety of features that are not found in FAT/FAT32 or HPFS volumes, including access security, automatic creation of MS-DOS (8.3) aliases, multiple data streams, and Unicode filenames. These features also offer POSIX functionality. (POSIX is a collection of international standards originally designed for UNIX-style operating systems; POSIX compliance is required by U.S. government standards.)

NOTE:

NTFS cannot be applied to floppy disks or other removable media.

NTFS Filenames

Under NTFS, filenames can include as many as 256 characters (including a terminating null). Where filenames are not MS-DOS compatible, NTFS also generates an 8.3 format filename, allowing DOS-based or 16-bit Windows applications to access the same files.

NOTE:

While an NTFS volume may be “invisible” and inaccessible to many non-Windows NT operating systems, NTFS files can be copied to non-NTFS volumes or to removable media where non-NT systems can access these files. (However, NTFS volumes may be visible to other network machines regardless of the OS used; Linux has the capability of reading an NTFS volume directly.)

When the NTFS directory name or filename contains spaces and fewer than eight characters, NTFS does not create an 8.3 filename. To translate a long-format filename into the 8.3 format, the following rules are applied:

- Any illegal characters and any spaces from the long filename are deleted, including: “/\[]:;=, ”
- Any excess periods—except for the extension delimiter—are removed. Any valid, nonspace characters following the final period are left. For example, the filename: “This is a really long filename.123.456.789.TXT” becomes “THISIS~1.TXT”.
- If the filename contains a final trailing period, this period is ignored and the next-to-last period is used to create the 8.3 filename. For example, the filename: “This is a really long filename.123.456.789.” becomes “THISIS~1.789”.
- Filename extensions longer than three characters are truncated.
- All letters in the filename and extension are converted to uppercase.
- The filename is truncated, if necessary, to a six-character length before a tilde (~) and a number are appended. In this fashion, each unique filename ends with ~1 while duplicate (8.3 format) filenames are created as ~2, ~3, etc.

When multiple duplicate filenames are created, the sequence of numbers does not necessarily reflect the order of the original files, only the order in which the 8.3 filenames were created.

Limitations on NTFS Volumes

While both MS-DOS and Windows 3.x applications can run under Windows NT and, therefore, have access to an NTFS volume, file operations from either type of application can produce unexpected results. For example, many older applications commonly save output as temporary files before deleting the original file and, finally, renaming the temporary file with the original filename.

However, because the DOS or Win 3.x applications use only the 8.3 filenames, the original long filename is lost during this process. Further, any unique permissions assigned to the original file are lost when the original file is deleted, while the newly renamed file does not have any permissions assigned. (Permissions can be reset from the parent directory.)

Also, like HPFS, NTFS organizes data on hard disks but not on floppy disks. Any files written to floppy disks (or other removable media) are written as FAT files—that is, files without security permissions or extended attributes. However, depending on the media (such as CDs, and Iomega Zip and Jaz drives), long filenames will still be supported.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Master File Table

Every directory, subdirectory, and file in an NTFS volume is recorded in a special file called a *Master File Table (MFT)*. The first 16 records in the table are reserved for special information, with the first of these describing the MFT table itself and followed, in the second entry, by a record of a mirrored MFT table. In this fashion, if the initial MFT record is corrupted, NTFS reads the second record to locate the mirror MFT table whose first record is identical to the first record of the original.

As further security, the locations of the data segments for both the MFT and MFT-mirror files are recorded in the boot sector while a duplicate of the boot sector is found at the logical center of the disk.

The third record in the MFT is the log file, which is used for file recovery, while the seventeenth and subsequent records are devoted to the files and directories on the NTFS volume.

Beginning with NT 4, each MFT entry is 1Kb (replacing the earlier and larger MFT entry format) and each entry will contain file attributes such as the entry type, the filename, and security and permission attributes. If the data contained by the file is small—or, of course, if the entry is a directory entry—the entire data for the file may also be contained in the MFT entry (as resident data) as shown in the following graphic:



If the data is too large, then the data is written elsewhere on the disk as a series of non-resident contiguous blocks while the Master File Table record receives a record of the data's location. The effect of this approach is to provide rapid file access through the NTFS file system.

For example, consider the mechanisms used by the FAT/FAT32 file system. FAT entries list the names and addresses of files and directories but these entries only refer to items listed in the root directory. (The root directory can contain only a limited number of entries.)

To locate a file in a subdirectory, the operation begins by reading the file allocation table to locate the directory (or subdirectory) entry, then follows a chain of allocation units to find the directory. Then, within the directory data, a further subdirectory or file entry needs to be found that provides additional directions to the file.

By contrast, with NTFS, all entries are in the MFT (the MFT grows as required) and access to the file data is more immediate. For a small directory, for example, the information (linking to other MFT entries for

specific files) is contained directly in the MFT entry itself. For a large directory, the MFT entry might point to a non-resident data block—as a B-tree data structure—which would contain indexes for further MFT entries for specific files.

The point, of course, is simply that the chain of allocation units using NTFS/MFT is shorter than a corresponding structure using FAT.

Non-NTFS File Systems

While it might be convenient to assume that Win32-based applications are likely to encounter only NTFS or FAT/FAT32 file systems, the actual fact is that, depending on the configuration of a computer, applications may require access to volumes using any of several different file systems, including:

- FAT file system
- Protected-mode FAT file system (VFAT)
- NTFS

NOTE:

HPFS support, which was available in NT 4.0, has been dropped in Windows 2000.

Since NTFS is the principal topic for this chapter and has already been covered in some detail, there is little point in recapping NTFS now. The remaining three file systems, however, do deserve some mention and explanation.

The FAT File System

Even on computers using NTFS, the FAT file system will still be used to organize data on floppy disks. While the FAT file system is not as sophisticated as the file systems used on hard drives today, Windows NT supports what is known as FastFat, and Windows 95/98 supports VFAT, and both of these versions of the FAT file system do support long filenames.

The main advantage of FAT volumes is that they are accessible by DOS and Windows systems for floppy disks and other removable media. The major disadvantages of a FAT volume (previous to Windows 95 and Windows NT 3.51) are the limitations on the length and content of filenames, and the lack of Unicode support.

Valid FAT filenames take the following form:

`[[drive:]][[directory\]]filename[.extension]`

The drive parameter specifies the name of an existing drive—a letter from A through Z—and must be followed by a colon (:).

The directory parameter specifies the directory containing the file and is separated from the drive designation and the filename by a backslash (\). The directory parameter must be fully qualified (it must include the names of all directories in the file's path) if the desired directory is not the current directory. A FAT directory name can consist of any combination (up to eight characters) of letters, digits, or these special characters: \$ % ' _ @ { } ~ ` ! # ().

A directory name can also have an extension (up to three characters) of any combination of letters, digits, or the previously listed special characters. The extension is preceded by a period.

Valid filenames can include embedded spaces, with the restriction that the spaces be preceded and followed by one or more letters, digits, or special characters. Thus “file 1” is a legal file name. However, FAT volumes are not case-sensitive (unlike UNIX volumes) and do not distinguish between upper- and lowercase letters. This means that the name “FILE 1” is the same as “file 1” or “FiLe 1” when accessed from a FAT volume.

The Protected-Mode FAT File System (VFAT)

The VFAT file system organizes data on fixed and floppy disks just like the FAT file system. It is also compatible with the FAT file system, using file allocation tables and directory entries to store information about the contents of a disk. VFAT supports long filenames by storing these names and other information,

such as the date and time the file was last accessed, in the FAT structures.

Directory paths are allowed with lengths up to 260 characters (including a terminating null character) while filenames can consist of as many as 255 characters (also with a terminating null). Remember, however, that the path information includes the filename.

NTFS File Attributes

NTFS treats each file, directory, or subdirectory as a set of file attributes which include elements such as the name of the file, the file's security attributes and access permissions, and, of course, the file's data. Each of these attributes is identified by an attribute type code and, in some cases, an attribute name.

These attributes may be either resident or nonresident; the difference lies in whether the attributes can be written within the MFT file record—in other words, resident—or, if they are too large for the MFT file record, non-resident.

Some elements are always included in the MFT file record, including the filename, time stamps, and security attributes. The non-resident attributes are allocated one or more runs of disk space—outside of the MFT records—on the NTFS volume. (A “run” of disk space is any contiguous linear area on the disk.)

Table 3.1 lists all of the file attributes currently defined by NTFS. However, since additional file attributes may be added in the future, this list should not be considered permanent.

NOTE:

All attributes, whether resident or nonresident, can be referenced as a string of bytes.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 3.1: NTFS File Attribute Types

Attribute type	Description
Standard Information	Includes time stamps, link count, and so on.
Attribute List	Lists all other attributes in large files only.
Filename	A repeatable attribute for both long and short filenames. The long name of the file can be up to 255 Unicode characters. The short name is the MS-DOS-readable, eight-plus-three, case-insensitive name for this file. Additional names, or hard links, required by POSIX may also be included as additional filename attributes.
Security Descriptor	Shows information about who can access the file, who owns the file, and so on.
Data	Contains file data. NTFS allows for multiple data attributes per file. Each file typically has one unnamed data attribute. In addition, a file can have one or more named data attributes, using a particular syntax.
Index Root	Used to implement directories.
Index Allocation	Used to implement directories.
Volume Information	Used only in the volume system file and includes, among other things, the version and name of the volume.
Bitmap	Provides a map representing records in use on the MFT or directory.
Extended Attribute Information	Used by file servers that are linked with OS/2 systems. This attribute type isn't useful to Windows NT.
Extended Attributes	Used by file servers that are linked with OS/2 systems. This attribute type isn't useful to Windows NT.

Querying File/Drive Information

The *NTFS_Ops* demo program (included on the CD) is written more for demonstration purposes than for any practical applications. The objective, of course, is to demonstrate how to access file information that is

available not only on NTFS drives but on other drives on the system as well. Naturally, some of the information reported will be NTFS-specific. While queries to non-NTFS volumes may or may not respond with complete information, the *NTFS_Ops* program should query all types of drive volumes without conflicts.

NOTE:

Drive types and differences between these are discussed later in this chapter, together with the `GetVolumeInformation` function demo.

The *NTFS_Ops* demo is a dialog-based application, allowing us to show a lot of information concerning the select file and drive in a formatted presentation, as shown in Figure 3.1.

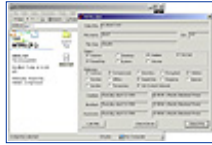


FIGURE 3.1 Examining file characteristics

Here both the Windows Explorer and the *NTFS_Ops* demo are showing information for the same file, *Test.txt*, on an NTFS volume. If you look carefully, you may note a few discrepancies between the two reports—discrepancies that we’ll discuss in a moment.

In the `CNTFS_OpsDlg` class, the first step—using the *Select File* button—is to invoke the common file dialog (using the `CFileDialog` class) as:

```
void CNTFS_OpsDlg::OnSelectFile()
{
    CFileDialog    *pCFD;
    CFile          cf;
    CFileStatus    cfStatus;
    CString        csSize;
    int            nLen1, nLen2;
    DWORD          dwFAttr;

    pCFD = new CFileDialog( TRUE );
    pCFD->m_ofn.lpstrTitle =
        "Select a drive or file to examine";
    if( pCFD->DoModal() == IDOK )
    {
```

Since the common file dialog should be familiar to all programmers (because this is the standard for file selection), we won’t bother discussing the dialog or the `CFileDialog` class at this point except to observe that a file selection is required.

When the dialog returns—assuming that a file selection has been made—we can query a number of elements directly from the dialog, including the complete drive/path/filename string, the file attributes (or some of them anyway), the filename and file extension, and the file size information.

```
    m_csPathFile = pCFD->m_ofn.lpstrFile;
    m_csDriveID = m_csPathFile;
    m_csDriveID = m_csDriveID.Left(
        m_csDriveID.Find( "\\\" ) + 1 );
    m_csFileName = pCFD->GetFileTitle();
    m_csFileExt  = pCFD->GetFileExt();
```

The common file dialog has given us information about the file selection, but we need to call the `CFileStatus` `GetStatus` function to query the attributes for the selected file:

```
    cf.GetStatus( m_csPathFile, cfStatus );
    csSize.Format( "%d", cfStatus.m_size );
```

The status information returned includes several attribute flags, the file size, and the Create, Modify, and

Access Date/Time stamps for the file.

Although this is most of the information that would be desired commonly, we're going to use one more query—by calling the `GetFileAttributes` function—to ask for some further information.

```
dwFAttr = GetFileAttributes( m_csPathFile );
```

Now that we have the raw information, before going further, we'll take a few lines of code to format the file size information in an intelligible fashion:

```
nLen1 = nLen2 = csSize.GetLength();
while( nLen1 > 3 )
{
    nLen1 -= 3;
    csSize.Insert( nLen1, "," );
}

if( nLen2 > 9 )                // 1024^3
    m_csFileSize.Format( " %s (%d Gb)",
        csSize, cfStatus.m_size / 1073741824 );
else
if( nLen2 > 6 )                // 1024^2
    m_csFileSize.Format( " %s (%d Mb)",
        csSize, cfStatus.m_size / 1048576 );
else
if( nLen2 > 3 )                // 1024^1
    m_csFileSize.Format( " %s (%d Kb)",
        csSize, cfStatus.m_size / 1024 );
else
    m_csFileSize.Format( " %s bytes", csSize );
```

Rendering larger file sizes as gigabytes, megabytes, or kilobytes isn't just a matter of presentation or style but is mostly a matter of making the information intelligible for the user. In this case, this may not seem to be greatly important, but it is a good practice to follow in all cases.

The next step is to 'decode' the attribute information returned by the six flags `CFileStatus::GetStatus` function as:

```
//=== attribute ===
m_cb_readonly.SetCheck(
    cfStatus.m_attribute & 0x01 );    //  readonly
m_cb_hidden.SetCheck(
    cfStatus.m_attribute & 0x02 );    //  hidden
m_cb_system.SetCheck(
    cfStatus.m_attribute & 0x04 );    //  system

m_cb_volume.SetCheck(
    cfStatus.m_attribute & 0x08 );    //  volume
m_cb_directory.SetCheck(
    cfStatus.m_attribute & 0x10 );    //  directory
m_cb_archive.SetCheck(
    cfStatus.m_attribute & 0x20 );    //  archive

if( cfStatus.m_attribute == 0x00 )    //  no flags
    m_cb_normal.SetCheck( TRUE );
```

Of course, if none of these six are set, the file can be flagged as 'normal'.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

We also have three date/time stamps available: the creation date/time—in other words, when the file was first created—the date/time of the last modification, and the last access date/time stamps. These are presented as formatted strings, thus:

```
m_cs_Accessed =  
    cfStatus.m_atime.FormatGmt( "%#x" );  
m_cs_Created =  
    cfStatus.m_ctime.FormatGmt( "%#x" );  
m_cs_Modified =  
    cfStatus.m_mtime.FormatGmt( "%#x" );  
  
m_cs_Access_Time =  
    cfStatus.m_atime.FormatGmt( "%X    (%z)" );  
m_cs_Create_Time =  
    cfStatus.m_ctime.FormatGmt( "%X    (%z)" );  
m_cs_Modify_Time =  
    cfStatus.m_mtime.FormatGmt( "%X    (%z)" );
```

If you look back at Figure 3.1, you should notice that the Modified timestamp reported by Windows Explorer (left) does not match the timestamp reported by NTFS_Ops dialog (right). The discrepancy is found in the fact that Explorer is reporting the timestamp as Pacific Daylight Time while the dialog is showing the true timestamp in Zulu or GMT time, even though we're reporting that the local time zone is Pacific Standard.

While this is not precisely a flaw, you should be aware of how date/time information is actually recorded on a volume—that is, in GMT or Zulu time—while Explorer (or other file utilities) covert this data to local date/time information.

Earlier, after reading the standard attribute information reported through the common file dialog, we mentioned that we would also query additional attribute information, which is supported by an NTFS volume (but not necessarily by other drive formats), and did so by calling the `GetFileAttributes` function.

Although we've already queried the information, we still need to 'decode' the flag bits returned in the `dwFAttr` variable.

In theory, the `dwFAttr` variable can contain some 13 flags that are identified by constants defined in the `WinNT.H` header. We stress ‘in theory’ because in practice, not all of the defined constants corresponded to the actual flags returned by the `GetFileAttributes` function. (Or, if you prefer, the actual flags returned by the operating system.)

WARNING:

Since this book is being written using a beta version of Windows 2000, the flags returned by the final version may change, or a revision of the `WinNT.h` header may be released to correct the discrepancy. In any case, you should check any code using these flags for accuracy.

The first five flags correspond to five of the standard attributes previously reported by the `CFileStatus::GetStatus` call.

```
/*
#define FILE_ATTRIBUTE_READONLY                0x00000001
#define FILE_ATTRIBUTE_HIDDEN                 0x00000002
#define FILE_ATTRIBUTE_SYSTEM                0x00000004
#define FILE_ATTRIBUTE_DIRECTORY             0x00000010
#define FILE_ATTRIBUTE_ARCHIVE               0x00000020
*/
    m_cb_faReadOnly.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_READONLY );
    m_cb_faHidden.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_HIDDEN );
    m_cb_faSystem.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_SYSTEM );
    m_cb_faDirectory.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_DIRECTORY );
    m_cb_faArchive.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_ARCHIVE );
```

The sixth flag, however, is distinctly at variance with the actual flag value returned.

```
/*
#define FILE_ATTRIBUTE_ENCRYPTED                0x00000040
*/
// FILE_ATTRIBUTE_ENCRYPTED defined in WinNT.H does not
// match flag value
//    m_cb_faEncrypted.SetCheck(
//        dwFAttr & FILE_ATTRIBUTE_ENCRYPTED );
//    m_cb_faEncrypted.SetCheck( dwFAttr & 0x4000 );
```

Since the defined flag is `0x0040` and the returned flag is `0x4000`, there is some reason to suspect an error in the `WinNT.h` header but this is difficult to prove or disprove. In any case, the value `0x4000` is the flag value for an encrypted file and the test in the *NTFS_Ops* demo has been written accordingly.

Unlike the `GetStatus` query previous, when the Normal flag had to be inferred by the absence of other flags, the `GetFileAttributes` query actually returns a ‘normal’ flag.

```
/*
#define FILE_ATTRIBUTE_NORMAL                0x00000080
#define FILE_ATTRIBUTE_TEMPORARY             0x00000100
*/
    m_cb_faNormal.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_NORMAL );
    m_cb_faTemporary.SetCheck(
        dwFAttr & FILE_ATTRIBUTE_TEMPORARY );
```

The last set of attribute flags are not reported for conventional (FAT) volumes but are relevant for NTFS volumes:

```

/*
#define FILE_ATTRIBUTE_SPARSE_FILE            0x00000200
#define FILE_ATTRIBUTE_REPARSE_POINT         0x00000400
#define FILE_ATTRIBUTE_COMPRESSED            0x00000800
#define FILE_ATTRIBUTE_OFFLINE               0x00001000
#define FILE_ATTRIBUTE_NOT_CONTENT_INDEXED   0x00002000
*/

m_cb_faSparse.SetCheck(
    dwFAttr & FILE_ATTRIBUTE_SPARSE_FILE );
m_cb_faReparse.SetCheck(
    dwFAttr & FILE_ATTRIBUTE_REPARSE_POINT );
m_cb_faCompressed.SetCheck(
    dwFAttr & FILE_ATTRIBUTE_COMPRESSED );
m_cb_faOffline.SetCheck(
    dwFAttr & FILE_ATTRIBUTE_OFFLINE );
m_cb_faContent.SetCheck(
    dwFAttr & FILE_ATTRIBUTE_NOT_CONTENT_INDEXED );

```

Last, after going through the final five flags, the dialog is updated to display the results of the queries.

```

        UpdateData( FALSE );
    } }

```

Since the purposes behind these last five flags are a little beyond the topic for this chapter, we won't attempt explanations at this time. Instead, before continuing with the next topic and looking at how different types of file system can be identified, a brief review of typical file systems would be appropriate.

Querying File Systems

We've described four file systems (volumes) that could be encountered on a computer system (as local hard drives/logical drives) and briefly mentioned some of the characteristics of each. The fact that we have variations in file systems means that we need some way to distinguish between these several file systems.

For this purpose, we have the `GetVolumeInformation` function, which is demonstrated in the *NTFS_Ops* demo as shown in Figure 3.2.



FIGURE 3.2 Querying the file system

The `GetVolumeInformation` function returns information about the current volume, such as the file system type (System Name), the maximum length of filenames (Max Component Length), and a series of flags identifying which capabilities are supported by the system.

The `GetVolumeInformation` function is called as:

```

BOOL GetVolumeInformation(
    lpRootPathName,           // LPCTSTR
    lpVolumeNameBuffer,       // LPCTSTR
    nVolumeNameSize,          // DWORD
    lpVolumeSerialNumber,     // LPDWORD
    lpMaximumComponentLength, // LPDWORD
    lpFileSystemFlags,        // LPDWORD
    lpFileSystemNameBuffer,   // LPSTR
    nFileSystemNameSize )     // DWORD

```

The `lpRootPathName` argument is a string identifying the root directory of the volume to query or, if `NULL`, the root of the current directory is queried.

On return, `lpVolumeNameBuffer` contains the name of the disk volume—the label (if any) assigned to the drive—while `nVolumeNameSize` is the length, in characters, of the volume name buffer but can be ignored if the volume name buffer is not supplied.

lpVolumeSerialNumber returns with the volume serial number or can be a NULL parameter if the serial number is not needed.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The `lpMaximumComponentLength` parameter returns the maximum length, in characters, of a filename component (the part of the filename between the backslashes) supported by the specified file system. You should note, however, that this does not change the maximum file/path length—in other words, reporting a maximum component length of 255 characters does not mean that the total length of 260 characters for the complete file/path has changed.

Most important, the `lpFileSystemFlags` argument contains a series of bit flags associated with the specified file system. Flags reported may include:

- `FS_CASE_IS_PRESERVED`: The file system preserves the case of filenames when it places a name on disk.
- `FS_CASE_SENSITIVE`: The file system supports case-sensitive filenames.
- `FS_UNICODE_STORED_ON_DISK`: The file system supports Unicode in filenames as they appear on disk.
- `FS_PERSISTENT_ACLS`: The file system preserves and enforces access-control lists (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.

More immediately useful, the `lpFileSystemNameBuffer` parameter returns with the name of the file system (FAT, HPFS, or NTFS) identifying the file system used on the queried volume. (Perhaps redundantly, the `nFileSystemNameSize` reports the length, in characters, of the file system name buffer.)

Universal Long Filenames

Win32 applications, especially those running under Windows 2000, can potentially read and write files to several different file systems. The following brief guidelines can assure filename compatibility with all file systems supported within Windows 2000.

WARNING:

These guidelines do not, of course, cover networked drives where potential file systems include BeOS, Macintosh, Linux/Unix, and others.

Any application following these guidelines can create valid names for files and directories regardless of the file system in use.

- Filenames can use any character in the current code page except for a path separator, characters in the range 0 to 31, or characters specifically disallowed by the file system. Filenames may include characters from the extended character set: 128 to 255.
- Path specifications may use either the backslash (\) character, the forward slash (/) character, or both, as component delimiters.
- The current directory can be represented as a directory component in a path using a period (.) character.
- The parent of the current directory can be represented as a directory component in a path using two consecutive period (..) characters.

Long Filenames and Novell NetWare

NT workstations and NT servers are often connected to Novell networks where other servers and/or workstations may include Macintosh or other operating systems. While later versions of Novell NetWare have no conflicts with long filenames, versions earlier than 3.12 require specifically enabling Macintosh or HPFS name-spaces. This process does involve costs on the server-side (RAM), and applications must use NetWare-specific name space APIs to access long-format filenames.

Before invoking special provisions, however, Win32 applications can inquire to determine if specific volumes support long filenames and, if so, can simply use these without worrying about the server system.

However, applications using a real-mode Novell client or a server that has not been enabled for long filenames must rely on 8.3-format filenames.

Multiple Data Streams in NTFS

One of the big differences found in an NTFS volume is support for multiple data streams. Essentially, a data stream is simply a named data attribute in a file.

In Figure 3.2 the report shows a file—Test.txt—which has a reported size of 0 bytes. In actual fact, however, this file has two “hidden” data streams whose contents are not reported as part of the file size.

Likewise, if this file were copied to any non-NTFS file system (such as a floppy disk using FAT or another drive using FAT32, etc.), the named data streams would not be copied although the default and unnamed data—presently 0 bytes—would be.

In Figure 3.3, the file Test.txt has been revised to include unnamed data with instructions for reaching the named streams.

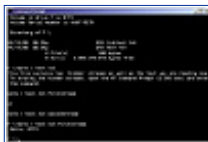


FIGURE 3.3 A file with named streams

Following the instructions, the named stream “firststream” is invoked to show the contents while the second named stream can be invoked in a similar fashion.

Creating a Multistream File for Demonstration Purposes

Since the CD accompanying this book is not an NTFS volume, the file Test.txt cannot be included for demonstration purposes. You may, however, create a simple multistream file of your own by following these steps:

1. Open the command prompt
2. Log in to an NTFS volume
3. At the prompt, type: `n:\>echo "Default data stream" > test.txt`
4. Type: `n:\>echo "this is a hidden stream" > test.txt:stream1`
5. Type: `n:\>echo "this stream is also hidden" > test.txt:stream2`

The resulting file will now contain two named (hidden) streams as well as the default data stream. For an alternative, refer also to the following section, "The *NTFS_Streams* Demo."

Multiple data streams can be used for a variety of purposes. (While concealment might be one of the more obvious uses one would consider, this is really a secondary effect.) The primary use is to allow related data to be managed as a single unit—on Macintosh systems, for example, a similar structure is used for managing resource and data forks.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

More immediately, an application using NTFS might employ an alternate stream to track a list of changes to the primary data, thus keeping archive information together with the primary version of the file. Alternately, streams could be used to store multiple versions of an image, or to store remarks or notes (as text) together with an image, audio, or other data file.

NOTE:

Under NTFS, the stream name identifies a new data attribute in the file. Each stream has separate opportunistic locks, file locks, allocation sizes, and file sizes, but file sharing is on a per file basis, not per stream.

There are no provisions, *per se*, for identifying which files have named streams or for identifying the stream names within a file. Instead, applications creating or using such streams are expected to 'know' the names of any streams they are permitted access to.

The *NTFS_Streams* Demo

Although you can create streams from the command prompt, this is not how you would normally want to create a stream. In using streams in applications, it is rather obvious that another approach is required.

Creating a stream is simple. For example, when opening a file for output, the filename is specified as:

```
[drive:][pathname\]filename.ext
```

In like fashion, to open a stream for output, the stream name is specified as:

```
[drive:][pathname\]filename.ext:streamname
```

And, of course, opening a stream for read is accomplished in exactly the same manner.

The *NTFS_Streams* demo (shown in Figure 3.4) is a very simple example that allows you to name a file, to optionally name a stream, and to enter a brief text message to be written to the stream—or to the file proper, if no stream name is provided. Once this is done, other options permit the stream or the entire file to be read, displayed, or deleted.

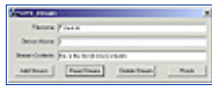


FIGURE 3.4 A simple stream demo

The *NTFS_Streams* program is quite simple in application. The [Add] button, for example, is used to create a file or a file with a named stream as:

```
void CNTFS_StreamDlg::OnAdd()
{
    CFileException e;
    CString csFileName, csMsg;

    UpdateData( TRUE );
    if( m_csFileName.IsEmpty() )
    {
        MessageBox( "A filename is required,\n"
                    "The stream name is optional", "Error" );
        return;
    }
}
```

If you haven't specified a filename, the subroutine begins with a simple warning. But, assuming the minimal requirement has been met, the next step is to check for a stream name and, if one is found, to call *CheckDrive* to ensure that the operation is being carried out on an NTFS drive.

The *CheckDrive* subroutine will attempt to separate a drive specification from the filename, and it then uses the result to call *GetVolumeInformation*. Or, if there is no drive specification, *GetVolumeInformation* queries the present default drive—the root of the active directory.

```
csDrive = m_csFileName;
csDrive = csDrive.Left(
    csDrive.FindOneOf( "/" ) + 1 );
GetVolumeInformation( csDrive, NULL, 0, NULL, NULL, NULL,
    {LPSTR}(LPCTSTR) csSysName, 30 );
if( csSysName == "NTFS" ) return TRUE;
```

We don't really care about most of the volume information—hence the NULLs—only the system name. If the system name is returned as "NTFS", then the subroutine returns TRUE and we can proceed.

Therefore, assuming the target is an NTFS volume, the program continues by concatenating the filename and stream name with a colon between them.

```
csFileName = m_csFileName;
if( ! m_csStreamName.IsEmpty() )
{
    if( ! CheckDrive() ) return;
    csFileName += ":";
    csFileName += m_csStreamName;
}
```

At this point, *csFileName* should contain either a simple filename (with the optional drive/path specification) or a filename:stream combination. Which doesn't really matter because our call to the *CFile::Open* function is the same in either case.

```
if( m_cf.Open( csFileName, CFile::modeCreate |
    CFile::modeNoTruncate |
    CFile::modeWrite, &ampe ) )
{
    if( ! m_csContents.IsEmpty() )
    {
        m_cf.Write( m_csContents,
            m_csContents.GetLength() );
    }
}
```



```
        ...  
    }  
    m_cf.Close();  
}  
}
```

And that's basically it—once the file is opened, the content is written to the file or to the named stream and we close the file.

The provisions in the CNTFS_StreamDlg::OnRead function are virtually identical except for the fashion in which the file is opened—that is, for read rather than for write. If an application needed to access both the primary file data (the unnamed stream) and one (or more) named streams at the same time, all that would be required would be to treat each as if it were a separate file, performing separate open operations for each and then reading or writing from each.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The provision in the *NTFS_Streams* demo for deleting a stream—or for deleting the file—is slightly different from the preceding example, as shown by the following:

```
TRY
{
    CFile::Remove( csFileName );
}
CATCH( CFileException, e )
{
    csMsg.Format( "%s could not be deleted",
                  csFileName );
}
END_CATCH
```

Here there is no real need to open the file or stream since we can simply invoke the `CFile::Remove` function. If the argument is a named stream, then only the stream is deleted and the remainder of the file is left as it was—including any other named streams. Alternately, if the argument simply names the file, then the entire file is deleted, including any named streams.

So that's basically it—named streams are treated exactly the same as any other files except for the filename nomenclature. And this brings us to the final topic for this chapter on NTFS file operations: linking files.

Linking Files under NTFS

Another feature found on NTFS file systems is the ability to create links between files—or, more accurately, to create a hard link between an existing file and a new file with a different name. An NTFS hard link is similar to a POSIX hard link.

Hard links provide multiple names for a single underlying file. Essentially, all of the names point to a single file and modifications to the linked file are also performed on the original (and vice versa). However, each 'filename' can have separate security attributes and permissions.

TIP:

Creating a hard link is demonstrated in the *NTFS_Ops* program—use the Link File button—by first requesting a name for the link file and then invoking the `CreateHardLink` function.

The `CreateHardLink` function is called with three parameters as:

```
BOOL CreateHardLink(
    LPCTSTR lpFileName,
    LPCTSTR lpExistingFileName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes );
```

The `lpFileName` argument specifies the new file to be linked to the original identified by the `lpExistingFileName` argument.

Optionally, the `lpSecurityAttributes` argument is a pointer to a `SECURITY_ATTRIBUTES` structure providing a security description for the new file and determining whether child processes can inherit the returned file handle. If null, the new file receives a default security descriptor and the handle cannot be inherited.

In the *NTFS_Ops* demo, the `OnLinkFile` procedure, which follows, demonstrates creating a link:

```
void CNTFS_OpsDlg::OnLinkFile()
{
    CString    csMsg;
    CLinkDlg *pLD = new CLinkDlg();

    if( m_csPathFile.IsEmpty() )
    {
        MessageBox( "A source file must be selected first",
                    "Attention" );
        return;
    }
    pLD->m_csFileName = m_csPathFile;
    if( pLD->DoModal() == IDOK )
    {
        if( ! CreateHardLink( pLD->m_csLinkName,
                             pLD->m_csFileName, NULL ) )
        {
            csMsg.Format(
                "%s could not be created as a linkfile",
                pLD->m_csLinkName );
        }
        else
        {
            csMsg.Format( "%s linked to %s",
                          pLD->m_csFileName,
                          pLD->m_csLinkName );
            MessageBox( csMsg, "Attention" );
        }
    }
}
```

If you create a file with named streams using the *NTFS_Streams* demo, you can then change to the *NTFS_Ops* demo and link the file to a new file name. Finally, if you return to the *NTFS_Streams* demo, you should be able to view the same streams (and contents) from either file. Also note that changes made to either of the linked files will be reflected in the other.

Finally, as one last experiment, try deleting one of the linked files and then observe the results on the other.

Summary

This chapter has provided an overview of the NTFS file system along with comments on differences from other common file systems and a few notes on creating file and directory names that are acceptable across a variety of file systems.

Probably more important, from a programmer's standpoint at least, we've looked at file access (with an

emphasis on NTFS volumes) and the attribute flags that may be assigned to files. We've also looked at how to identify the characteristics of a file system—the file system identifier, and what features are supported by the file system.

Finally, we've devoted a little time and a couple of demo programs to operations that are specific to NTFS: named streams in files, and linking files.

While none of these may be exceptionally relevant to the average application, simply knowing that these capabilities are available and how they function may suggest circumstances where one (or more) of these features might be the ideal solution to a problem.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 4

Windows 2000–Specific Services

- Multiple monitor support
- NTFS changes
- Power management
- The Windows 2000 kernel
- The Windows 2000 user interface/GDI

While multiple monitor support was originally introduced with Windows 98, the use of multiple monitors is the exception rather than the rule. Further, like Windows 98, Windows 2000 also supports multiple monitors and it appears safe to say that their use will become more common in the future.

However, given the past history and prevalence of the single monitor system, providing support for multiple monitors—or even anticipating the use of multiple monitors—is not common practice among programmers. Thus, we do not plan to discuss programming *per se* for multiple monitors, but a discussion of the principles and uses for multiple displays is appropriate.

WARNING:

Because this book is being written prior to the final release of Windows 2000, some features present in the Beta and Release Candidate versions may be absent in the final release version. While multiple monitors have been supported in early versions of Windows 2000 (and have worked quite well) there have been rumors suggesting that this feature may be dropped. However, no reasons for such a change have been offered.

Multiple Monitor Support

The first requirements for using a multiple display system, rather obviously, are to have more than one monitor and also more than one video card—that is, one card for each monitor, and vice versa.

You should be aware, however, when installing a second video card in a system, that the results may not be precisely as expected. For example, if you install a PCI video card on a system where a high resolution AGP video card is already installed, when the system is powered up, the PCI video card could become the default, with the AGP video card—initially, at least—ignored.

TIP:

On systems providing an AGP slot, the BIOS will usually offer an option to select which video card is activated—AGP or PCI—upon bootup. However, the default is usually PCI. Of course, if only one video card is installed, selection is handled automatically by the BIOS.

For this reason, if the existing video resolution and the resolution of the second, newly installed video card are drastically different, you may find that the video mode Windows attempts to apply to the new primary display is not initially compatible with the hardware. The solution, of course, is to reboot in Safe Mode and reset the resolutions appropriately.

TIP:

A few manufacturers are already offering multiple monitor video cards, and many more are likely to appear as users discover the convenience and advantages of multiple monitor desktop real estate.

How Multiple Monitors Are Used

The support provided by Windows 2000 for multiple monitors is to treat two or more monitors (or, more accurately, the video cards) as part of a single virtual desktop. That is, when two (or more) displays are used on a single system, Windows treats the total display space as a single contiguous display.

Of course, the relationship between the displays is not fixed—that is, Windows 2000 allows the user to configure the relationships of multiple monitors by separately defining the size of each and their relative positions. In this fashion, two monitors might be positioned side by side with the virtual display extending across the pair or the two monitors might be at different levels with the positions of each offset vertically as well as horizontally. For an example of how two physical monitors are oriented, see Figure 4.1, where the primary monitor appears at the left and the secondary monitor higher and to the right.

The dialog shown appears only after a second video card is installed. Here, the Display list shows the identified video card(s) for selection and the settings for each can be adjusted separately.

TIP:

Since the system does not know the actual orientation of the monitors, responsibility for setting the relative positions lies with the user. However, setting the display positions at odds with the physical positions can be extremely confusing and is not recommended.



FIGURE 4.1 Orienting two monitors in a virtual desktop

Characteristics of a Virtual Desktop

In a virtual desktop—that is, a system using multiple monitors—one monitor is always the primary display. Rules applied on a virtual desktop are:

- The primary display always has the 0,0 coordinates at the top left corner.
- Secondary monitors may have negative coordinates—that is, a display located to the left and above the primary might have its upper-left origin point identified in negative coordinates as -1024,-768. In Figure 4.1, the secondary monitor (upper right) has an origin point (top-left corner) located at 1024,-488.
- The system metrics—the system video characteristics reported by the `GetDeviceCaps` function—always

refer to the primary display device.

- Although two display units can be offset, displays are not permitted to overlap and no gap is permitted between displays. That is, two units must have common borders which can be:
 - Right and left edges
 - Top and bottom edges
 - Diagonal corners
- While wallpaper is duplicated on each display unit, your screensaver will blank (cover) only the primary display, leaving secondary displays on the virtual desktop unaffected.
- The mouse, rather essentially, can move freely between display units *wherever they share a common border*.
- Maximized windows occupy a single monitor but cannot extend across multiple monitors. (Non-maximized windows, however, are not subject to this restriction.)
- Menus do not extend across monitors and all child windows appear on the same monitor as the parent application.

TIP:

Screen capture utilities may or may not be able to capture images from secondary monitors.

Remote Displays

As an alternative to a single virtual display, multiple monitors can be used as duplicate screens so that each monitor shows the same display. In this fashion, one (or more) monitors could be used for a remote display or used to drive a video projector. Using this capability, a person doing a presentation would be able to concentrate their attention on a regular monitor while the audience would be watching the same display on a large screen.

Displaying Multiple Applications

The big advantage in using a virtual desktop, obviously, is in having additional screen “real estate.” For example, you could display a word processor on one screen, an Internet browser on a second, and a spreadsheet on a third—or, of course, any combination of open windows on any of the monitors.

Compatibility Issues

On a virtual desktop, most applications are perfectly functional and treat the virtual desktop as a single large display. In some cases, however, applications may have difficulties with multiple monitors. Compatibility issues include:

- Some applications may be unable to accept negative coordinates.
- Some applications may restrict display or center themselves only on the primary display.
- Direct hardware access may be incompatible with a virtual desktop (does not apply to DirectX drivers).
- Patches to the GDI or DISPLAY drivers may be incompatible with a virtual desktop.

For the most part, compatibility issues with virtual desktops are minor and may simply result in some applications that restrict themselves to the primary display or that can only be used on the primary display. However, if serious incompatibilities are encountered, as a last-ditch solution, it may be necessary to disable secondary monitors.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

NTFS Changes

Under Windows 2000, NTFS (discussed in Chapter 3, “The NTFSFile System”) now includes support for features such as the Change Journal, reparse points, sparse files, encryption, volume quotas, native structured storage, and volume mount points, as well as supplying improved defragmentation tools.

The Change Journal

The Change Journal is a new feature in Windows 2000’s NTFS and provides a mechanism for tracking additions, deletions, and changes to files. New APIs allow applications to view volume changes without resorting to name space traversal, while the Change Journal makes it possible for storage applications to efficiently determine changes in a given name space. As an example, a backup application might refer to the Change Journal to create a list of files prior to an incremental backup.

Reparse Points

Reparse points are file system objects provided by NTFS that offer a new mechanism for file and directory enhancements when used with file system filter drivers provided by an independent software vendor (ISV). Reparse points are used to support and enable new classes of storage management tools as, for example, remote storage systems.

In operation, reparse points are objects within the NTFS file system that contain specialized tags and provide triggers to extend the I/O subsystem. These extended capabilities are provided through installable file system filter drivers, where each filter driver is associated with a unique reparse point tag.

The intention behind the introduction of reparse points is to allow ISVs to have a consistent mechanism for creating and implementing extended storage functionality within the Windows 2000/NTFS system and to relieve developers of the need to design proprietary systems in order to extend value-added functionality.

At the same time, Microsoft is introducing reparse point-based features for Windows 2000 using installable file system filter drivers, including:

- Encrypted file systems (EFS)
- Native structured storage

- NTFS directory junctions
- NTFS mount points (volume mount points)
- Remote storage servers

For ISV-provided file system extensions, Microsoft is responsible for assigning unique reparse point identifiers. When a reparse point attribute is encountered while resolving a pathname, NTFS transfers responsibility for resolution to the vendor-specific file system filter driver, which assumes the task of executing specific I/O functionality.

TIP:

An Installable File System (IFS) Kit is available (see www.microsoft.com/hwdev/ntfskit/default.htm) and provides detailed documentation and source code for writing installable file system filter drivers.

Sparse Files

Sparse data is data that contains large consecutive blocks that are essentially empty, consisting only of 0s. Using sparse file support, a user (or administrator) is able to mark such data files as sparse so that the file system will allocate space only for the meaningful portions of the data. In a sparse file system, the file system stores range information that describes where the empty data would have been if it had been allocated. In turn, when one of the empty blocks is accessed, by default, the file system returns an appropriate block of zeros. The advantages in storage efficiency, of course, should be obvious.

TIP:

Sparse file support complies with the C2 security requirement specifications.

Encryption

One of the file system filter drivers identified by reparse points under Windows 2000/NTFS is the Encrypting File System (EFS) provided by Microsoft. EFS is a driver allowing data to be encrypted for storage and decrypted on retrieval, while the Win32 APIs and utilities have been enhanced to support encrypted files. EFS supports both file- and directory-level encryption.

Although NTFS alone provides C2-level security for files and directories on NTFS volumes, this does not prevent a physical storage volume from being removed and mounted on another system. Once the volume is mounted, the system administrator would be able to assume ownership of all data, in effect simply bypassing the NTFS security—a vulnerability that applies particularly to laptops.

With the addition of the Encrypting File System, Windows 2000 makes it both possible and convenient for data to be stored in encrypted format such that physical removal and remounting will no longer permit security to be bypassed.

EFS encryption keys are generated on a per domain, per user basis. To allow data to be recovered in the case of a forgotten password or in the event of a terminated employee's absence, a recovery key is also generated that is accessible to administrators.

Currently, EFS implements a 128-bit version of the DESX cryptography scheme, with a 40-bit scheme for international configurations. However, the EFS architecture allows for future versions of Windows 2000 to offer other encryption schemes.

TIP:

For more information on EFS, refer to the white paper "Encrypting File System for Windows 2000", available at www.microsoft.com/ntserver/. Also refer to: www.microsoft.com/Windows/server/technical/security/encrypt.asp.

Volume Quotas

Volume quotas or *disk quotas* are a new feature in NTFS and provide for granular control of network-based storage. Volume quotas permit both hard and soft storage limits to be implemented, and provide system administrators the means for managing storage growth—in distributed environments—by setting usage quotas under NTFS on a per volume basis. Quotas are managed by reporting the available disk space according to a

user context where the user can be defined as either a domain user or a local user. Further, volume quota mechanisms encompass remote storage (via the Remote Storage Server facilities) as well as taking into account sparse files in quota calculations.

Native Structured Storage

Also new to Windows 2000's NTFS implementation is *native structured storage*, which permits ActiveX documents to be stored physically in the same multi-stream format used by ActiveX to logically process structured storage. The effect is improved efficiency in storing ActiveX compound documents.

Volume Mount Points

Volume mount points are NTFS objects that are used to resolve Windows 2000 volume device names in a robust, consistent manner. A volume mount point in an NTFS directory causes the storage subsystem to resolve the directory to a specific local volume.

Volume mounting is accomplished transparently, does not require a drive letter to represent a volume, and allows users or administrators to add storage space without disrupting the name space. Remember, however, that volume mount points always resolve to the root directory on the desired volume and do not mount the directory structure within a volume.

NOTE:

Volume mount points are based on NTFS reparse points and, naturally, require the NTFS file system.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Power Management

While the much publicized ‘energy crisis’ is now a chapter in the history books and flat panel monitors are relieving the biggest energy waste on the desktop, power management for computers is only now becoming a reality. Still, even if we have no energy crisis (in terms of general usage), with the increasing use of laptop computers—where energy (battery power) is always at a premium—power management now has a new and very real importance.

While the most obvious objective behind power management is to conserve battery power, simply conserving power is really secondary to preserving data—that is, avoiding the loss or corruption of data.

For this more important objective, the Windows 2000 power management system is designed to allow applications to:

- Be notified of power-related events
- Query and retrieve power status
- Inform the operating system about application power requirements

NOTE:

Windows 2000 supports two types of power management hardware: ACPI or APM compliant systems.

Creating Power-Aware Applications

Probably the simplest method for creating a power-aware application is to monitor the WM_POWERBROADCAST message that is sent to notify applications of power-management events. Two parameters—the dwPowerEvent parameter (in wParam) and the dwData parameter (lParam)—accompany the WM_POWERBROADCAST message.

The dwPowerEvent parameter identifies the type of event and can be any of the values listed in Table 4.1.

TABLE 4.1: Power Event Types

Value

Meaning

PBT_APMBATTERYLOW	Low battery power
PBT_APMOEMEVEN	An OEM-defined event has occurred
PBT_APMPOWERSTATUSCHANGE	The power status has changed
PBT_APMQUERYSUSPEND	Request for permission to suspend
PBT_APMQUERYSUSPENDFAILED	Suspension request denied
PBT_APMRESUMEAUTOMATIC	Operation resuming automatically after event
PBT_APMRESUMECRITICAL	Operation resuming after critical suspension
PBT_APMRESUMESUSPEND	Operation resuming after suspension
PBT_APMSUSPEND	System is suspending operation

In most cases, the `dwData` parameter is unused. However, if the `dwPowerEvent` parameter identifies any of the three `PBT_APMRESUMExxxx` resume messages, the `dwData` parameter can be used to specify the `PBT_APMRESUMEFROMFAILURE` flag, indicating that an attempted suspend operation—following a `PBT_APMSUSPEND` message—has failed.

By default, applications should return `TRUE` to `WM_POWERBROADCAST` messages but, for the `PBT_APMQUERYSUSPEND` message, the return value can be `BROADCAST_QUERY_DENY` to deny a suspend request.

Querying Power Status

The `GetSystemPowerStatus` function is used to retrieve a `LPSYSTEM_POWER_STATUS` structure, which contains information reporting whether the system is running on AC or DC power, whether the battery is presently being charged, and if not, how much life remains in the battery.

The `LPSYSTEM_POWER_STATUS` structure is defined (in `WinBase.h`) as:

```
typedef struct _SYSTEM_POWER_STATUS
{
    BYTE    ACLineStatus;
    BYTE    BatteryFlag;
    BYTE    BatteryLifePercent;
    BYTE    Reserved1;
    DWORD   BatteryLifeTime;
    DWORD   BatteryFullLifeTime;
} SYSTEM_POWER_STATUS;
```

The `ACLineStatus` member identifies the presence or absence of AC power and can be one of the following values:

Value	Meaning
0	Offline (battery)
1	Online
255	Unknown status

The `BatteryFlag` value reports the battery charge status and can be a combination of the following values:

Value	Meaning
1	High
2	Low
4	Critical
8	Charging
128	No system battery
255	Unknown status

Rather obviously, some combinations would not be expected, but 8 (charging) could be reported in combination with bit values 1, 2, or 4.

A more detailed report on the battery status is available with smart battery subsystems. When a smart battery subsystem is present, a second and more detailed report of battery status is available in the BatteryLifePercent field as a percentage of the full battery charge in the range 0..100 or, if the status is unknown, as 255.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)
Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The `BatteryLifeTime` field reports the remaining battery life (estimated) in seconds or reports `0xFFFFFFFF` if the value is unknown. In like fashion, the `BatteryFullLifeTime` field reports the battery life in seconds when fully charged or, again, reports `0xFFFFFFFF` if the value is indeterminate.

WARNING:

In the absence of a smart battery subsystem, the time values will probably not be accurate enough to be useful although the battery life percentage may still be reported correctly (see Figure 4.2).

The `BatteryTest` program (included on the CD) provides a simple demonstration of the `GetSystemPowerStatus` function. Of course, this demo program is intended for a laptop or portable and, if it is executed on a desktop system, the program will have little to report except for AC Present (see Figure 4.2).



FIGURE 4.2 Testing the battery level

Making Applications “Power-Friendly”

Applications can be made ‘power-friendly’ by following either of two approaches.

The first approach is to ensure that the behavior of the application takes into account the possibility that power is transient (in other words, batteries can run down) and that portable systems, in particular, may shut down some subsystems—such as the hard drive—to conserve power.

For an application to perform in a power-friendly manner, it should:

- Check battery levels before commencing long or power-intensive operations such as formatting or defragmenting drives, reindexing database files, initiating lengthy print jobs, and so on
- Automatically flush application data to a disk file when low battery conditions are reported
- Check device status for idle operations before demanding optional services—for example, use the `GetDevicePowerState` function to determine if the disk drive is ready for use (powered up) before performing non-essential operations

The second approach—although one that should be employed with restraint—is to notify the system when your application has special requirements and (temporarily only) needs to force the idle timer to reset.

Normally, on desktop systems as well as laptops, the system suspends itself when no activity has been detected for a specified period of time.

On desktops, this usually consists of first loading a screen saver and only later of blanking (powering down) the monitor, often turning off the hard drive and optionally putting the entire system on standby. Typical settings for a desktop system might be to power down the monitor after 15 minutes of inactivity and to power down the hard drive after an hour. The time intervals, of course, are user-determined though the screen saver settings and through the power management options (see the Control Panel for settings).

On laptops and portables—where power conservation is dictated by a need to limit unnecessary drains on the batteries—there are often two sets of power monitoring options. One is applied while there is AC present, and a more restrictive set is used when the system is on battery power.

Thus, in both situations—on desktop or portable systems—it is possible for the system to go to various levels of standby operation. However, when applications are executing critical operations (but without user input or activity), applications may call the `SetThreadExecutionState` function to force a reset of the idle timer.

TIP:

The system automatically detects activities, which include local keyboard or mouse input, server activity, and changing window focus. Activities that are not detected are disk or CPU activity and changes to the video display.

SetThreadExecutionState

The `SetThreadExecutionState` function is called by an application to inform the system that system resources are in use and that the system should not enter the sleeping power state while the application (or application thread) is active. The `SetThreadExecutionState` function is called as:

```
EXECUTION_STATE SetThreadExecutionState(  
    EXECUTION_STATE esFlags); // execution requirements
```

The `esFlags` parameter specifies the thread's execution requirements and can be one or more of these flag values:

Flag	Description
ES_SYSTEM_REQUIRED	Informs the system that the thread is performing some operation that is not normally detected as activity by the system. This resets the system idle timer and prevents the system from putting the computer into the sleeping state.
ES_DISPLAY_REQUIRED	Informs the system that the thread is performing some operation that is not normally detected as display activity by the system. This resets the display idle timer and prevents the system from turning off the display.
ES_CONTINUOUS	Informs the system that the state being set should remain in effect until the next call that uses <code>ES_CONTINUOUS</code> and one of the other state flags is cleared. If the function is called without <code>ES_CONTINUOUS</code> , the idle timer is simply reset. To maintain the system or display in the working state, <code>SetThreadExecutionState</code> must be called at regular intervals.
ES_USER_PRESENT	Informs the system that a user is present. In response the system employs the power management policies set by the user. If no user is present, the system does not wake the display device and returns to the sleeping state as quickly as possible.

On success, the function returns the previous thread execution state or, on failure, returns `NULL`.

Applications that require a constant active presence—such as answering machines, backup utilities, fax servers, or network management programs—should use `ES_SYSTEM_REQUIRED | ES_CONTINUOUS` while processing events.

Video multimedia or presentation applications that require the display to remain active in the absence of user input would use ES_DISPLAY_REQUIRED.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Most applications, however, such as Internet browsers, games, spreadsheets, or word processors—where regular user input is expected and is required to trigger operations—do not need to employ `SetThreadExecutionState` calls.

NOTE:

In the future, the OnNow ACPI—a specification for a new design initiative—may appear in the form of PCs that are always on even when they appear powered down and that respond immediately to user input or other requests. At present, this is only a specification that may eventually be supported by new hardware and components, but currently is not fully implemented, although Windows 2000 does include preliminary support for ACPI systems.

The Windows 2000 Kernel

The Windows 2000 kernel is designed to target database, engineering, and scientific applications. Because many high-end applications make extensive use of high levels of memory, the old 2GB limit on memory has been lost. While the Windows 2000 Professional edition (workstation) does have a 4GB limit on physical memory, the Advanced Server and Data Center versions raise the limit to 64GB while the Alpha CPU supports an undefined very large memory (VLM) model.

NOTE:

Remember when a certain CEO announced with great certainty that 640KB of RAM was “enough for anyone”?

Note that the sizes mentioned apply only to the physical memory (RAM) installed and are not limits on memory addressing (virtual memory addresses), which have also changed. There is, however, no clear documentation suggesting what limits presently exist for virtual memory addresses except for citing the address space as a 32-bit space (in other words, 4GB addressable).

In addition to increasing the memory limits, Windows 2000 also provides for fast mapping of physical pages within the 32-bit address space.

Job Objects and Functions

Another area where Windows 2000 offers marked changes is found in the introduction of new kernel objects called job objects.

All Win32 applications consist of one or more *processes* where a process is a task performed by an executing program. Of course, an application may use only a single process, in which case the process is the application (and vice versa), but other applications may contain two or more processes that are essentially independent subprograms.

Further, within any process, there may be more than one *thread* executing. 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.

While processes, threads, and fibers form a descending hierarchy, *job objects* appear at a higher level and are used to manage groups of processes, controlling these groups as units. Thus, a job object is a nameable, securable, and sharable object that has control over the attributes of the processes associated with the job object. At the same time, any operations performed on the job object affect all processes associated with the job object.

Also, with the use of job objects, system security is able to enforce job quotas and to maintain and enforce security contexts.

Job Object Functions

Functions to create and manage job objects are found in the Process and Thread Functions group but are applicable only under Windows NT 5.0 (Windows 2000) or later. The principal functions supplied include:

CreateJobObject	Creates and (optionally) names a job object. An attribute's structure provides the security descriptor for the job object and determines whether child processes can inherit the returned handle.
OpenJobObject	Opens an existing job object and specifies the desired access mode.
AssignProcessToJobObject	Assigns processes to a specific job object; the process is subject to the limits set for the job object.
TerminateJobObject	Terminates all processes currently associated with the job. This is an unconditional exit and should be used only in extreme circumstances.
QueryInformationJobObject	Obtains limit and job state information from the job object.
SetInformationJobObject	Sets limit and job state information for a job object.
UserHandleGrantAccess	Grants or denies access to a USER handle to a job with a user-interface restriction. When access is granted, all processes associated with the job can subsequently recognize and use the handle.

These functions are detailed further in the Visual Studio online documentation.

Job object functions allow setting basic limits, UI restrictions, and security limits. These include:

- Basic Limits, which can be set for:
 - Per-process and per-job user time limits
 - Minimum/maximum working set size
 - Number of processes
 - Processor affinity (for multi-processor systems)
- UI Restrictions, which cover:
 - ExitWindows (Ex)
 - Access to non-job USER objects
 - Clipboard access
 - SystemParametersInfo
- Security Limits, which include:

- No administrator token
- Only restricted token
- Only specific token
- Filter token

Querying job information reports accounting information and a process ID list. The accounting information details such items as total and this period user/kernel time, page faults, and total, active, and terminated processes.

Thread Pooling

Windows 2000 introduces thread pooling with a system-managed thread pool for applications. A thread pool is used to save expendable resources by reducing the number of threads waiting for single objects. In connection thread pools, the RegisterWaitForSingleObject and UnregisterWaitEx functions control how threads wait for callback functions. The timer queue functions—CreateTimeQueue and DeleteTimerQueueEx—offer a means for timing thread pools and the BindIoCompletionCallback function offers a means for threads to be notified when I/O operations are completed.

WARNING:

Some of the functions mentioned in this section are subject to change and may or may not be documented in early releases of the SDK.

The Windows 2000 User Interface/GDI

Under Windows 2000, the user interface and graphical device interface (GDI) have largely duplicated the appearance of and the features found in Windows 95/98—many of which were not found in Windows NT 4.0. At the same time, there are some elements that are entirely new to Windows 2000.

Some of the new features have been announced or discussed in various sources but have not yet been documented in publicly available sources. Enough, however, has been documented to make them worth mention.

Message-Only Windows

Somewhere between undocumented and documented is the `HWND_MESSAGE` constant, which can be passed as the parent window handle to a `CreateMessage` or `CreateMessageEx` function to create a message-only window. Message-only windows can be used to send and receive messages but they are not visible, do not appear in the Z-order, are not enumerated, and do not receive broadcast messages.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Alphablend

The Alphablend function is similar to StretchBlt, except that Alphablend has the capability to display bitmaps or images that contain transparent or semi-transparent areas.

GetGuiResources

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

LockWorkStation

The LockWorkStation function locks the workstation's display and can be released only by a user logging in again. This is the same as pressing Ctrl+Alt+Del and clicking the Lock Workstation option.

SendInput

The SendInput function synthesizes keystrokes, mouse motions, and button clicks. (Introduced in Windows 98 and in NT 4, SP 3.)

TrackMouseEvent

The TrackMouseEvent function posts messages when the mouse pointer leaves a window (WM_MOUSELEAVE) or hovers over a window for a specified amount of time (WM_MOUSEHOVER).

HTML Resource Type

HTML resources can now be included in projects either by creating a new resource or by importing an existing HTML file. Although HTML resources can include bitmaps or other graphic images, the images should also be included in the resource script.

New Common Controls

New common controls supported in Windows 2000 include an IP address display, date/time controls, IE-style

toolbars, and a page scroller.

SystemParametersInfo

The SystemParametersInfo function is used both to query and to set a wide variety of system-wide parameters and, at the same time, it can also update the user profile with settings. The list of settings is too long to repeat here in its entirety but there are a number of settings that originally appeared in Windows 95 or 98 (but not in Windows NT 4.0) that are now implemented in Windows 2000 (NT5) as enhancements to the GUI.

Active Window Tracking

SPI_GETACTIVEWINDOWTRACKING SPI_SETACTIVEWINDOWTRACKING

Reports or sets active window tracking (activating the window the mouse is on).

SPI_GETACTIVEWNDTRKZORDER SPI_SETACTIVEWNDTRKZORDER

Reports or sets bringing to the top windows activated through active window tracking.

SPI_GETACTIVEWNDTRKTIMEOUT SPI_SETACTIVEWNDTRKTIMEOUT

Reports or sets the active window tracking delay, in milliseconds.

Handicapped Preferences

SPI_GETKEYBOARDPRE

Reports the user's keyboard preference—that is, whether the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden.

SPI_GETSCREENREADER SPI_SETSCREENREADER

Reports or sets whether a screen reviewer utility is running. A screen reviewer utility directs textual information to an output device, such as a speech synthesizer or Braille display. When this flag is set, an application should provide textual information in situations where it would otherwise present the information graphically.

Foreground Switching

SPI_GETFOREGROUNDFLASHCOUNT SPI_SETFOREGROUNDFLASHCOUNT

Reports or sets the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request.

SPI_GETFOREGROUNDLOCKTIMEOUT SPI_SETFOREGROUNDLOCKTIMEOUT

Reports or sets the length of time (in milliseconds) following user input during which the system will not allow applications to force themselves into the foreground.

GUI Appearance

SPI_GETCOMBOBOXANIMATION SPI_SETCOMBOBOXANIMATION

Reports or sets the slide-open effect for combo boxes.

SPI_GETGRADIENTCAPTIONS SPI_SETGRADIENTCAPTIONS

Reports or sets the gradient effect for window title bars. For more information about the gradient effect, refer to GetSysColor.

SPI_GETHIGHCONTRAST SPI_SETHIGHCONTRAST

Reports or sets information about the HighContrast accessibility feature.

SPI_GETLISTBOXSMOOTHSCROLLING SPI_SETLISTBOXSMOOTHSCROLLING

Reports or sets the smooth-scrolling effect for list boxes.

Mouse Operations

SPI_GETMOUSESPEED SPI_SETMOUSESPEED

Reports or sets the current mouse speed, determining how far the pointer will move based on the distance the mouse moves.

Screen Saver Status

SPI_GETSCREENSAVERRUNNING

Reports whether a screen saver is currently running on the window station of the calling process. Only the interactive window station ("WinSta0") can have a screen saver running.

Task Switching

SPI_GETSWITCHTASKDISABLE SPI_SETSWITCHTASKDISABLE

(NT5 only) Reports or sets ALT+TAB and ALT+ESC task switching status

Power Settings

Collectively, the power setting flags listed here are supported under Windows 95 for 16-bit applications only, under Windows 98 for both 16- and 32-bit applications, and under Windows 2000 (NT5) for 32-bit applications only.

SPI_GETLOWPOWERACTIVE SPI_SETLOWPOWERACTIVE

Reports whether the low-power phase of screen saving is enabled, or enables/disables the low-power phase of screen saving.

SPI_GETLOWPOWERTIMEOUT SPI_SETLOWPOWERTIMEOUT

Reports or sets the time-out value (in seconds) for the low-power phase of screen saving.

SPI_GETPOWEROFFACTIVE SPI_SETPOWEROFFACTIVE

Reports whether the power-off phase of screen saving is enabled, or enables/disables the power-off phase of screen saving.

SPI_GETPOWEROFFTIMEOUT SPI_SETPOWEROFFTIMEOUT

Reports or sets the time-out value for the power-off phase of screen saving.

Summary

Windows 2000 offers quite a variety of new capabilities. While one of the more visible changes appears as support for multiple monitors, the changes to NTFS—including the Change Journal, reparse points, sparse files, volume quotas, native structures storage, and volume mount points—are equally significant, even though these are virtually invisible.

Another new element in Windows 2000 is found in the form of advanced power management capabilities. While power management is aimed at portable systems, applications should be aware of and responsive to power management conditions. After all, portable computers are no longer toys, nor are they uncommon—and modern applications must be prepared to recognize modern realities.

In the Windows 2000 kernel, changes have also taken place with the introduction of job object functions, thread pooling, and the disappearance of old limits on physical (and logical) memory.

Last, while the Windows 2000 User Interface and GDI is similar in appearance to Windows 95/98, underlying the appearance are a variety of enhancements, new tools,x and even new API functions.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART II ***Application Design***

- CHAPTER 5: Pop-Ups: Tip Windows and Menus
- CHAPTER 6: Creating and Synchronizing Multiple Threads
- CHAPTER 7: Creating Processes and Pipes
- CHAPTER 8: Using the Registry
- CHAPTER 9: Exception Handling
- CHAPTER 10: Memory Management
- CHAPTER 11: Security and Cryptography

CHAPTER 5 **Pop-Ups: Tip Windows and Menus**

- Pop-Up windows for tips and messages
- Pop-Up (floating) menus
- Creating menus on the fly

One of the problems that every programmer faces eventually is the need to present information to the user or to offer the user a series of choices. The obvious solution to these two problems in a Windows environment has always been to create a dialog box either to offer a series of controls or to display a message—or use a MessageBox, a specialized type of dialog box.

The advantage of using a MessageBox is, of course, simplicity. Using the MFC foundation classes, all you have to supply is the message string. Everything else is done for you. But there are some problems with using a MessageBox. The MessageBox display:

- Is modal—in other words, it won't go away automatically, which can disrupt the user

- Takes up more space on the screen than is needed to display the information

Using a conventional dialog box to display a message rather than a `MessageBox` does offer the opportunity to display more information or to display information in a more organized format than a simple message. However, the problems with using a conventional dialog box to offer a series of choices parallel the `MessageBox` dialog box shortcomings. The conventional dialog box must be:

- Designed in advance
- Explicitly dismissed

NOTE:

Granted, this first objection is not completely correct because you can create a dialog box and populate it with controls at runtime rather than depend solely on a predesigned dialog box, but the problems inherent in calculated layout and spacing make this approach generally impractical.

Pop-Up Tip Windows

Rather than simply pointing out shortcomings in dialogs and message boxes, however, this chapter will examine some alternatives, beginning with pop-up tip windows. Pop-up tip windows:

- Present brief messages in a minimal space
- Do not require a user response—they can vanish automatically if desired
- Can be positioned anywhere on the screen

Common examples of pop-up tip windows are found in the tooltips that appear when the mouse is positioned over a toolbar control. These tip windows, however, are a function of the `CToolBar` class and depend on tip messages that were defined when the toolbar resource was defined (see Chapter S6 on the CD).

Because tooltips are rather limited—being connected to toolbar controls—this chapter will introduce an alternative by creating a `CTipWnd` class. This class will let you display a pop-up message anywhere on the screen. Further, the tip window message, like the toolbar tip window, will occupy a minimum amount of screen real estate and will vanish automatically.

In most cases, when you need to offer a selection “on the fly,” you’re looking at a limited series of choices that could be presented in a simple menu format, instead of as a complex dialog box. And, more importantly, a menu:

- Can be created on the fly, in response to runtime conditions
- Can be displayed as a pop-up anywhere on the screen without being attached to a frame window
- Does not force a selection or a response because the pop-up menu can be dismissed automatically if the user wishes to ignore it

Examples of on-the-fly popup menus can be found in a variety of applications including Microsoft Word (Office 95/97 or Office 2000), Corel Word Perfect, and Eudora Pro. For example, when the on-the-fly spell checker in Microsoft Word is enabled and you enter a word it does not recognize, Word underlines that word in red. When you right-click on the unrecognized word, the spell checker presents a menu listing alternate spellings followed by options for Ignore All, Add, AutoCorrect, and Spelling. In this example, the list of alternate spellings has been created at runtime in response to an unrecognized word and inserted in the pop-up menu along with the four mentioned options.

In similar fashion, anytime you can define a list of options—actions, items to select from, or alternative suggestions—you can instruct the application to create a menu showing these options. Once the menu has been defined, it can be displayed as a pop-up (floating) menu, positioned anywhere on the screen you choose to specify.

While both of these features—on-the-fly pop-up menus and pop-up menus—could be implemented using standard C/C++ programming and conventional Windows APIs, they are both much easier to create using the Microsoft Foundation Classes (MFC) as a basis. For this reason, the *Popups* demo is offered only as an MFC-based application. If you wish, however, you will find parallel, non-MFC APIs for all of the important functions (commonly with the same names as the MFC class functions).

The *Popups* Demo Program

The *Popups* demo program is a simple MFC-based application using a single document interface. Once the application executes, the view window (using the `CPopupsView::OnDraw` function) is divided horizontally and vertically into four quadrants.

Also, using the Developer Studio's ClassWizard, message-handling functions are established for four event messages: `OnInitialUpdate()`, `OnSize()`, `OnLButtonDown()`, and `OnRButtonDown()`.

The `OnInitialUpdate` and `OnSize` functions serve similar purposes; in the examples that follow, each has one line of code (in bold) added to the ClassWizard-supplied handling. In each case, the object is to retrieve the size of the client window (the view window) and to store this information in a member variable (`m_cRect`), where the information will be available later.

The `OnInitialUpdate` event occurs only once, after the window is created. This is, however, the earliest opportunity for our application to query the window size. In other applications, we might use this event as our cue to obtain other types of information that are available only after the application window has been created.

```
void CPopupsView::OnInitialUpdate()  
{  
    CView::OnInitialUpdate();  
    GetClientRect( m_cRect );  
}
```

The `OnSize` function performs the same task as the `OnInitialUpdate` function, but it is called any time the size of the application's view window is changed.

```
void CPopupsView::OnSize( UINT nType, int cx, int cy )  
{  
    CView::OnSize(nType, cx, cy);  
    GetClientRect( m_cRect );  
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The two remaining message handler functions, `OnLButtonDown` and `OnRButtonDown`, provide responses to the left (or primary) and right (or secondary) mouse buttons. These events are being used in this demo to display either a pop-up message (a tip window) or a pop-up menu.

TIP:

While the Windows interface allows the left and right mouse buttons to be swapped or, with a three-button mouse, the buttons to be reassigned in any order, `OnLButtonX` events always refer to the button selected by the user as the primary button. `OnRButtonX` events always refer to the secondary button regardless of the user's preferences. Optionally, for a three-button mouse, the `OnMButtonX` events reference the third mouse button. However, this message should not be relied on because the third mouse button is not standard. Refer to Chapter S3 on the CD for more on mouse messages under Windows.

In both the `OnLButtonDown` and `OnRButtonDown` event functions, the mouse position is checked to determine which of the four quadrants the user clicked in.

```
void CPopupsView::OnLButtonDown( UINT nFlags, CPoint point )
{
    if( point.x > ( m_cRect.Width() / 2 ) )
        m_nQuad = 2; else m_nQuad = 0;
    if( point.y > ( m_cRect.Height() / 2 ) )
        m_nQuad ++;
    PopupTips( m_nQuad );
    // CView::OnLButtonDown( nFlags, point );
}
```

```
void CPopupsView::OnRButtonDown( UINT nFlags, CPoint point )
{
    if( point.x > ( m_cRect.Width() / 2 ) )
        m_nQuad = 2; else m_nQuad = 0;
    if( point.y > ( m_cRect.Height() / 2 ) )
        m_nQuad ++;
    PopupMenu( m_nQuad );
}
```

```
// CView::OnRButtonDown( nFlags, point );
}
```

Notice also that the default provisions calling the parent CView class functions have been commented out. As long as the mouse clicks have occurred within the client window, which is the only time these messages will be received within the CPopupsView class, there is no need for any default response. These two lines of code could be restored, however, without affecting the application's operation.

Pop-Up Message Windows

The PopupTips function in the CPopupsView class is here only for demonstration purposes to provide a selection of different messages and to position each one in a different quadrant of the application view window. While the *Popups* demo displays only one pop-up tip at a time, Figure 5.1 shows a composite with four pop-up messages displayed.

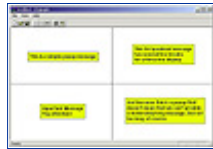


FIGURE 5.1 A selection of pop-up messages

The pop-up messages in the demo program are displayed in response to a left mouse click in any of the four quadrants. However, in your own applications, you should have other reasons and the appropriate algorithms to determine what message you want to display and where within the application (or even outside it) you want the pop-up to appear. Likewise, your application is certainly free to display more than one pop-up window at a time.

NOTE:

See “Pop-ups in Dialogs” later in this chapter for an example of using conditions to trigger a message.

In order to present several different responses according to the quadrant where the event has occurred, the PopupTips function is called with a quadrant argument that was calculated in the OnLButtonDown or OnRButtonDown functions according to the event position. In actual practice, you will want to use some more relevant criteria for deciding what kind of message to display. The code offered here is simply to provide examples.

```
void CPopupsView::PopupTips( int nQuad )
{
    CPoint    cPoint;
    CString   csTip;
    CTipWnd * pTip;

    cPoint = CPoint( m_cRect.Width()/6, m_cRect.Height()/6 );
    switch( nQuad )
    {

        case 0:
            csTip = "This is a simple popup message";
            break;

        case 1:
            cPoint.y += m_cRect.Height()/2;
            csTip = "Important Message\r\nPay attention!";
            break;

        case 2:
            cPoint.x += m_cRect.Width()/2;
            csTip = "This 3rd quadrant message\r\n"
                  "has several line breaks\r\n"
                  "for a three line display";
    }
}
```

```

        break;

    case 3:
        cPoint.y += m_cRect.Height()/2;
        cPoint.x += m_cRect.Width()/2;
        csTip = "Just because this is a popup that\r\n"
               "doesn't mean that we can't provide\r\n"
               "a moderately long message. But not\r\n"
               "too long, of course.";
        break;
    }
    ClientToScreen( &cPoint );

```

Rather than arbitrarily breaking a long line, as you will see in a moment, there is a provision that will handle the embedded `\r\n` (CRLF) characters to provide a multiple line display. If you prefer, however, you could create algorithms to provide line breaks and simply pass an unformatted string, allowing the `CTipWnd` class to decide where breaks should occur for best effects.

Once you've selected a message and position, you will call an instance of the `CTipWnd` class to display the pop-up window, passing the message and position as arguments to the `CTipWnd::Create` method.

```

        pTip = new CTipWnd();
        pTip->Create( csTip, cPoint );
    }

```

Creating the CTipWnd Class

To create the `CTipWnd` class, use the Developer Studio's ClassWizard and base the new class on the `CWnd` class. In the generated *TipWnd.h* header, ClassWizard has defined a `Create` method for the new class as:

```

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTipWnd)
public:
    virtual BOOL Create( LPCTSTR lpszClassName,
                        LPCTSTR lpszWindowName,
                        DWORD dwStyle, const RECT& rect,
                        CWnd* pParentWnd,
                        UINT nID, CCreateContext* pContext = NULL );

//}}AFX_VIRTUAL

```

Normally, when creating a new child class, you would simply accept the generated `Create(...)` method. However, for a tip window, you are not interested in having to supply such a wealth of information that you have no need for. Instead, the supplied `Create(...)` method is replaced with a much simpler `Create(...)` method defined as:

```

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTipWnd)
public:
    virtual BOOL Create( CString csMessage, CPoint cPoint );
//}}AFX_VIRTUAL

```

Here, only two arguments are passed to the `Create` method—the message string and the position (in absolute screen coordinates) where you want the pop-up window to appear.

Are you cheating by not supplying the arguments that the original `Create` method expected? In one sense, yes, but also no. In a moment, you'll see that the missing arguments are still being supplied to the parent `CWnd::CreateEx` method. The difference is that some of these arguments will be calculated within the `CTipWnd::Create` method while others are simply treated as constants rather than being supplied as calling arguments.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

But first we have a few other tasks to take care of. Because the CTipWnd::Create method is called only once when the instance is instantiated, the csMessage argument needs to be stored in a local member variable so that this one critical piece of information will be available later when needed. Also, once the string value has been copied to the member variable, the CalcWndSize method is called to determine what the size of the window necessary to display the message should be. The CalcWndSize method is discussed momentarily, but in brief, it parses the supplied string and uses default font information to calculate a width and height for the pop-up window. At the same time, the cPoint argument positions the window, but this argument and the returned cSize argument do not need to be retained for future use.

```
BOOL CTipWnd::Create( CString csMessage, CPoint cPoint )
{
```

```
    CBrush cBrush;
    UINT nResult;
    CSize cSize;
```

```
    m_csMessage = csMessage;
    cSize = CalcWndSize();
```

A CBrush instance is created to supply a window background. Like the size and position arguments, the background brush is needed only once, when the window class is registered. The CBrush instance becomes part of the pop-up window instance but does not require separate maintenance.

```
    cBrush.CreateSolidBrush( DWCOLOR );
    LPCTSTR lpszTipClass =
        AfxRegisterWndClass( CS_HREDRAW | CS_VREDRAW,
                           ::LoadCursor( NULL, IDC_ARROW ),
                           HBRUSH( cBrush ), NULL );
    nResult = CWnd::CreateEx(
        WS_EX_NOPARENTNOTIFY | WS_EX_TOPMOST |
        WS_EX_WINDOWEDGE,
        lpszTipClass, csMessage,
        WS_POPUPWINDOW | WS_VISIBLE,
```



```

        cPoint.x, cPoint.y, cSize.cx, cSize.cy,
        GetSafeHwnd(), 0 );

```

Now, when you call the parent class `CWnd::CreateEx` method, the arguments that would have been passed in the original `CTipWnd::Create` method are supplied as constants or as calculated values. Notice that you've passed the original message argument as the window title, even though this window instance will not have a title, nor will it have a frame or system menu or so on.

Also notice that the original return value, a call to the `CWnd::Create` method (which was supplied by ClassWizard) has been commented out because it was replaced by the `CreateEx` function. Instead, the value reported by the call is returned to the `CreateEx` function.

Before exiting the `CTipWnd::Create` method, you have one more task to handle: calling `SetCapture` to capture the mouse input. The reasons for this action are discussed in the next section, "Capturing the Mouse and Keyboard."

```

        SetCapture();
        return nResult;
// return CWnd::Create( lpszClassName, lpszWindowName,
//                      dwStyle, rect, pParentWnd, nID, pContext );
}

```

Once you have created the pop-up window, the `OnPaint` method, which was supplied in skeletal form by the ClassWizard, provides a forum for actually displaying the intended message.

First, because you did not store the window size as a member variable, you will call `GetClientRect` to find the client window rectangle. Then, since the window was deliberately created slightly larger than the message required in the `CalcWndSize()` method, call the `CRect::DeflateRect` method to reduce the rectangle size (and position) and to supply a margin around the displayed message. Without this, when the message is drawn, it would be positioned full left and full top within the window. This provision is not simply cosmetic; it also makes the information easier to read.

```

void CTipWnd::OnPaint()
{
    CRect      cRect;
    CPaintDC dc(this);          // device context for painting
    GetClientRect( cRect );
    cRect.DeflateRect( 5, 5, 5, 5 );
    dc.SetBkColor( DWCOLOR );
    dc.DrawText( m_csMessage, &cRect, DT_LEFT | DT_WORDBREAK );
    // Do not call CWnd::OnPaint() for painting messages
}

```

Similarly, the `SetBkColor` function ensures that the text background and the window background match. By default, the text background color would be white. However, in this example, a yellow background was used for the window, so you need to set the same background color before writing the message.

And last, you will use the `DrawText` method to write the actual message within the reduced rectangle. Note that the `DT_LEFT` and `DT_WORDBREAK` flags are necessary to permit the embedded `\n\r` (CRLF) codes to be recognized as formatting instructions.

Capturing the Mouse and Keyboard

As you remember from the previous section, you called the `SetCapture` method to capture the mouse input before completing the `CTipWnd::Create` method. (Given the predilection of programmers for puns, it seems a wonder that this API was not named `MouseTrap`.) In this application, the mouse input capture is used so the pop-up window is automatically closed whenever the user clicks any of the mouse buttons anywhere on the Desktop. To accomplish this, six message-mapped functions are included as:

```

void CTipWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
    ShutDown();
    CWnd::OnLButtonDown(nFlags, point);
}

```



```

}

void CTipWnd::OnNcLButtonDown( ...
void CTipWnd::OnMButtonDown( ...
void CTipWnd::OnNcMButtonDown( ...
void CTipWnd::OnRButtonDown( ...
void CTipWnd::OnNcRButtonDown( ...

```

In each case, the response calls the `ShutDown` method before passing the mouse event to the default `CWnd::On...` method for further handling.

Because you also want the pop-up window to vanish whenever a key is pressed, you supply two additional message-mapped methods for the key and system key events, as:

```

void CTipWnd::OnKeyDown( UINT nChar, UINT nRepCnt, UINT nFlags )
{
    ShutDown();
    CWnd::OnKeyDown( nChar, nRepCnt, nFlags );
}

void CTipWnd::OnSysKeyDown( UINT nChar, UINT nRepCnt, UINT nFlags )
{
    ShutDown();
    CWnd::OnSysKeyDown( nChar, nRepCnt, nFlags );
}

```

Again, the only special provision is to call the `ShutDown` method before passing the event along for default handling.

With these provisions in place, the pop-up windows disappear immediately whenever a mouse button is clicked or when any key is pressed on the keyboard. The objective—to make the pop-up window vanish without placing a demand on the user to respond—is amply satisfied.

Of course, you could also supply a time delay to make the pop-up close without any action of any kind, but this provision, if desired, is left as an exercise for you to perform.

Calculating the Window Size

Calculating the window size is not particularly difficult, but it does seem a bit surprising that there is no default function for this purpose (for example, a function that calculates a size from a string while taking into consideration line breaks within the string). However, surprising or not, the shortcoming is real enough because the only available functions—the `CDC::GetTextExtent` methods—simply report sizes for unbroken strings.

Before you can calculate the size of the string in terms of the display space required, you need to know several other factors, such as the video resolution, the default font, and the font size. To get this type of information, you need to start with a device context. (Determining system capabilities will be covered in more detail in Chapter 12, “Windows 2000 Graphics and Multimedia.”)

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

```
CSize CTipWnd::CalcWndSize()
{
    CDC        cDC;
    CSize       cSize, cSizeFinal;
    CString     csTemp;
    int         nLen, nLenTrim, nLines;

    cDC.CreateIC( _T("DISPLAY"), NULL, NULL, NULL );
```

In this case, because you are not actually drawing anything, you will use the `CDC::CreateIC` method instead of an active device context to create an information context, using the `_T("DISPLAY")` argument to identify the display device.

Having created an information context, the remainder of the task is relatively simple. The message string that you intend to display is scanned for `\r\n` character pairs to identify carriage returns and new line breaks, and then substrings are copied to a local `CString` variable.

Once each substring has been copied, you can obtain the display length—not the character length—of the substring using the `GetTextExtent` function.

```
nLines = 0;
nLenTrim = 0;
cSizeFinal.cx = 0;
cSizeFinal.cy = 0;
csTemp = m_csMessage;
do
{
    nLen = csTemp.Find( "\r\n" );
    if( nLen > 0 )
    {
        csTemp = csTemp.Left( nLen );
        nLenTrim += ( nLen + 2 );
        nLines++;
    }
}
```

```

        cSize = CDC.GetTextExtent( csTemp );
        csTemp = m_csMessage.Right( m_csMessage.GetLength()
                                    - nLenTrim );
    }
    else
    {
        nLines*#43++;
        cSize = CDC.GetTextExtent( csTemp );
    }
    cSizeFinal.cy = cSize.cy;
    cSizeFinal.cx = max( cSizeFinal.cx, cSize.cx );

```

For the horizontal size of the pop-up window, you're interested in finding the longest individual line. For the vertical size, you can simply assume that all lines are the same height and keep the height of any of the lines. You do, however, need to count the number of lines.

```

    }
    while( nLen >= 0 );

```

Once the message string has been parsed, you'll add a margin for borders to the maximum width, calculate the height as the height of an individual line multiplied by the number of lines, and then add a border margin to the height.

```

        cSizeFinal.cx += 15;    // add margin for borders
        cSizeFinal.cy *= nLines;
        cSizeFinal.cy += 15;    // add margin for borders
        return cSizeFinal;
    }

```

The return value—returned as a `CSize` argument—contains the dimensions necessary to display the message with multiple lines and completes the handling necessary for the pop-up message windows.

Pop-Up (Floating) Menus

The second provision in the *Popups* demo, floating menus, is somewhat simpler than the pop-up messages. Because the MFC `CMenu` class already has all of the important provisions to handle pop-up menus, you do not need to create a custom class for this purpose although, for other applications, you might want to consider creating a custom class to simplify some elements.

The *Popups* demo is shown in Figure 5.2 (as a composite) with four floating menus displayed. The pop-up menus are displayed in response to a right mouse click in any of the four quadrants, summoning the menu for that quadrant. Also, partially to demonstrate how you can control as well as create menus on the fly, one element in each menu that arbitrarily corresponds to the quadrant has been disabled.

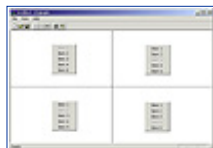


FIGURE 5.2 Floating pop-up menus

The `PopupMenu(...)` method in *CPopupsView.CPP* is written simply as a demonstration. In all probability, you will want to use other methods for positioning the pop-up menus and, of course, for populating the menu selections for your applications.

For the purposes of this example, however, you begin by determining a position according to the quadrant where the mouse click occurs. This position is a purely arbitrary one used for demonstration purposes, and it places each pop-up menu roughly in the center of the selected quadrant, as shown in Figure 5.2.

```

void CPopupsView::PopupMenu( int nQuad )
{
    CPoint    cPoint;
    CMenu     * pMenu;

```

```

cPoint = CPoint( m_cRect.Width()/5, m_cRect.Height()/5 );
switch( nQuad )
{
    case 0:                                     break;
    case 1:      cPoint.y += m_cRect.Height()/2; break;
    case 2:      cPoint.x += m_cRect.Width()/2;  break;
    case 3:      cPoint.y += m_cRect.Height()/2;
                cPoint.x += m_cRect.Width()/2;  break;
}
ClientToScreen( &cPoint );

```

NOTE:

An interesting revision, which is left as an exercise for you to perform, would be to convert the present demo program to pass the mouse event position as an argument and to use the event position to position the pop-up menu.

Once you've determined a position, the next step is to create a CMenu instance, call the CreatePopupMenu() method, and then add items to the menu.

```

pMenu = new CMenu();
pMenu->CreatePopupMenu();
pMenu->AppendMenu( MF_STRING, ID_POPUP_MENU1, "Item 1" );
pMenu->AppendMenu( MF_STRING, ID_POPUP_MENU2, "Item 2" );
pMenu->AppendMenu( MF_STRING, ID_POPUP_MENU3, "Item 3" );
pMenu->AppendMenu( MF_STRING, ID_POPUP_MENU4, "Item 4" );
pMenu->AppendMenu( MF_STRING, ID_POPUP_MENU5, "Item 5" );

```

The CreatePopupMenu() method supplies a blank menu where you can use the AppendMenu(...) and InsertMenu(...) functions to add entries (items) to the menu. The AppendMenu(...) function adds items to the end of the menu list, while the InsertMenu(...) function permits an item to be placed anywhere in the list.

TIP:

The InsertMenu(...) function is especially useful when adding entries to an existing menu.

Menus on the Fly

In this example, when appending items to create a menu, a series of predefined constants are used to identify the menu selections. However, when you want to create a menu on the fly in many cases, you will not have the luxury of having a series of identifiers; instead, you will simply use a series of numerical values that are also generated on the fly.

Still, as long as you can identify the selection by the value returned and match the value to something, such as an entry in a linked list of any kind of elements, it really doesn't matter whether the value is supplied by a defined mnemonic or not.

Once you have created the menu, you can use the EnableMenuItem(...) function to enable or disable individual menu entries.

```

pMenu->EnableMenuItem( ID_POPUP_MENU1 + nQuad,
                      MF_GRAYED | MF_BYCOMMAND | MF_DISABLED );

```

TIP:

The ability to disable a menu item and to indicate that an item is disabled by graying the item is particularly useful for indicating options that may be invalid under certain circumstances or that you may not want users to select at various times.

Menu items can also be checked, unchecked, enabled, or disabled at any time, either when the menu is created or while the menu is in use. The CheckMenuItem(...) and CheckMenuRadioItem(...) functions (which will

be demonstrated later in the Test Dialog section of the *Popups* demo) are used to place a checkmark or a bullet next to a menu item.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Once you have finished working with the individual menu items, enabling or disabling them as needed, call the TrackPopupMenu(...) function to display the menu as a floating window.

```
pMenu->TrackPopupMenu( TPM_LEFTALIGN | TPM_RIGHTBUTTON,  
                        cPoint.x, cPoint.y, this );  
}
```

The TrackPopupMenu(...) function supports three flag values controlling the horizontal position of the pop-up menu. The supported alignment flags are as follows:

TPM_LEFTALIGN: Aligns the menu with the left side at the x-coordinate.

TPM_CENTERALIGN: Centers the menu horizontally on the x-coordinate.

TPM_RIGHTALIGN: Aligns the menu with the right side at the x-coordinate.

The top of the pop-up menu is always aligned on the specified y-coordinate.

Two additional flags cause the menu to track the position of either a left or right mouse-button click. These flag values are as follows:

TPM_LEFTBUTTON: Sets the pop-up menu to track hits from the left mouse button.

TPM_RIGHTBUTTON: Sets the pop-up menu to track hits from the right mouse button.

The next two arguments shown in the example specify the x and y-axis coordinates relative to the screen where the pop-up menu will appear. The last argument shown is a pointer identifying the window owning the pop-up menu. In the example, the this argument identifies the current window instance.

An optional final argument—omitted in the current example—is an lpRect or CRect argument identifying a rectangular area where a mouse click will not dismiss the pop-up menu. If this parameter is NULL or is omitted, the menu vanishes automatically any time the mouse is clicked outside of the menu window.

Pop-Ups in Dialogs

The first two demonstrations showed pop-up messages and pop-up menus overlying an application window. Both of these utility features can also be used with dialog boxes, but how they are summoned can be subject to different constraints. In Figure 5.3, you can see several simple dialog boxes with a half-dozen different

pop-up messages, each triggered by a different set of conditions.



FIGURE 5.3 Pop-up messages in a dialog box window

In Figure 5.3, beginning at the top left, the first dialog box image shows a dialog box without a pop-up message. Below that, the center-left illustration shows the message that appears when the user has not entered a name, and the bottom-left illustration shows the message that appears when the user fails to supply a password.

On the right in Figure 5.3, the first (top) illustration shows the message that is triggered when the name entered is too short, and, in the center dialog box, the message that indicates the password entered is not long enough.

The two bottom right pop-up messages, which are generated from the dialog class, appear when the dialog box closes: the first message appears after the OK button is clicked if the edit fields are both supplied correctly, and the second when the Cancel button is clicked.

While you can read the code to decipher the details of the conditions summoning each message, a few elements have been used in the handling that make some tests mutually exclusive (see the sidebar on mutually exclusive conditions at the end of this chapter). Therefore, the Select menu in the dialog (see Figure 5.4) allows you to choose which fields are tested.



FIGURE 5.4 Selecting test options

The Select menu also has a third option for a pop-up menu. This option is provided to demonstrate one of the hazards encountered with a dialog box. If you select the Popup Menu option from the Select menu in the dialog box and then click the right mouse button in the dialog box window, you will see a pop-up menu—but only if the mouse click occurs outside one of the dialog box controls.

The conflict is simple. The dialog box controls have their own right-mouse-button handling functions, and when a mouse event occurs on a control, the event message does not reach the parent dialog class.

This does not mean that you cannot use a pop-up menu, only that the pop-up menu must be activated by some condition other than the mouse click—just as pop-up message windows have been activated by conditions within the application.

Mutually Exclusive Conditions

In the dialog box example in the *Popups* demo, an `OnKillfocusX` message map has been created for the Name and Password edit boxes. These functions are called whenever the edit field loses the focus—in other words, the focus is shifted to another control such as another edit box or one of the buttons. By calling `UpdateData` when the focus is lost, the contents of the field can be checked immediately.

As you can see in the following fragments, the tip window is opened when the conditions of the test are not met.

```
void CTestDlg::OnKillfocusUserName()  
{  
    if( m_bCheckName )  
    {  
        GetClientRect( &m_cRect );  
        m_cPoint.y = m_cRect.left;  
        m_cPoint.x = m_cRect.top;  
        UpdateData();  
        if( m_csUserName.IsEmpty() )  
        {  
            m_csTip = "No skipping out now,\r\n\r\n"  
                    "Please enter your name first.";  
        }  
    }  
}
```

```

        ClientToScreen( &mpm_cPoint );
        m_pTip = new CTipWnd();
        m_pTip->Create( m_csTip, m_cPoint );
        GetDlgItem( IDC_USER_NAME )->SetFocus();
    } } }

void CTestDlg::OnKillfocusUserPassword()
{
    if( m_bCheckPassword )
    {
        GetClientRect( &mpm_cRect );
        m_cPoint.y = m_cRect.left;
        m_cPoint.x = m_cRect.top;
        UpdateData();
        if( m_csPassword.IsEmpty() )
        {
            m_csTip = "Sorry, Security demands that\r\n"
                    "you must supply a password.";
            ClientToScreen( &mpm_cPoint );
            m_pTip = new CTipWnd();
            m_pTip->Create( m_csTip, m_cPoint );
            GetDlgItem( IDC_USER_PASSWORD )->SetFocus();
        }
    } } }

```

Now, if the test fails, notice that the focus is returned to the tested control using the SetFocus function. If both tests are implemented at the same time, this is where the conflict occurs. When the focus is moved from one edit field to the other, both of the edit fields are trying to regain the focus and produce a cascade of pop-up tip windows.

If, however, as illustrated by the conditional flags that allow only one of the tests to be implemented, only one control is allowed to reclaim the focus, this conflict does not occur.

And, of course, another alternative would be simply to allow the control(s) to lose the focus and display a message if the entries are not acceptable but not try to reclaim the focus.

In any case, here's one you can play with ... and find your own conflicts.

Summary

Pop-up menus and pop-up tips may not solve all of your programming problems, but they do provide a pair of very useful tools, allowing you to impart information to users and offer them choices. And those two elements are important, if not to you as a programmer, then definitely to the person(s) using your program.

TIP:

Remember, how the end user sees your program is far more important than how you, as a programmer, see the application.

In the next chapter, we will look at how using threads allows us to perform multiple tasks synchronously and, on the increasingly popular multiple-CPU systems, share tasks across multiple processors.

[Previous](#)
[Table of Contents](#)
[Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 6

Creating and Synchronizing Multiple Threads

- How threads work
- Commands for creating and modifying threads
- Mutexes, events, semaphores, and critical sections for synchronizing threads
- A demonstration of threads in action

A program of any complexity at all is a maze of instructions that loop and branch and are full of forks, jumps, and returns. The processor makes decisions as the program runs, and each decision leads through a different set of instructions. Each time the program executes, the processor follows a different path from start to finish. When Theseus fought the Minotaur, he marked his way through the labyrinth by unwinding a thread behind him. When a program wants to do several things at once, it creates objects called *threads*, and each thread winds its own way through the program's code.

Another way to look at it is to say that threads let a program be in two places at once. The system keeps a list of all the threads and cycles through them, giving each a slice of time on the processor. When a time slice ends, the system records the current CPU register values in the thread object, including a pointer to whatever instruction(s) the thread was about to execute. The system then selects another thread, restores *its* CPU registers, and resumes execution wherever the last thread left off. A thread marks a location in the program, and by marking several locations with different threads, the program effectively clones itself and seems to execute in many places at the same time.

This chapter begins with a conceptual introduction to threads, then surveys the commands for using threads, and finishes with a sample program demonstrating threads. By the end of this chapter, you will understand when and how to create threads in your programs, how to manage them while they run, and how to synchronize them so they don't interfere with each other. Mutexes, semaphores, events, and critical sections will hold no more mysteries for you.

Thread Concepts

Threads are closely related to *processes*. A process is a program loaded into memory, complete with all the resources assigned to the program, but a process is static and does nothing by itself.

A thread executes program commands, following a path through the code. Every process possesses one initial thread. Optionally, the *initial thread* (also called the *primary thread*) may create other threads. All the threads belonging to one process share the assets of that process. They all follow instructions from the same code image, refer to the same global variables, write to the same private address space, and have access to the same objects. Think of a process as being a house that is inhabited by threads.

Life without Threads versus Life with Threads and Multiprocessors

While an operation without threads does accomplish similar tasks, the process of doing so is less versatile. For a single-threaded application, initiating a *subprocess* (or child process) requires temporarily tabling the main process and only resuming when the subprocess is completed. In earlier versions of Windows, all applications were single-threaded of necessity, even though Windows itself performed something like a multithreaded process, sharing processor time among multiple applications so each had its own chance to execute.

At the same time, background processes—such as spell-checking, image-rendering, or file maintenance—are often relegated to low priority while GUI processes, which are immediately visible to the user, take precedence. Without threads, the allocation of resources often results in a slow response from the GUI elements when tasks of this nature do become active. Threads, while not in themselves a cure-all, do permit a smoother time-share in such circumstances.

Another circumstance where multiple threads are often used is in solving mathematical problems where two or more sets of equations require simultaneous solutions. That is, the solutions for each equation at each step are needed by other equations in subsequent steps. By assigning a separate thread to solve each equation and by synchronizing successive steps, the values returned by each become available at the proper time. Of course, if one thread completes before another, the early thread can wait (suspend) until the next piece of data becomes available.

Also, funneling multiple threads through a single processor is its own limitation, but hopefully it won't be too long before we will see multiple-CPU systems appearing on desktops. Granted, there are systems available today supporting two processors—and even a few supporting four—but these are usually relegated to employment as servers, rather than desktop systems.

The uses of a multiprocessor are obvious. Suppose, for example, you have one application making heavy demands for CPU resources. With multiple processors to handle tasking, the system is able to allocate different threads to different processors; it spreads the load, so to speak, across several CPUs with different CPUs executing separate tasks simultaneously for different threads.

One area where this will be particularly advantageous is found in 3-D modeling and image processing, where different threads can share aspects of creating an image among two or more CPUs. Another area is in Windows itself, which must handle a variety of tasks of its own in addition to the applications it is supporting. Again, a workload shared among several processors makes faster progress than sharing time-slices on a single processor.

When to Create Threads and Processes

You should consider creating new threads any time your program handles asynchronous activity. Programs with multiple windows, for example, generally benefit from creating a thread for each window. Most *Multi-Document Interface (MDI)* applications create threads for the child windows. A program that interacts with several asynchronous devices creates threads for responding to each device.

A desktop publisher, for example, might assign responsibility for the main window to a single thread of high priority. When the user initiates a lengthy operation, such as pouring text into an empty layout, the program creates a new thread to do the formatting in the background. Meanwhile, the first thread continues to manage the main window and responds quickly to new commands from the user. If the user then asks to cancel the formatting, the input thread can interrupt the formatting thread by terminating it.

In like fashion, threads can also be useful for performing slow disk operations in the background or for communicating with other processes. One thread sends messages, and another waits to receive them.

Another example of synchronous activity that is familiar to many users would be using a contemporary word processor with the automatic spell-checker enabled. Here one (or probably more) threads are responsible for responding to the keyboard activity, updating the text on the screen, and managing a regular update of the backup version of the working file. At the same time, one thread is busy checking what has been written against the selected dictionary or dictionaries and, when a word is not recognized, sending a message to tell the display thread to highlight the unknown word. As long as the word processor is working smoothly, all of these tasks are occurring synchronously even though at different rates.

Similarly, if you are relying on voice recognition—an especially task-intensive process in itself—one thread is probably dedicated to processing the voice input, another to searching the database for matching phrases and trigrams, and a third (with a relatively light task load) to transferring the processed information as text to the front end application.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Any thread can create other threads. Any thread can also create new processes. When a program needs to do several things at once, it must decide whether to create threads or processes to share the work. As a general rule, you should choose threads whenever you can, simply because the system creates them quickly and they interact with each other easily. Creating a process takes longer because the system must load a new executable file image from the disk.

Conversely, creating a new process has advantages since each process receives its own private address space. You might also choose processes over threads as a way of preventing them from interfering, even accidentally, with each other's resources. For more details about processes, refer to Chapter 7, "Processes and Pipes."

Threads and Message Queues

Each window a program creates belongs to the thread that creates it. When a thread creates a window, the system gives it a message queue, and the thread must enter a message loop to read from its queue. If a single thread creates all of a program's windows, the program needs only a single message loop. Conversely, any thread that wants to receive messages must create a window for itself, even if the window remains hidden. Only threads that create windows get message queues.

Thread Objects

At the system level, a thread is an object created by the Object Manager. Like all system objects, a thread contains attributes (or data) and methods (or functions). Figure 6.1 represents a thread object schematically, listing its attributes and methods.

[illegible]

FIGURE 6.1 A thread object contains attributes and methods.

Most of the thread methods have corresponding Win32 functions. When you call `SuspendThread`, for example, the Win32 subsystem responds by calling the thread's `Suspend` method. In other words, the Win32 *application program interface (API)* exposes the `Suspend` method to Win32 applications.

NOTE:

The Win32 API is a library (or libraries) of methods that are called directly or indirectly by applications to request services performed by the operating system. For a simple example, asking for a list of files or directories is accomplished though an API call. Under DOS, a similar task would have been accomplished by calling a DOS interrupt function. Under Windows, the principal remains the same; an API call is invoking a system-supplied function to perform a task.

The Thread Context attribute is the data structure for saving the machine state whenever the thread stops executing. I'll explain other attributes as you proceed through the chapter.

Objects and Handles

Windows has always protected some internal structures, such as windows and brushes, from direct manipulation. Programs that run at the system's user level (as opposed to the more privileged kernel level) may not directly examine or modify the inside of a system object. Only by calling Win32 API routines can you do anything at all with an object. Windows gives you a handle to identify the object, and you pass the handle to functions that need it. Threads, too, have handles, as do processes, semaphores, files, and many other objects. Only the Object Manager touches the inside of an object.

The function that creates a thread returns a handle to the new object. With the handle, you can:

- Raise or lower the thread's scheduling priority
- Make the thread pause and resume
- Terminate the thread
- Find out what value the thread returned when it ended

Scheduling and Synchronizing Threads

Working with threads requires more than just starting and stopping them. You also need to make threads work together, and effective interaction requires control over timing. Timing control takes two forms: priority and synchronization. Priority controls how often a thread gets processor time. Synchronization regulates threads when they compete for shared resources and imposes a sequence when several threads must accomplish tasks in a certain order.

Process, Base, and Dynamic Priority

When the system scheduler preempts one thread and looks for the next thread to run, preference is given to threads of high priority. Some activities, such as responding to an unexpected power loss, always execute at a very high priority. Likewise, system interrupt handlers have a higher priority than user processes. In short, every process has a priority rating, and threads derive their base scheduling priority from the process that owns them.

As shown earlier in Figure 6.1, a thread object's attributes include a *base priority* and a *dynamic priority*. When you call commands to change a thread's priority, you change the base priority. You cannot, however, push a thread's priority more than two steps above or below the priority of its process. In other words, a

thread can't grow up to be very much more important than its parent is.

Although a process cannot promote its threads very far, the system can. The system grants a sort of field promotion—dynamic priority—to threads that undertake important missions. When the user gives input to a window, for example, the system always elevates all the threads in the process that owns the window. When a thread waiting for data from a disk drive finally receives it, the system promotes that thread, too. These temporary boosts, added to the thread's current base priority, form the dynamic priority. The scheduler chooses threads to execute based on their dynamic priority. Process, base, and dynamic priorities are distinguished in Figure 6.2.



FIGURE 6.2 How the range of a thread's priority derives from the priority of the process

Dynamic priority boosts begin to degrade immediately. A thread's dynamic priority slips back one level each time the thread receives another time slice and finally stabilizes at the thread's base priority.

How Scheduling Happens

To select the next thread, the scheduler begins at the highest priority queue, executes the threads there, and then works its way down the rest of the list. But the dispatcher ready queue may not contain all the threads in the system. Some may be suspended or blocked. At any moment, a thread may be in one of six states:

Ready	Queued, waiting to execute
Standby	Ready to run next (in the batter's box, so to speak)
Running	Executing, interacting with the CPU
Waiting	Not executing, waiting for a signal to resume
Transition	About to execute after the system loads its context
Terminated	Finished executing, but the object is not deleted

When the scheduler selects a ready thread from the queue, it loads a *context* for the thread. The context includes a set of values for the machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. (If part of the context has been paged to disk, the thread enters the transition state while the system gathers the pieces.) Changing threads means saving all the pieces of one context and loading all the pieces of the next one into the processor.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The newly loaded thread runs for one time slice, which is likely to be on the order of 20 milliseconds. The system maintains a counter measuring the current time slice. On each clock tick, the system decrements the counter; when it reaches zero, the scheduler performs a context switch and sets a new thread running.

For those familiar with how multiple executables function under, for example, Windows 3.1, executing threads is very much like executing separate applications. The real difference is simply that single applications are now able to separate tasks for parallel execution instead of serial execution.

How Synchronization Happens

To run at all, threads must be scheduled; to run well, they often need to be synchronized. Suppose one thread creates a brush and then creates several threads that share the brush and draw with it. The first thread must not destroy the brush until the other threads finish drawing. Or suppose one thread accepts input from the user and writes it to a file, while another thread reads from the file and processes the text. The reading thread mustn't read while the writing thread is writing. Both situations require a means of coordinating the sequence of actions among several threads.

One solution would be to create a global Boolean variable that one thread uses to signal another. The writing thread might set `bDone` to `TRUE`, and the reading thread might loop until it sees the flag change. That would work, but the looping thread wastes a lot of processor time. Instead, Win32 supports a set of synchronization objects:

- A *mutex* object works like a narrow gate for one thread to pass through at a time.
- A *semaphore* object works like a multiple-lane tollgate that a limited number of threads can pass through together.
- An *event* object broadcasts a public signal for any listening thread to hear.
- A *critical section* object works just like a mutex but only within a single process.

All of these are system objects created by the Object Manager. Although each synchronization object coordinates different interactions, they all work in a similar way. A thread that wants to perform some coordinated action waits for a response from one of these objects and proceeds only after receiving it. The scheduler removes waiting objects from the dispatch queue so they do not consume processor time. When the signal arrives, the scheduler allows the thread to resume.

How and when the signal arrives depends on the object. For example, the one essential characteristic of a mutex is that only one thread can own it. A mutex doesn't do anything apart from letting itself be owned by one thread at a time (*mutex* stands for *mutual exclusion*.) If several threads need to work with a single file, you might create a mutex to protect the file. Whenever any thread begins a file operation, it first asks for the mutex. If no one else has the mutex, the thread proceeds. If, on the other hand, another thread has just grabbed the mutex for itself, the request fails and the thread *blocks*, becoming suspended while it waits for ownership. When one thread finishes writing, it releases the mutex, and the waiting thread revives, receives the mutex, and performs its own file operations.

The mutex does not actively protect anything. It works only because the threads that use it agree not to write to the file without owning the mutex first, and there is nothing that actually prevents all the threads from trying to write at once. The mutex is just a signal, much like the Boolean `bDone` in our looping example. You might create a mutex to protect global variables, a hardware port, a handle to a pipe, or a window's client area. Whenever several threads share any system resource, you should consider whether to synchronize their use of it.

Mutexes, semaphores, and events can coordinate threads in different processes, but critical sections are visible only to threads in a single process. When one process creates a child process, the child often inherits handles to existing synchronization objects. Critical section objects cannot be inherited.

Fundamentally, a synchronization object, like other system objects, is simply a data structure. Synchronization objects have two states: signaled and not signaled. Threads interact with synchronization objects by changing the signal or waiting for the signal. A waiting thread is blocked and does not execute. When the signal occurs, the waiting thread receives the object, turns the signal off, performs some synchronized task, and turns the signal back on when it relinquishes the object.

Threads can wait for other objects besides mutexes, semaphores, events, and critical objects. Sometimes it makes sense to wait for a process, a thread, a timer, or a file. These objects serve other purposes as well, but like the synchronization objects, they also possess a signal state. Processes and threads signal when they terminate. Timer objects signal when a certain interval passes. Files signal when a read or write operation finishes. Threads can wait for any of these signals.

Bad synchronization causes bugs. For example, a deadlock bug occurs when two threads wait for each other. Neither will end unless the other ends first. A race condition occurs when a program fails to synchronize its threads. Suppose one thread writes to a file, and another thread reads the new contents. Whether the program works depends on which thread wins the race to its I/O operation. If the writing thread wins, the program works. If the reading thread tries to read first, the program fails.

About Win32 Object Handles

Under 16-bit Windows, an object has only one handle. The handle may be copied into several variables, but it is still one handle and, when the object is destroyed, all the copies of the handle become invalid.

Starting with Windows 95, however, some new objects in Win32 began to work differently. Several threads or processes may have different handles to the same object. Brushes, windows, and device contexts still support only one handle; but a single thread, process, or mutex, for example, may have many different handles. As each finishes with the object, it calls `CloseHandle`. When the last handle closes, the system destroys the object.

NOTE:

Although the total number of handles in the system is limited only by available memory, no single process may possess more than 65,536 open handles.

Thread-Related Commands

As an introduction to the features of multithreaded programming, this section surveys the parts of the Win32 API that relate to threads. The first half explains commands for creating and modifying threads, and the second half concentrates on commands for synchronizing threads.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



ITKNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Making and Modifying Threads

The life cycle of a thread begins when you call `CreateThread`. Other functions let you examine the thread, suspend or resume it, change its priority, and terminate it.

Creating Threads

Any thread can create another thread by calling `CreateThread`, which provides similar services for a thread that `main` or `WinMain` provide for a full program. In other words, the arguments to `CreateThread` specify the properties a thread needs to begin life—primarily, security privileges and a starting function.

The thread's life coincides with the life of its main function. When the function returns, the thread ends. A thread can start at any function that receives a single 32-bit parameter and returns a 32-bit value.

TIP:

The calling parameter and the return value are for your convenience. Although you must declare them, you are not required to use them.

`CreateThread` lets you pass a `DWORD` into the starting function. If several threads execute the same function, you might pass each one a different argument. Each might receive a pointer to a different filename, for example, or a different object handle to wait for.

The Parameters `CreateThread` requires six parameters:

<code>HANDLE</code>	<code>CreateThread</code> (prototype for the <code>CreateThread</code> function)	
<code>LPSECURITY_ATTRIBUTES</code>	<code>lpThreadAttributes</code>	<i>access privileges</i>
<code>DWORD</code>	<code>dwStackSize</code>	<i>say 0 for default</i>
<code>LPTHREAD_START_ROUTINE</code>	<code>lpStartAddress</code>	<i>pointer to function</i>
<code>LPVOID</code>	<code>lpParameter</code>	<i>val passed to function</i>
<code>DWORD</code>	<code>dwCreationFlags</code>	<i>active or suspended</i>
<code>LPDWORD</code>	<code>lpThreadId</code>	<i>system returns ID here</i>

The first parameter points to a `SECURITY_ATTRIBUTES` structure that determines who may share the object and

whether other processes may modify it. The structure contains a security descriptor that assigns access privileges for various system users and groups of users. Most programs simply accept the default descriptor that comes with the current process.

Also, the security structure contains an inheritance flag that, if set to TRUE, allows any child processes that are created to automatically inherit a handle to this object.

```
typedef struct _SECURITY_ATTRIBUTES          /* sa */
{
    DWORD      nLength          size of (SECURITY_ATTRIBUTES)
    LPVOID      lpSecurityDescriptor NULL to accept process's descriptor
    BOOL        bInheritHandle   TRUE if children may inherit object
} SECURITY_ATTRIBUTES *LPSECURITY_ATTRIBUTES
```

You don't need to create a SECURITY_ATTRIBUTES structure unless you want the thread to be inherited. If you pass NULL as the first parameter to CreateThread, the new thread receives the default descriptor and will not be inherited. Alternately, if you do want to create a handle with limited access rights to its object, investigate the four SetSecurityDescriptor functions.

The next three parameters give the new thread material to work with. By default, each thread receives a stack the same size as that of the primary thread. You can change the size with the second parameter, but if the stack later needs more room, the system expands it automatically.

The third parameter points to the function where the thread will start, and the value in the fourth parameter becomes the argument passed to the starting function.

WARNING:

Beware of using a local variable to pass a value to a new thread. The local variable will be destroyed when its procedure ends, and the thread may not have used it yet. Use global variables, allocate memory dynamically, or make the first thread wait for the new thread to terminate before it returns.

The dwCreationFlags parameter may be one of two values: 0 or CREATE_SUSPENDED. A suspended thread does not actually begin to run until you give it a push with ResumeThread. A program that creates a number of threads might suspend them, accumulate their handles, and, when ready, start them all off at once. That's what the sample program later in the chapter does.

The last parameter points to an empty DWORD where CreateThread places a number to identify the thread uniquely in the system. A few functions require you to identify threads by their ID number instead of by their handles.

The Return Value The CreateThread function returns a handle to the new thread. If the thread could not be created, the handle will be NULL. You should be aware that the system will create the thread even if the lpStartAddress or lpParameter values are invalid or point to inaccessible data. In those cases, CreateThread returns a valid handle, but the new thread terminates immediately, returning an error code. You can test a thread's viability with GetExitCodeThread, which returns STILL_ACTIVE if the thread has not ended.

Unless you give CreateThread an explicit security descriptor, the new handle comes with full access rights to the new object. In the case of threads, full access means that with this handle you can suspend, resume, terminate, or change the priority of the thread. The thread handle remains valid even after the thread terminates. To destroy the thread object, close its handle by calling CloseHandle. If more than one handle exists, the thread will not be destroyed until the last handle is closed. If you forget to close the handle, the system will do it automatically when your process ends.

Creating Threads Using MFC

An alternative method of creating threads using MFC is to create a class based on the CWinThread class. A skeletal example of the process follows.

```
// CThreadExample
```

```
IMPLEMENT_DYNCREATE( CThreadExample, CWinThread )
```

```
CThreadExample::CThreadExample()
```

```

{
    ...    // class member variables are initialized here
}

CThreadExample::CThreadExample()
{
}

BOOL CThreadExample::InitInstance()
{
    // TODO:  perform any per-thread initialization here
    ...
    // this is the point where non-variable initializations,
    // such as creating instances of other class objects, are
    // handled
    return TRUE;
}

int CThreadExample::ExitInstance()
{
    // TODO:  perform any per-thread cleanup here
    ...
    return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP(CThreadExample, CWinThread)
//{{AFX_MSG_MAP(CThreadExample)
// NOTE - ClassWizard will add/remove mapping macros here.
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

A CWinThread object represents an execution thread within an application. Where the application's main thread of execution is normally provided by a CWinApp-derived object, the CWinApp class itself is derived from CWinThread. Additional CWinThread objects, as shown in the example, allow multiple threads within the application.

MFC-based applications must use CWinThread-derived classes to ensure that the application is thread-safe. Under MFC, the framework uses thread local data to maintain thread-specific information for CWinThread-managed objects.

WARNING:

Any thread created by the runtime function `_beginthreadex` cannot use any MFC APIs.

MFC Thread Types Two general types of threads are supported: *working threads* and *user-interface threads*. Working threads do not require a message handler. For example, a thread performing background calculations in a spreadsheet that functions without any interactions with the user and does not need to respond to messages.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

In contrast, user-interface threads must process messages received from the system (from the user), so they require a message handler. User-interface threads can be derived from CWinApp or directly from the CWinThread class.

A CWinThread object commonly exists for the duration of the thread, but the behavior of the object may be modified by setting the m_bAutoDelete member to FALSE.

How to Create an MFC Thread Threads are created by calling AfxBeginThread. For a user-interface thread, you call AfxBeginThread with a pointer to the CRuntimeClass of the CWinThread-derived class. For a working thread, you call AfxBeginThread with a pointer to the controlling function and the parameter for the controlling function.

For both working and user-interface threads, you may specify additional parameters to modify the priority, stack size, creation flags, and security attributes for the thread. AfxBeginThread returns a pointer to the new CWinThread object.

Alternatively, you can construct and then create a CWinThread-derived object by calling the CreateThread function for the class. Using this format permits the CWinThread-derived object to be reused between successively creating and terminating thread executions.

Changing a Thread's Priority

High-priority threads get more time on the processor, finish their work more quickly, and are more responsive to the user. But making all of your threads high priority entirely defeats the purpose of priorities. If a number of threads have the same priority—whether their priority is high or low—the scheduler must give them equal processor time. One thread can be more responsive only if the others are less responsive. The same rule applies equally to processes.

TIP:

To maintain overall system performance, threads and processes should be kept at low or average priority levels as much as possible and only raised to high levels when absolutely necessary.

These functions retrieve or modify any thread's base priority:

```

BOOL SetThreadPriority(
    HANDLE hThread          // a thread to modify
    int     iPriority );     // its new priority level

int GetThreadPriority( HANDLE hThread );

```

SetThreadPriority returns TRUE or FALSE for success or failure while GetThreadPriority returns the thread's current priority. To name the possible priority values for both functions, use the following set of constants:

THREAD_PRIORITY_LOWEST	Two levels below process
THREAD_PRIORITY_BELOW_NORMAL	One level below process
THREAD_PRIORITY_NORMAL	Same level as process
THREAD_PRIORITY_ABOVE_NORMAL	One level above process
THREAD_PRIORITY_HIGHEST	Two levels above process
THREAD_PRIORITY_TIME_CRITICAL	Level 15 (in normal user processes)
THREAD_PRIORITY_IDLE	Level 1 (in normal user processes)

The first five values adjust the thread's base-priority level with respect to the level of its parent process, as shown earlier in Figure 6.2. The last two, for critical and idle priority, express absolute priority levels at the upper and lower extremes of the parent's priority class. (The extremes for real-time priority codes are 16 and 31.) The idle priority level works well for screensavers because they should not execute unless nothing else is happening.

WARNING:

Use the time-critical level with extreme caution and only for short periods because it will starve lower-priority threads of processor time.

Suspending and Resuming a Thread's Execution

A suspended thread stops running and will not be scheduled for processor time. It remains in this state until some other thread makes it resume. Suspending a thread might be useful if, for example, the user interrupts a task. You could suspend the thread while waiting for the user to confirm the cancellation. If the user chooses to continue, the interrupted thread can resume where it left off. The sample program later in this chapter suspends several drawing threads whenever the user resizes the window. Then when the window is repainted, the threads continue drawing.

A thread calls these functions to make another thread pause and resume:

```

DWORD SuspendThread( HANDLE hThread );
DWORD ResumeThread( HANDLE hThread );

```

A single thread may be suspended several times in succession without any intervening resume commands, but every SuspendThread command must eventually be matched with a ResumeThread command. The system counts the number of pending suspension commands for each thread. (See the Suspension Count attribute in Figure 6.1.) SuspendThread increments the counter and ResumeThread decrements it, while both functions return the previous value of the counter in a DWORD. Only when the counter returns to 0 does the thread resume execution.

While a thread can suspend itself but cannot resume itself, a thread can put itself to sleep for a set amount of time. The Sleep command delays execution, removing the thread from the scheduler's queue until some interval passes. Interactive threads that write or draw information for the user often take short naps to give the user time to see the output. Sleep is better than an empty loop because it doesn't use processor time.

A thread calls these functions to pause for a set time:

```

VOID Sleep( DWORD dwMilliseconds );

DWORD SleepEx(
    DWORD dwMilliseconds, // duration of pause
    BOOL bAlertable );    // TRUE to resume if I/O operation finishes

```


The extended SleepEx function typically works in conjunction with background I/O functions and can be used to initiate a read or write operation without waiting for the operation to finish. The operation continues in the background. When it finishes, the system notifies the user by invoking a callback procedure from the program. Background I/O (also called *overlapping I/O*) is particularly helpful in interactive programs that must remain responsive to the user while working with relatively slow devices, such as tape drives and network disks.

The Boolean parameter in SleepEx lets the system wake the thread prematurely if an overlapping I/O operation finishes before the sleep interval expires. If SleepEx is interrupted, it returns WAIT_IO_COMPLETION. If the interval passes without interruption, SleepEx returns 0.

Getting Information about Existing Threads

A thread can easily retrieve the two pieces of its own identity: a handle and an identifier.

These functions return information identifying the current thread:

```
DWORD GetCurrentThreadID( VOID );  
HANDLE GetCurrentThread( VOID );
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The return value from `GetCurrentThreadID` matches the value in `lpIDThread` after a `CreateThread` command. It is the value that uniquely identifies the thread to the system. Although few of the Win32 API commands require you to know a thread's ID, it can be useful for monitoring threads system-wide without needing to keep handles open for each one. Remember that open handles prevent threads from being destroyed.

The handle that `GetCurrentThread` returns serves the same purpose as the handle returned from `CreateThread`. Although it works in the same manner as other handles, it is actually a *pseudohandle*—a special constant that the system always interprets a certain way, much as a single dot (.) in DOS always refers to the current directory and this in C++ always points to the current object. The pseudohandle constant returned from `GetCurrentThread` always refers to the current thread.

WARNING:

Unlike real handles, a pseudohandle does not work when passed to other threads.

For a thread to acquire a real, transferable handle to itself, the `DuplicateHandle` function can be called as:

```
HANDLE hThread;
```

```
hThread = DuplicateHandle(  
    GetCurrentProcess(),           // source process  
    GetCurrentThread(),            // original handle  
    GetCurrentProcess(),           // destination process  
    &amphThread,                    // new duplicate handle  
    0,                             // access rights (overridden by last parameter)  
    FALSE,                        // children do not inherit the handle  
    DUPLICATE_SAME_ACCESS );      // copy access rights from original handle
```

While `CloseHandle` has no effect on a pseudohandle, the handle `DuplicateHandle` creates is real and must eventually be closed. Using a pseudohandle lets `GetCurrentThread` work more quickly, because it assumes that a thread should have full access to itself and returns its result without bothering with any security considerations.

Terminating the Execution of a Thread

Just as a Windows program ends when it comes to the end of WinMain, a thread normally meets its demise when it comes to the end of the function where it began. When a thread comes to the end of its starting function, the system automatically calls ExitThread:

```
VOID ExitThread( DWORD dwExitCode );
```

Although the system calls ExitThread automatically, you may call it directly if some condition forces a thread to an untimely end:

```
DWORD ThreadFunction( LPDWORD lpdwParam )
{
    HANDLE hThread = CreateThread( <parameters> );
    // initialization chores happen here
    // test to see if there was a problem

    if( <error condition> )
    {
        ExitThread( ERROR_CODE );    // cancel the thread
    }
    //
    // no error, work continues
    //
    return( SUCCESS_CODE );          // this line causes the system
                                    // to call ExitThread
}
```

ERROR_CODE and SUCCESS_CODE are whatever you define them to be. In this simple example, you could just as easily have canceled with a return command:

```
if( <error condition> )
{
    return( ERROR_CODE );            // cancel the thread
}
```

This return command has exactly the same effect as ExitThread; in fact, it even results in a call to ExitThread. The ExitThread command is genuinely useful for canceling from within any subroutine's ThreadFunction calls.

When a thread ends at a return statement, the 32-bit return value becomes the exit code passed automatically to ExitThread. After a thread terminates, its exit code is available through this function:

```
// one thread calls this to find out how another ended
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpdwExitCode );
```

GetExitCodeThread returns FALSE if an error prevents it from determining the return value.

ExitThread, whether called explicitly or implicitly as a consequence of return, permanently removes a thread from the dispatch queue and destroys the thread's stack. It does not, however, destroy the thread object that makes it possible for you to ask about the thread's exit status even after the thread stops running. When possible, thread handles should be closed explicitly (by calling CloseHandle) to avoid wasting space in memory.

A thread is terminated by the system when two conditions are met:

- When the last handle to a thread is closed
- When the thread is no longer running

The system will not destroy a running thread, even if all its handles are closed. Instead, the thread is not destroyed until the thread stops running. However, if a process leaves handles open when it terminates, the system closes them automatically and removes orphaned objects no longer held by any process.

With ExitThread, a thread stops itself gracefully at a place of its own choosing while the TerminateThread function allows one thread to stop another abruptly and arbitrarily:

```
// one thread calls this to stop another
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode );
```

A thread cannot protect itself from termination. Anyone with a handle to the thread can force the thread to stop immediately, regardless of its current state (providing, of course, that the handle allows full access to the thread). Using the default security attributes in `CreateThread` produces a handle with full access privileges.

`TerminateThread` does not destroy the thread's stack, but it does provide an exit code. Both `ExitThread` and `TerminateThread` set the thread object to its signaled state, so any other threads waiting for this one to end may proceed. After either command, the thread object lingers lifelessly until all its handles have been closed.

Using Equivalent C Runtime Functions

Several C runtime library commands duplicate some of the Win32 thread commands:

```
unsigned long _beginthread(
    void( *start_address )( void * ), // starting function
    unsigned stack_size,              // initial stack size
    void *arglist );                 // parameter for starting function
void _endthread( void );
void _sleep( unsigned long ulMilliseconds );
```

`_beginthread` performs some of the internal initialization for a new thread that other C runtime functions, such as `signal`, depend on. The rule is consistency: If your program manipulates threads with C runtime functions, then use only C runtime functions wherever you have a choice. If your program uses Win32 functions with its threads, then stick to `CreateThread` and `ExitThread`.

Also, if the thread calls C runtime functions, then it should be created using the C functions rather than the Win32 API. A few C routines require the initialization performed by the `_beginthread` function.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Synchronizing Threads

To work with threads, you must be able to coordinate their actions. Sometimes coordination requires ensuring that certain actions happen in a specific order. Besides the functions to create threads and modify their scheduling priority, the Win32 API contains functions to make threads wait for signals from objects, such as files and processes. It also supports special synchronization objects, such as mutexes and semaphores.

The functions that wait for an object to reach its signaled state best illustrate how synchronization objects are used. With a single set of generic waiting commands, you can wait for processes, threads, mutexes, semaphores, events, and a few other objects to reach their signaled states. This command waits for one object to turn on its signal:

```
DWORD WaitForSingleObject( HANDLE hObject,      // object to wait for
                           DWORD dwMilliseconds ); // maximum wait time
```

WaitForSingleObject allows a thread to suspend itself until a specific object gives its signal. In this command, a thread also states how long it is willing to wait for the object. To wait indefinitely, set the interval to INFINITE. If the object is already available, or if it reaches its signal state within the designated time, WaitForSingleObject returns 0 and execution resumes. If the interval passes and the object is still not signaled, the function returns WAIT_TIMEOUT.

WARNING:

Beware when setting the interval to INFINITE. If for any reason the object never reaches a signaled state, the thread will never resume. Also, if two threads establish a reciprocal infinite wait, they will deadlock.

To make a thread wait for several objects at once, call WaitForMultipleObjects. You can make this function return as soon as any one of the objects becomes available, or you can make it wait until all the requested objects finally reach their signaled states. An event-driven program might set up an array of objects that interest it and respond when any of them signals.

```
DWORD WaitForMultipleObjects(
```

```

DWORD    dwNumObjects,    // number of objects to wait for
LPHANDLE lpHandles,      // array of object handles
BOOL     bWaitAll,        // TRUE, wait for all; FALSE, wait for any
DWORD    dwMilliseconds ); // maximum waiting period

```

Again, a return value of WAIT_TIMEOUT indicates that the interval passed and no objects were signaled. If bWaitAll is FALSE, a successful return value, which holds a flag from any one element, indicates which element of the lpHandles array has been signaled. (The first element is 0, the second is 1, and so on.) If bWaitAll is TRUE, the function does not respond until all flags (all threads) have completed.

Two extended versions of the wait functions add an alert status allowing a thread to resume if an asynchronous read or write command happens to end during the wait. In effect, these functions say, “Wake me up if the object becomes available, if a certain time passes, or if a background I/O operation runs to completion.”

```

DWORD WaitForSingleObjectEx(
    HANDLE hObject,          // object to wait for
    DWORD  dwMilliseconds,   // maximum time to wait
    BOOL   bAlertable );     // TRUE to end wait if I/O completes

```

```

DWORD WaitForMultipleObjectsEx(
    DWORD    dwNumObjects,    // number of objects to wait for
    LPHANDLE lpHandles,      // array of object handles
    BOOL     bWaitAll,        // TRUE wait for all; FALSE wait for any
    DWORD    dwMilliseconds,  // maximum waiting period
    BOOL     bAlertable );    // TRUE to end wait if I/O completes

```

Successful wait commands usually modify the awaited object in some way. For example, when a thread waits for and acquires a mutex, the wait function restores the mutex to its unsignaled state so other threads will know it is in use. Wait commands also decrease the counter in a semaphore and reset some kinds of events.

Wait commands do not modify the state of the specified object until all objects are simultaneously signaled. For example, a mutex can be signaled, but the thread does not receive ownership immediately because it is required to wait until the other objects are also signaled; therefore, the wait function cannot modify the object. In addition, the mutex may come under the ownership of another thread while waiting, which will further delay the completion of the wait condition.

Of course, you must create an object before you can wait for it. Start with mutexes and semaphores because they have parallel API commands to create the objects, acquire or release them, get handles to them, and destroy them.

Creating Mutexes and Semaphores

The creation functions for mutexes and semaphores need to be told what access privileges you want, some initial conditions for the object, and an optional name for the object.

```

HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpsa, // optional security attributes
    BOOL bInitialOwner         // TRUE if creator wants ownership
    LPTSTR lpszMutexName );     // object's name

HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpsa, // optional security attributes
    LONG lInitialCount,        // initial count (usually 0)
    LONG lMaxCount,            // maximum count (limits # of threads)
    LPTSTR lpszSemName );      // name of the semaphore (may be NULL)

```

If the security descriptor is NULL, the returned handle will possess all access privileges and will not be inherited by child processes. The names are optional, but useful for identification purposes only when several different processes want handles to the same object.

By setting the bInitialOwner flag to TRUE, a thread both creates and acquires a mutex at once. The new mutex

remains unsignaled until the thread releases it.

While only one thread at a time may acquire a mutex, a semaphore remains signaled until its acquisition count reaches iMaxCount. If any more threads try to wait for the semaphore, they will be suspended until some other thread decreases the acquisition count.

Acquiring and Releasing Mutexes and Semaphores

Once a semaphore or a mutex exists, threads interact with it by acquiring and releasing it. To acquire either object, a thread calls WaitForSingleObject (or one of its variants). When a thread finishes whatever task the object synchronizes, it releases the object with one of these functions:

```
BOOL ReleaseMutex( HANDLE hMutex );
```

```
BOOL ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lRelease,           // amount to increment counter on release  
                             // (usually 1)  
    LPLONG lplPrevious );    // variable to receive the previous count
```

Releasing a mutex or a semaphore increments its counter. Whenever the counter rises above 0, the object assumes its signaled state, and the system checks to see whether any other threads are waiting for it.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



▶ KEYWORD SEARCH

[GO!](#)[▶ Search Tips](#)[▶ Advanced Search](#)

▶ PUBLICATION LOOKUP

[GO!](#)

▶ JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Only a thread that already owns a mutex—in other words, a thread that has already waited for the mutex—can release it. Any thread, however, can call ReleaseSemaphore to adjust the acquisition counter by any amount up to its maximum value. Changing the counter by arbitrary amounts lets you vary the number of threads that may own a semaphore as your program runs. You may have noticed that CreateSemaphore allows you to set the counter for a new semaphore to something other than its maximum value. You might, for example, create it with an initial count of 0 to block all threads while your program initializes, and then raise the counter with ReleaseSemaphore.

WARNING:

Remember to release synchronization objects. If you forget to release a mutex, any threads that wait for it without specifying a maximum interval will deadlock; they will not be released.

A thread may wait for the same object more than once without blocking, but each wait must be matched with a release. This is true of mutexes, semaphores, and critical sections.

Working with Events

An *event* is the object a program creates when it requires a mechanism for alerting threads if some action occurs. In its simplest form—a manual reset event—the event object turns its signal on and off in response to the two commands SetEvent (signal on) and ResetEvent (signal off). When the signal is on, all threads that wait for the event will receive it. When the signal is off, all threads that wait for the event become blocked. Unlike mutexes and semaphores, manual reset events change their state only when some thread explicitly sets or resets them.

You might use a manual reset event to allow certain threads to execute only when the program is not painting its window or only after the user enters certain information. Here are the basic commands for working with events:

```
HANDLE CreateEvent(  
    LPSECURITY_ATTRIBUTES lpssa,    // security privileges  
                                     // (default = NULL)  
    BOOL bManualReset,              // TRUE if event must be reset manually
```

```

        BOOL bInitialState,           // TRUE to create event in signaled
state
        LPTSTR lpzEventName );       // name of event (may be NULL)

```

```

BOOL SetEvent( HANDLE hEvent );

```

```

BOOL ResetEvent( HANDLE hEvent );

```

Using the `bInitialState` parameter, `CreateEvent` allows the new event to arrive in the world already signaled. The `Set` and `Reset` functions return `TRUE` or `FALSE` to indicate success or failure.

With the `bManualReset` parameter, `CreateEvent` lets you create an automatic reset event instead of a manual reset event. An automatic reset event returns to its unsignaled state immediately after a `SetEvent` command. `ResetEvent` is redundant for an auto-reset event. Furthermore, an automatic reset button always releases only a single thread on each signal before resetting. An auto-reset event might be useful for a program where one master thread prepares data for other working threads. Whenever a new set of data is ready, the master sets the event and a single working thread is released. The other workers continue to wait in line for more assignments.

Besides setting and resetting events, you can pulse events.

```

BOOL PulseEvent( hEvent );

```

A pulse turns the signal on for a very short time and then turns it back off. Pulsing a manual event allows all waiting threads to pass and then resets the event. Pulsing an automatic event lets one waiting thread pass and then resets the event. If no threads are waiting, none will pass. Setting an automatic event, on the other hand, causes the event to leave its signal on until some thread waits for it. As soon as one thread passes, the event resets itself.

NOTE:

The sample named pipe program in Chapter 7 demonstrates the use of automatic and manual reset events.

Sharing and Destroying Mutexes, Semaphores, and Events

Processes, even unrelated processes, can share mutexes, semaphores, and events. By sharing objects, processes can coordinate their activities, just as threads do. There are three mechanisms for sharing. One is inheritance, where one process creates another and the new process receives copies of the parent's handles. Only those handles marked for inheritance when they were created will be passed on.

The other methods involve calling functions to create a second handle to an existing object. Which function you call depends on what information you already have. If you have handles to both the source and destination processes, call `DuplicateHandle`. If you have only the name of the object, call one of the `Open` functions. Two programs might agree in advance on the name of the object they share, or one might pass the name to the other through shared memory (see Chapter 10, "Memory Management") or a pipe (see Chapter 7).

```

BOOL DuplicateHandle(
    HANDLE hSourceProcess, // process that owns the original object
    HANDLE hSource,        // handle to the original object
    HANDLE hTargetProcess, // process that wants a copy of the handle
    LPHANDLE lpTarget,     // place to store duplicated handle
    DWORD   fdwAccess,     // requested access privileges
    BOOL    bInherit,      // may the duplicate handle be inherited?
    DWORD   fdwOptions ); // optional actions, e.g., close source handle

```

```

HANDLE OpenMutex(
    DWORD fdwAccess, // requested access privileges
    BOOL bInherit,   // TRUE if children may inherit this handle
    LPTSTR lpzName ); // name of the mutex

```

```

HANDLE OpenSemaphore(

```

```
DWORD fdwAccess,      // requested access privileges
BOOL bInherit,        // TRUE if children may inherit this handle
LPTSTR lpzName );     // name of the semaphore
```

```
HANDLE OpenEvent(
    DWORD fdwAccess,      // requested access privileges
    BOOL bInherit,        // TRUE if children may inherit this handle
    LPTSTR lpzName );     // name of the event
```

NOTE:

By the way, those LPTSTR variable types are not a misprint. LPTSTR is a generic text type that compiles differently depending on whether an application uses Unicode or ASCII strings.

Mutexes, semaphores, and events persist in memory until all the processes that own them end or until all the object's handles have been closed with CloseHandle.

```
BOOL CloseHandle( hObject );
```

Working with Critical Sections

A critical-section object performs exactly the same function as a mutex except that critical sections may not be shared. They are visible only within a single process. Critical sections and mutexes both allow only one thread to own them at a time, but critical sections work more quickly and involve less overhead.

The functions for working with critical sections do not use the same terminology as they do for mutexes, but they do roughly the same things. Instead of creating a critical section, you *initialize* it. Instead of waiting for it, you *enter* it. Instead of releasing it, you *leave* it. Instead of closing its handle, you *delete* the object.

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION lpcs );
VOID EnterCriticalSection( LPCRITICAL_SECTION lpcs );
VOID LeaveCriticalSection( LPCRITICAL_SECTION lpcs );
VOID DeleteCriticalSection( LPCRITICAL_SECTION lpcs );
```

The variable type LPCRITICAL_SECTION names a pointer (not a handle) to a critical-section object. InitializeCriticalSection expects to receive a pointer to an empty object—&amps —, which you can allocate like this:

```
CRITICAL_SECTION cs;
```

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

An Example with Multiple Threads: The *Threads* Program

The *Threads* demo, shown in Figure 6.3, puts into code some of the ideas explained in this chapter. It creates four secondary threads, each of which draws randomly sized and colored rectangles in a child window until the program ends.



FIGURE 6.3 The Threads demo

The top of the window contains a list box showing information about all four threads. By selecting a thread and choosing a menu command, you can suspend, resume, and change the priority of any thread. From the Options menu, you can also activate a mutex so only one thread draws at a time.

Initialization Procedures

The initialization procedures register two window classes, one for the main overlapping window and one for the child windows where the threads draw. They also create a timer. At five-second intervals, the list box updates the information about each thread. The CreateWindows function creates and positions all the windows, including the list box that shows information about each thread. The four threads are created during the WM_CREATE message handler.

TIP:

Note the absence of a PeekMessage loop in WinMain. That's a clear sign that you have entered the world of preemptive multitasking. The threads can draw continuously without monopolizing the processor. Other programs can still run at the same time. (The difference between preemptive and permissive multitasking is that permissive multitasking waits for a thread to relinquish control of the CPU while, in preemptive multitasking, the system interrupts executing threads to permit other threads access to the CPU.)

```
//-----
// WIN MAIN
//      Calls initializing procedures and runs the message loop
//-----

int WINAPI WinMain( HINSTANCE hinstThis, HINSTANCE hinstPrev,
                    LPSTR lpszCmdLine,  int iCmdShow )
{
    MSG msg;

    hInst = hinstThis;          // store in global variable
    if( ! InitializeApp() )
    {
        // if the application was not initialized, exit here
        return( 0 );
    }
    ShowWindow( hwndParent, iCmdShow );
    UpdateWindow( hwndParent );
        // receive and forward messages from our queue
    while( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    return( msg.wParam );
}
```

In addition to registering the application class and following the usual setup procedures, the InitializeApp procedure sets the thread priority and starts each thread in a suspended state.

```
//-----
// INITIALIZE APP
//      Register two window classes and then create the windows
//-----

BOOL InitializeApp ( void )
{
    ...
    // Mark the initial state of each thread as SUSPENDED.
    // That is how they will be created.
    for( iCount = 0; iCount < 4; iCount++ )
        iState[iCount] = SUSPENDED;
    // make primary thread more important to facilitate user i/o
    SetThreadPriority( GetCurrentThread(),
        THREAD_PRIORITY_ABOVE_NORMAL );
    // create all the windows
    return( CreateWindows( ) );
}
```

The call to SetThreadPriority increases the priority of the primary thread. If all the secondary threads were busy working at the same priority as the main thread, the menus would respond sluggishly. You can test this yourself by raising the priority of the secondary threads as the program runs.

The CreateWindows function creates not only the main window but also a list box and a series of child windows for the threads.

```
//-----
// CREATE WINDOWS
//      Create the parent window, the list box window, and the four
//      child windows.
//-----
```

```

BOOL CreateWindows()
{
    char    szAppName[MAX_BUFFER];
    char    szTitle[MAX_BUFFER];
    char    szThread[MAX_BUFFER];
    HMENU    hMenu;
    int     iCount;

    // load the relevant strings
    LoadString( hInst, IDS_APPNAME, szAppName,  sizeof(szAppName));
    LoadString( hInst, IDS_TITLE,  szTitle,    sizeof(szTitle));
    LoadString( hInst, IDS_THREAD,  szThread,   sizeof(szThread));

    // create the parent window
    hMenu = LoadMenu( hInst, MAKEINTRESOURCE(MENU_MAIN) );
    hwndParent = CreateWindow( szAppName, szTitle,
                               WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               NULL, hMenu, hInst, NULL );

    if( ! hwndParent ) return( FALSE );
    // create the list box
    hwndList = CreateWindow( "LISTBOX", NULL,
                             WS_BORDER | WS_CHILD | WS_VISIBLE |
                             LBS_STANDARD | LBS_NOINTEGRALHEIGHT,
                             0, 0, 0, 0, hwndParent, (HMENU)1,
                             hInst, NULL );

    if( ! hwndList ) return( FALSE );
    // create the four child windows
    for( iCount = 0; iCount < 4; iCount++ )
    {
        hwndChild[iCount] = CreateWindow( "ThreadClass", NULL,
                                           WS_BORDER | WS_CHILD |
                                           WS_VISIBLE | WS_CLIPCHILDREN,
                                           0, 0, 0, 0, hwndParent, NULL,
                                           hInst, NULL );

        if( ! hwndChild ) return( FALSE );
    }
    return( TRUE );
}

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Window and Message Handler Procedures

Most of the message-handler functions are simple. `Main_OnTimer` calls a procedure to clear the list box, generate four new strings of information, and display them. The `Main_OnSize` function suspends all the secondary threads while the program repositions the child windows to accommodate the new size. Otherwise, the busy threads would slow down the display operation. In addition to creating threads, `Main_OnCreate` creates a mutex.

```
/*-----
MAIN_WNDPROC
    All messages for the main window are processed here.
-----*/
LRESULT WINAPI Main_WndProc( HWND hWnd,           // message address
                             UINT uMessage,       // message type
                             WPARAM wParam,       // message contents
                             LPARAM lParam )      // more contents
{
    switch( uMessage )
    {
        HANDLE_MSG( hWnd, WM_CREATE, Main_OnCreate );
            // create the window and the threads
        HANDLE_MSG( hWnd, WM_SIZE, Main_OnSize );
            // reposition the child windows when the main window changes
        HANDLE_MSG( hWnd, WM_TIMER, Main_OnTimer );
            // update the list box every five seconds
        HANDLE_MSG( hWnd, WM_INITMENU, Main_OnInitMenu );
            // put check by Use Mutex menu item if bUseMutex is TRUE
        HANDLE_MSG( hWnd, WM_COMMAND, Main_OnCommand );
            // process menu commands
        HANDLE_MSG( hWnd, WM_DESTROY, Main_OnDestroy );
            // clean up and quit
        default:
```



```

        return( DefWindowProc(hWnd, uMessage, wParam, lParam) );
    }
    return( 0L );
}

```

The *Threads* program uses the message-cracker macro, `HANDLE_MSG`. As a result, the compiler may produce a series of warnings saying, “unreferenced formal parameter” for the message handlers. To avoid these, include this argsused pragma:

```

#ifdef __BORLANDC__
#pragma argsused
#endif

```

The `Main_OnCreate` function completes the process of initializing the several threads and setting up the mutex synchronization.

```

//-----
//  MAIN_ONCREATE
//      Create the four threads and set the timer
//-----
BOOL Main_OnCreate( HWND hWnd, LPCREATESTRUCT lpCreateStruct )
{
    UINT uRet;
    int iCount;
    // create the four threads, initially suspended
    for( iCount = 0; iCount < 4; iCount++ )
    {
        iRectCount[iCount] = 0;
        dwThreadData[iCount] = iCount;
        hThread[iCount] = CreateThread( NULL, 0,
                                     (LPTHREAD_START_ROUTINE) StartThread,
                                     (LPVOID) ( & ( dwThreadData[iCount] ) ),
                                     CREATE_SUSPENDED,
                                     (LPDWORD) ( & ( dwThreadID[iCount] ) ) );
        if( ! hThread[iCount] ) // was the thread created?
            return( FALSE );
    }
    // Create a timer with a five-second period.
    // The timer is used to update the list box.
    uRet = SetTimer( hWnd, TIMER, 5000, NULL );
    if( ! uRet ) return( FALSE ); // unable to create the timer
    // create a mutex synchronization object
    hDrawMutex = CreateMutex( NULL, FALSE, NULL );
    if( ! hDrawMutex )
    {
        // unable to create mutex
        KillTimer( hWnd, TIMER ); // stop the timer
        return( FALSE );
    }
    // start the threads with a priority below normal
    for( iCount = 0; iCount < 4; iCount++ )
    {
        SetThreadPriority( hThread[iCount],
                          THREAD_PRIORITY_BELOW_NORMAL );
        iState[iCount] = ACTIVE;
        ResumeThread( hThread[iCount] );
    }
    return( TRUE ); // Now all four threads are running!
}

```

Of course, the `Main_OnSize` function not only has to resize the application window but also needs to resize—and move—all of the child windows at the same time. This is because you have multiple child

windows and because the main application window is resizable.

```
//-----
//  MAIN_ONSIZE
//      Position the list box and the four child windows.
//-----
void Main_OnSize( HWND hWnd, UINT uState,
                  int cxClient, int cyClient )
{
    char* szText = "No Thread Data";
    int    iCount;

    // Suspend all active threads while the windows
    // resize and repaint themselves. This pause
    // enables the screen to update more quickly.
    for( iCount = 0; iCount < 4; iCount++ )
    {
        if( iState[iCount] == ACTIVE )
            SuspendThread( hThread[iCount] );
    }

    // place the list box across the top 1/4 of the window
    MoveWindow( hWndList, 0, 0, cxClient, cyClient / 4, TRUE );
    // Spread the 4 child windows across the bottom 3/4 of the
    // window. (The left border of the first one should be 0.)
    MoveWindow( hWndChild[0], 0, cyClient / 4 - 1, cxClient / 4 + 1,
                cyClient, TRUE );
    for( iCount = 1; iCount < 4; iCount++ )
        MoveWindow( hWndChild[iCount], (iCount * cxClient)/4,
                    cyClient/4 - 1, cxClient/4 + 1, cyClient, TRUE );
    // Add the default strings to the list box, and initialize
    // the number of figures drawn to zero
    for( iCount = 0; iCount < 4; iCount++ )
    {
        iRectCount[iCount] = 0;
        ListBox_AddString( hWndList, szText );
    }
    ListBox_SetCurSel(hWndList, 0);
    // reactivate the threads that were suspended while redrawing
    for( iCount = 0; iCount < 4; iCount++ )
    {
        if( iState[iCount] == ACTIVE )
            ResumeThread( hThread[iCount] );
    }
    return;
}
```

A timer message may not be the optimum choice for updating the list box. Ideally, the operations responsible for causing a change that needed to be reported would direct this event. In this case, however, a timer is a simple provision for accomplishing the task.

```
//-----
//  MAIN_ONTIMER
//      Process the timer message by updating the list box.
//-----
void Main_OnTimer( HWND hWnd, UINT uTimerID )
{
    UpdateListBox(); // update the data shown in the list box
    return;
}
```

The Main_OnInitMenu function is simply used to check (or uncheck) the Use Mutex menu command.

```
//-----  
//  MAIN_ONINITMENU  
//    Check or uncheck the Use Mutex menu item based on the  
//    value of bUseMutex  
//-----  
void Main_OnInitMenu( HWND hwnd, HMENU hMenu )  
{  
    CheckMenuItem( hMenu, IDM_USEMUTEX, MF_BYCOMMAND |  
        (UINT)( bUseMutex ? MF_CHECKED : MF_UNCHECKED ) );  
    return;  
}
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Main_OnCommand function is provided to parcel out all messages that have not already been handled by message crackers. (The 'message crackers' are found in the Main_WndProc where the HANDLE_MSG macros are used to redirect messages to the appropriate functions.)

```
//-----  
//  MAIN_ONCOMMAND  
//      Respond to commands from the user  
//-----  
void Main_OnCommand( HWND hWnd, int iCmd, HWND hwndCtl, UINT uCode )  
{  
    switch( iCmd )  
    {  
        case IDM_ABOUT:        // display the about box  
            MakeAbout( hWnd );  
            break;  
  
        case IDM_EXIT:          // exit this program  
            DestroyWindow( hWnd );  
            break;  
  
        case IDM_SUSPEND:       // modify priority or state of a thread  
        case IDM_RESUME:  
        case IDM_INCREASE:  
        case IDM_DECREASE:  
            DoThread( iCmd ); // adjust a thread  
            break;  
  
        case IDM_USEMUTEX:      // toggle the use of the mutex  
            ClearChildWindows( ); // make all thread windows white  
            bUseMutex = !bUseMutex; // toggle mutex setting  
            break;  
    }  
}
```

```
        default:
            break;
    }
    return;
}
```

Modification Procedures

The DoThread procedure responds to menu commands by modifying whichever thread is currently selected in the list box. DoThread can raise or lower a thread's priority, and suspend or resume threads. The iState array records the current state, either active or suspended, of each thread. The hThreads array holds handles to each of the four secondary threads.

```
//-----
// DO THREAD
//      Modify a thread's priority or change its state in response
//      to commands from the menu.
//-----
void DoThread( int iCmd )
{
    int iThread;
    int iPriority;

    // determine which thread to modify
    iThread = ListBox_GetCurSel( hwndList );
    switch( iCmd )
    {
        case IDM_SUSPEND:    // if thread is not suspended, then suspend
            if( iState[iThread] != SUSPENDED )
            {
                SuspendThread( hThread[iThread] );
                iState[iThread] = SUSPENDED;
            }
            break;

        case IDM_RESUME:     // if thread is not active, then activate
            if( iState[iThread] != ACTIVE )
            {
                ResumeThread( hThread[iThread] );
                iState[iThread] = ACTIVE;
            }
            break;

        case IDM_INCREASE:   // Increase the thread's priority (unless
                                // it is already at the highest level)
            iPriority = GetThreadPriority( hThread[iThread] );
            switch( iPriority )
            {
                case THREAD_PRIORITY_LOWEST:
                    SetThreadPriority( hThread[iThread],
                                        THREAD_PRIORITY_BELOW_NORMAL );
                    break;

                case THREAD_PRIORITY_BELOW_NORMAL:
                    SetThreadPriority( hThread[iThread],
                                        THREAD_PRIORITY_NORMAL );
                    break;

                case THREAD_PRIORITY_NORMAL:
                    SetThreadPriority( hThread[iThread],
                                        THREAD_PRIORITY_ABOVE_NORMAL );
                    break;
            }
            break;
    }
}
```

```

        break;

    case THREAD_PRIORITY_ABOVE_NORMAL:
        SetThreadPriority( hThread[iThread],
                           THREAD_PRIORITY_HIGHEST );
        break;

    default:
        break;
}
break;

case IDM_DECREASE: // Decrease the thread's priority (unless
                   // it is already at the lowest level)
    iPriority = GetThreadPriority( hThread[iThread] );
    switch( iPriority )
    {
        case THREAD_PRIORITY_BELOW_NORMAL:
            SetThreadPriority( hThread[iThread],
                               THREAD_PRIORITY_LOWEST );
            break;

        case THREAD_PRIORITY_NORMAL:
            SetThreadPriority( hThread[iThread],
                               THREAD_PRIORITY_BELOW_NORMAL );
            break;

        case THREAD_PRIORITY_ABOVE_NORMAL:
            SetThreadPriority( hThread[iThread],
                               THREAD_PRIORITY_NORMAL );
            break;

        case THREAD_PRIORITY_HIGHEST:
            SetThreadPriority( hThread[iThread],
                               THREAD_PRIORITY_ABOVE_NORMAL );
            break;

        default:
            break;
    }
    break;

    default:
        break;
}
return;
}

```

Thread Procedures

When `Main_OnCreate` constructs the secondary threads, it passes a pointer to the `StartThread` function in each call to `CreateThread`. `StartThread` becomes the main procedure for all the threads and the point where they begin and end execution.

If `bUseMutex` is `TRUE`, then the threads will wait to acquire the mutex before they draw and only one thread will draw at a time.

```

//-----
//  START THREAD
//      This is called when each thread begins execution
//-----
LONG StartThread( LPVOID lpThreadData )
{
    DWORD *pdwThreadID; // pointer to DWORD for storing thread's ID

```

```

DWORD dwWait;          // return value from WaitForSingleObject

pdwThreadID = lpThreadData;    // retrieve the thread's ID
// draw continuously until bTerminate becomes TRUE
while( ! bTerminate )
{
    if (bUseMutex)            // are we using the mutex?
    {
        // draw when this thread gets the mutex
        dwWait = WaitForSingleObject( hDrawMutex, INFINITE );
        if( dwWait == 0 )
        {
            DrawProc( *pdwThreadID );    // draw rectangles
            ReleaseMutex( hDrawMutex );    // let someone else draw
        }
    }
    else                      // not using mutex; let the thread draw
        DrawProc( *pdwThreadID );
}
// This return statement implicitly calls ExitThread.
return( 0L );
}

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The DrawProc procedure draws a series of rectangles. However, to avoid the overhead of passing many small messages between a program and the Win32 subsystem, *Graphics Device Interface (GDI)* calls do not always occur immediately. For this reason, graphics commands are held in a queue and periodically flushed. These delays somewhat exaggerate the effect of changing priorities in the *Threads* demo program.

```
//-----  
// DRAW PROC  
// Draw five random rectangles or ellipses  
//-----  
void DrawProc ( DWORD dwID )  
{  
    ...  
    if (bUseMutex)          // If only one thread draws at a time,  
        iTot = 50;          // let it draw more shapes at once.  
    else  
        iTot = 1;  
    srand( iRandSeed++ );   // reseed random generator  
                             // get window's dimensions  
    bError = GetClientRect( hwndChild[dwID], &amprcClient );  
    if( ! bError ) return;  
    cxClient = rcClient.right - rcClient.left;  
    cyClient = rcClient.bottom - rcClient.top;  
    // do not draw if the window does not have any dimensions  
    if( ( ! cxClient ) || ( ! cyClient ) )  
        return;  
    hDC = GetDC( hwndChild[dwID] ); // get device context for drawing  
    if( hDC )  
    {  
        // draw the five random figures  
        for( iCount = 0; iCount < iTot; iCount++ )  
        {  
            iRectCount[dwID]++;  
            xStart = (int)( rand() % cxClient ); // set coordinates
```

```

xStop = (int)( rand() % cxClient );
yStart = (int)( rand() % cyClient );
yStop = (int)( rand() % cyClient );
iRed   = rand() & 255;           // set the color
iGreen = rand() & 255;
iBlue  = rand() & 255;

hBrush = CreateSolidBrush(        // create a solid brush
    GetNearestColor( hDC,         // avoid dithered colors
        RGB( iRed, iGreen, iBlue ) ) );
hbrOld = SelectBrush( hDC, hBrush );

Rectangle( hDC,
    min( xStart, xStop ), max( xStart, xStop ),
    min( yStart, yStop ), max( yStart, yStop ) );
// delete the brush
DeleteBrush( SelectBrush(hDC, hbrOld) );
}
// If only one thread is drawing at a time, clear
// the child window before the next thread draws
if( bUseMutex )
{
    SelectBrush( hDC, GetStockBrush(WHITE_BRUSH) );
    PatBlt( hDC, (int)rcClient.left, (int)rcClient.top,
        (int)rcClient.right, (int)rcClient.bottom,
        PATCOPY );
}
ReleaseDC( hwndChild[dwID], hDC );    // release the HDC
}
return;
}

```

After you have run the *Threads* program and experimented with priorities, turned the mutex on and off, and suspended and resumed a few threads, you might want to try running the Process Viewer that comes with the Win32 SDK. In Figure 6.4, you can see what Process Viewer says about Threads. Note that it shows five threads for the program because it includes the primary thread as well. Browsing with Process Viewer gives you a better sense of how Windows 2000 works.

WARNING:

Some process viewers written for Win9x/NT may not function correctly under Windows 2000.

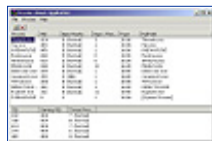


FIGURE 6.4 Examining Threads using the Win32 SDK Process Viewer

In addition to the Threads.exe process (top window) and the threads belonging to Threads.exe (bottom window), notice also that Eudora and WinWord have six threads executing, MSDEV has 10 threads, and Windows Explorer has 14.

NOTE:

The Process Viewer can be found at x:\Developer Tools\Common Tools\Tools\WinNT\Tools\Pview.exe, as part of your Visual Studio installation.

The *MultiThreads* Program

A second threads demonstration is found in the \MULTITHREAD_2 subdirectory, in the *MultiThreads* program, where three different types of thread objects—lines, rectangles, and circles—are combined in a

single window as shown in Figure 6.5.



FIGURE 6.5 Multiple threads in a single GDI context

Where the *Threads* demo uses four threads, each in a separate window, the *MultiThreads* demo uses any number of threads, all drawing objects in the same GDI window. When you run the *MultiThreads* demo, you should observe irregularities as different groups of threads execute and then rest, while other groups assume the focus.

In the demo program (or in the source code) you will also find an example of using a critical section to manage execution by controlling which thread has access to the GDI—in other words, preventing collisions between threads simultaneously trying to update the shared GDI. (This is discussed further in Chapter 9, “Exception Handling.”)

Also, for your attention, there are two options for closing threads: the `OnKillthreadsSlow` and `OnKillthreadsFast` operations.

The `OnKillthreadsSlow` operation is easy to understand, since the process of killing a thread is encapsulated in `KillThread`. However, if there are more than 50 threads running, the difference between this and the `OnKillthreadsFast` methods are quite noticeable.

The `OnKillthreadsFast` method tells all of the threads to go away, then waits for them to be scheduled and to terminate themselves. This is much faster than killing them individually and waiting for each one to terminate. Of course, this process is also a bit more complex, but the code should serve as its own explanation.

Summary

This chapter has explored the use of multiple threads with the *Threads* and *MultiThreads* programs, providing practical examples of threads in action.

In this discussion, I’ve mentioned processes and interprocess communication through pipes and shared memory. In Chapter 7, you’ll learn how to create and manage processes.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 7

Creating Processes and Pipes

- Child and parent processes
- Pipes for communication between processes
- Commands for creating and modifying processes and pipes
- Anonymous and named pipes

In Chapter 6, “Creating and Synchronizing Multiple Threads,” you saw how one process can create many threads. A program that needs to do several things at once usually creates threads rather than whole new processes. Threads are generally preferable to processes because they are created more quickly with less overhead, they share resources such as handles and variables, and they are easy to synchronize. This chapter shows what happens when you decide instead to divide a task among several distinct processes, each with its own threads and its own private address space.

Processes and pipes are closely related topics. Processes often pass pipe handles to new processes they create in order to establish a channel for exchanging information. Along with the threads and synchronization objects you read about in Chapter 6, pipes and processes form the basic core of tools for multitasking in Windows 2000. In this chapter, you’ll learn how to launch a new child process, pass it handles, connect the parent to the child with a pipe, and send information through the pipe.

Process Concepts: Division of Labor

The multitasking capabilities of both Windows 98 and Windows 2000 depend on the system’s handling of processes and *threads*. Both terms designate ways of dividing labor in the operating system; processes own resources, and threads execute instructions. The term “process” names an executable program and all the program’s resources as it runs in the system, including:

- Its virtual address space
- The access token that assigns it a privilege level

- The resource quotas determined by the user's privilege level
- One or more execution threads
- Other objects as they are assigned dynamically, such as an open file or a shared memory block

Every process has at least one thread of execution—one path the processor follows through its code. A process can also choose to divide its work into several tasks and create separate threads for each one. “Multitasking” happens when one machine runs several programs at once; “multithreading” happens when one program runs several threads at once. The Process Manager creates and destroys processes and their threads. It also suspends threads and resumes them.

Preemptive versus Permissive Multitasking

The simple difference between Windows 98 and Windows 2000 is that Win2000 uses a “preemptive multitasking” system while Win98 exercises “permissive multitasking.”

- In a preemptive multitasking system (WinNT/2000), the operating system summarily suspends executing threads to allow other threads an appropriate time slice for operation.
- In a permissive multitasking system (Win98), the operating system depends on the applications to regularly relinquish control of the operating system so that other applications have the opportunity to gain their time slices.

Obviously, a preemptive multitasking system has the ability to provide much smoother execution of multiple tasks and threads. This type of system does not need to depend on the good graces of each program (or the skills of the respective programmers) in surrendering control of the CPU.

For permissive multitasking, however, all applications—and all threads within those applications—must be well behaved and regularly relinquish control to permit other applications and other threads access to the CPU.

Inheritance

Documentation often refers to processes as *parents*, *children*, and *siblings*. A process that creates other processes is called the parent, and the new processes are its children. All the children of a single parent are siblings. However, once a parent creates a child, the child no longer depends on the parent and, if the parent terminates, the child may continue. The Win32 subsystem does not enforce any strong hierarchical dependence between parents and children.

The familial metaphor extends to one other action: inheritance. A new process *inherits* some attributes from its parent. Inherited attributes include the following:

- An environment block with environment variables
- A current working directory
- Standard input and output handles
- Any other handles the parent may want its children to have

A child of a “console process”—that is, a child of a parent that uses the console API for drawing on the screen instead of the Windows GUI—also inherits its parent's console window. A child may inherit handles to processes, threads, mutexes, events, semaphores, and pipes (see Chapter 5, “Pop-Ups: Tip Menus and Windows”) as well as file-mapping objects. A child process may also inherit handles created with `CreateFile`, including files, console input buffers, console screen buffers, serial communication devices, and mailslots.

NOTE:

Consoles simulate character-based displays. The Win98/2000 command shells, for example, run in a console window.

When a child inherits a handle, both the parent and the child end up with handles to the same object. Whatever one does to the object affects the other. If a child's thread waits for and acquires a shared mutex, any parent threads that wait for the same object will be blocked. If a parent writes to a file, a child using the same handle will find its file position marker has moved forward too.

When a child inherits a handle, it really inherits only access to the object. It does not inherit handle variables. When creating a child, the parent must both arrange for inheritance and pass the handles explicitly. These handles may be passed in several ways: on the command line, through a file, or through a pipe.

An easier but less intuitive option involves the standard I/O channels. Recall that character-based C programs frequently direct their I/O through three standard channels called `stdin`, `stdout`, and `stderr`. Win32 processes automatically possess the same three channels, although they do not have the same predefined names. The standard I/O channels are generally useless to a GUI application because they are not actually connected to any device. (A GUI application may, however, open a console window and use its standard I/O devices there.)

Normally a child inherits the same standard I/O devices its parent uses, but when creating the child, the parent may specify a different set of handles for the child to receive. The child retrieves its inherited handles with `SetStdHandle`. The parent may also change one or more of its own devices before creating the child. The child inherits the changes.

Children do not inherit all kinds of objects. Memory handles and pseudohandles are excluded from inheritance, for example. Each process has its own address space, so it cannot inherit memory objects from another process. Pseudohandles, such as the value returned by `GetCurrentThread` are, by definition, valid only in the place where they originate. Nor do children inherit DLL module handles, GDI handles, or USER handles, including `HBRUSH` and `HWND` objects. Similarly, children do not inherit their parent's priority class. By default, a child's priority will be `NORMAL_PRIORITY_CLASS`. There is one exception: The `IDLE_PRIORITY_CLASS` is inheritable. By default, low-priority parents create low-priority children, while parents of any other priority create children of normal priority.

Life Cycle of a Process

Process resources include an access token, a virtual address space description, and a handle table. Every process has a primary thread, which is the one that begins execution at `WinMain`, and may create other threads as it runs.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

All the threads in one process share a single image of the executable file, but each new process requires the system to map the executable file into a new address space. The command for creating a process asks for the name of an .EXE file. An application that creates child processes to perform part of its work must be written and compiled as several distinct programs.

All Win32 processes begin life when some other process invokes them with CreateProcess. The Win32 subsystem starts up all its clients with that command. During CreateProcess, the system sets up a virtual address space, loads an executable file image, and creates a primary thread. CreateProcess returns a handle to the new process and a handle to its primary thread. The parent may close the handles, relinquishing control over its child. Or the parent may keep them to change the child's priority class, terminate the child, or read the exit code when the child ends by itself. Like threads, even after they've finished executing, processes remain in memory until all handles to the process have been closed.

A process remains active until its primary thread reaches the end of its starting procedure and stops executing or until any of its threads call ExitProcess. ExitProcess causes the system to notify all supporting DLLs that this module has stopped. It also causes any other threads in the same process to terminate immediately.

Pipe Concepts: Communication between Processes

Quite frequently, the processes (applications) that the user runs have nothing to do with each other. However, there are circumstances where the ability to exchange information among processes could produce a synergistic effect allowing applications together to accomplish tasks neither program could manage alone.

For example, a phone dialer and a paint program probably have little in common, but if we wanted the paint program to dial a remote connection to transfer information, a link to a phone dialer would make perfect sense. Windows allows many channels of communication, including the clipboard, DDE, and OLE, while Windows 2000 offers additional channels. For example, in Chapter 9, "Exception Handling," you'll see how processes can share memory by opening views onto the same memory-mapped file.

Pipes are an easy mechanism for different programs to use for exchange of information. Unlike some other channels, pipes have no formal standards or protocols to govern how information is passed. That makes pipes easier to use and more flexible than, say, DDE conversations, but it also limits them to programs that recognize each other and know how to parse the information they agree to exchange.

TIP:

The use of pipes was originally introduced under Windows NT and later supported by Windows 95/98 in the form of anonymous pipes. The use of named pipes, however, remains the *parvenu* of Windows NT/2000 and is not supported by Win95/98.

A pipe is a section of shared memory where processes leave messages for each other. A pipe resembles a file where one program writes and another reads. Because a pipe is dedicated to interprocess communication, the Win32 API can provide a range of commands to facilitate the exchange of information. Conceptually, a pipe is a cross between a computer file and a post office mailbox. One process writes something in the file, and another process looks to see what was left behind.

The Life Cycle of a Pipe

A pipe appears when one program decides to create it. The program that creates a pipe is called the pipe's server. Other processes, called clients, may connect to the pipe's other end. The server assumes responsibility for the life of the pipe. Any process may be a server or a client, or both at once on different pipes.

After the pipe is created and another process connects to it, either the client or the server, or sometimes both the client and the server, write into its end. Anything written at one end is read from the other. Reading and writing operations rely on the same commands you use to work with any file: `ReadFile` and `WriteFile`. Typically, a process that expects to receive a series of messages creates a new thread to wait for them. The thread repeatedly calls `ReadFile` and blocks, remaining suspended until each new message arrives.

Eventually the server decides the conversation has ended and breaks the connection by calling `CloseHandle` to destroy the pipe. (The pipe will not actually be destroyed until all handles to it, both the server's and the client's, have been closed.) Alternatively, the server may decide to connect the old pipe with a new client.

Varieties of Pipes

Pipes come in several varieties:

- Inbound, outbound, or duplex
- Byte or message
- Blocking or non-blocking
- Named or anonymous

Most of these attributes are determined when the pipe is created.

Inbound, Outbound, and Duplex Pipes

The first set of terms—inbound, outbound, and duplex—distinguishes the direction in which information flows through the pipe. *Inbound* and *outbound* describe one-directional pipes, where one side only writes and the other side only reads. An inbound pipe lets the client send and the server receive. An outbound pipe lets the server send and the client receive. A *duplex* pipe allows both sides to send and receive.

Byte and Message Pipes

What the participants write determines whether the pipe should have a reading mode of type *byte* or type *message*. The reading mode helps the system decide when a read operation should stop. With a byte-mode pipe, a read operation stops either when it reaches the last byte of data in the pipe or when it fills its reading buffer. With a message-mode pipe, however, a read operation stops when it reaches the end of a single message.

Internally, the system marks messages in a message-mode pipe by prefacing each newly written segment with a header stating its length. The programs on either end of the pipe never see the message headers, but `ReadFile` commands on a message-mode pipe automatically stop when they reach the end of a segment.

Blocking and Non-Blocking Pipes

Pipes may also be either *blocking* or *non-blocking*. This attribute affects read, write, and connect commands. When any of these commands fail on a non-blocking pipe, the command returns immediately with an error result. On a pipe that allows blocking, the commands do not return until they succeed or an error occurs. Table 7.1 summarizes the ways in which blocking and non-blocking modes affect three operations.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 7.1: The Effects of Blocking and Non-Blocking Modes

Operation	Blocking Mode	Non-Blocking Mode
ConnectNamedPipe	Blocks until a client connects to the other end.	Returns FALSE immediately.
ReadFile	If the pipe is empty, blocks until a message arrives.	If the pipe is empty, returns an error immediately.
WriteFile	If the pipe is nearly full, blocks until another process reads from the other end.	If the pipe is nearly full, returns immediately. For a byte-mode pipe, WriteFile first writes as much as it can. For a message-mode pipe, WriteFile returns TRUE and writes no bytes.

Named and Anonymous Pipes

A pipe may be *named*, in which case the creator has endowed it with an identifying name string, or it may be *anonymous*, meaning it has no name string. Like synchronization objects, such as mutexes and semaphores, a pipe may be given a name to help other processes identify it.

Anonymous pipes require less overhead, but perform only a limited subset of the services that named pipes can perform. Anonymous pipes pass messages in only one direction—either server to client or client to server, but not both. Also, anonymous pipes do not work over networks; both the server and client must inhabit the same machine.

Named Pipes in Windows 95/98 versus Windows NT/2000

If you are accustomed to programming in the Win95/98 environment, one curiosity that you will encounter between Windows 95/98 and Windows NT/2000 lies in the use of named pipes. Under Windows 98, you cannot compile an application to use named pipes. Or, more accurately, you can compile the application—without error—but any calls to the `CreateNamedPipe` API, which is integral to using named pipes, will simply fail.

The curious side of this is that you can compile the application under Windows NT and once it has been compiled, it will run under Windows 98, executing the `CreateNamedPipe` API by using files as shown in the *AnonPipe* demo example.

Rather unhelpfully, if you debug the *Named Pipe* example under Windows 95/98, the `ShowErrorMsg` function will display a system error message reporting: “This function is valid only in Win32 mode.”

Of course, for applications created under Windows 95/98, the interprocess communication functions demonstrated in the *AnonPipe* example still remain valid.

Named pipes can do several things that anonymous pipes cannot. They can pass information in both directions through one pipe, connect across a network to a process on a remote machine, exist in multiple instances, and use more pipe modes. The ability to exist in multiple instances permits a named pipe to connect one server with many clients. Each instance is an independent channel of communication. Messages in one instance do not interfere with messages in another instance. Multiple instances of a single pipe result when one or more servers pass the same identifying name to `CreateNamedPipe`.

Process-Related Commands

In many ways, the commands for creating and managing processes resemble the commands for creating and managing threads, which you saw in Chapter 6. To create a process, you specify security attributes and receive a handle. With the handle, you can manage the process; for example, you can change the priority of the process or terminate it. Even after a process stops running, it continues to exist as long as any handles to it remain open. To destroy a process, close all of its handles.

This section describes the Win32 API commands for creating processes. The two sample programs described in “Processes Communicating through a Pipe,” the final section of this chapter, demonstrate most of the commands explained here.

Making and Modifying Processes

In current versions of Windows (both 95/98 and NT/2000) the `CreateProcess` command is used to initiate every new process. While the old `WinExec` and `LoadModule` commands still exist, providing backwards compatibility, both of these now place internal calls to the `CreateProcess` function.

The `CreateProcess` function is called as:

```
BOOL CreateProcess(
    LPCTSTR lpszImageName,           // image file (.EXE) name
    LPCTSTR lpszCmdLine,             // command line for new process
    LPSECURITY_ATTRIBUTES lpsaProcess, // how process will be shared
    LPSECURITY_ATTRIBUTES lpsaThread, // how new threads will be shared
    BOOL bInheritHandles,           // TRUE to inherit handles
    DWORD fdwCreate,                 // creation flags
    LPVOID lpvEnvironment,          // new environment (default =
NULL)
    LPTSTR lpszCurrentDir,           // name of new current directory
    LPSTARTUPINFO lpStartupInfo,     // gives info about new process
    LPPROCESS_INFORMATION lpProcInfo // receives info about new// process
```

The Parameters

The `CreateProcess` function’s parameters permit control over the new process’s starting conditions. First, every process requires code to execute. Together, the first two parameters create a complete command line naming a program file and passing it arguments.

Executable and Arguments The first parameter, `lpzImageName`, must point only to the name of the executable file (for example, `Child`) and must not include any arguments. In locating a file named in this parameter, the system does not search the `PATH` directories. The program must be in the current directory, or the string must contain a full path name.

Caveat on Testing the *AnonPipe* Demo

To test the *AnonPipe* demo, you will want to begin by compiling both the Parent and Child executables. Once these are compiled, you can execute the Parent program—in the `Debug` directory—and use the Parent program to launch the Child process. However, if you attempt to execute the Parent program from the Developer's Workshop (as for debugging), you will receive a message stating: "The system cannot find the file specified."

However, if you first copy the compiled `Child.exe` program from the `/Debug` subdirectory to the `/AnonPipe` directory, the `Parent.exe` application will again be able to find the `Child.exe` program for launching.

The problem—if you are not already ahead of me—is that under the debugger, the `Parent.exe` application is not looking in the directory where it is located, but in the immediate root directory (where the source files are found).

On the other hand, when the Parent application is executed directly, it looks in the current directory to find the `Child.exe` program for launch.

For alternatives, refer to the notes immediately following on `CreateProcess` parameters and feel free to modify the *AnonPipe* `Parent.c` source to experiment with different arrangements.

The second parameter, `lpzCmdLine`, may be `NULL` if you have no arguments to pass. If you do pass arguments, the first item in the `lpzCmdLine` string must be what C programmers call the `argv[0]` value.

The first item is typically, but not necessarily, the name of the program, even if you have already given it in `lpzImageName`. You can omit the path and extension if they appear in the `lpzImageName`, but some item must precede the first argument.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

To make CreateProcess search along the environment PATH for the file, leave the first parameter blank and pass the complete command line in the second parameter. Table 7.2 lists some examples of using CreateProcess parameters.

TABLE 7.2: Using CreateProcess Parameters

First Parameter	Second Parameter	Result
d:\dev\bin\qgrep	NULL	Runs qgrep without arguments only if qgrep is found in the \dev\bin directory.
qgrep.exe	qgrep -L-y "ERROR_" *.h	Runs qgrep with arguments only if qgrep is in the current directory.
NULL	qgrep -L-y "ERROR_" *.h	Runs qgrep with arguments if it can be found anywhere on the path.

Security Attributes The next two parameters in CreateProcess provide security attributes for both the new process and its primary thread. (Every process receives an initial thread; otherwise, nothing would ever execute.) You saw the same SECURITY_ATTRIBUTES structure when you created new threads in Chapter 6, and you will see it often in commands that create new objects. The information in this structure controls what another process may do with the object if it opens a duplicate handle. It also controls whether child processes may inherit a handle to the new object. By default, other processes receive full access to the new object and children do not inherit it.

TIP:

Under Windows 98, the security attributes do not apply and are simply ignored. If your application is intended only for Windows 98 and not for the Windows NT/2000 environment or if no security is needed, this argument may be passed as NULL.

Blocking Inheritance The bInheritHandles parameter of CreateProcess offers a second chance to prevent a child from inheriting other handles already created. If bInheritHandles is FALSE, the new process inherits no

handles, even if they were marked inheritable when they were created. Blocking inheritance is useful because many objects, including threads and processes, persist in memory until all the handles to them are closed. If every new child inherits a full set of handles from its parent, many objects will stay in memory until all of the child processes exit and their handles are destroyed.

TIP:

Children should be allowed to inherit only the handles they actually need.

Process Type and Priority The creation flags in the `fdwCreate` parameter govern the type and priority of the new process. The new process may be initially suspended; it may have a console or a GUI window; it may receive debugging information from its child; and it may receive a particular priority class. Here are some of the values that can be combined in the creation flag parameter:

DEBUG_PROCESS The parent receives debugging information about the child and any of its children.

DEBUG_ONLY_THIS_PROCESS The parent receives debugging information about the child but not any of the child's children.

CREATE_SUSPENDED The primary thread of the new process is initially suspended.

DETACHED_PROCESS The new process does not use a console window.

CREATE_NEW_CONSOLE The new process receives its own new console window.

IDLE_PRIORITY_CLASS The process runs only when the system is idle.

NORMAL_PRIORITY_CLASS The process has no special scheduling needs.

HIGH_PRIORITY_CLASS The process preempts all threads of lower priority in order to respond quickly to some critical situation.

REALTIME_PRIORITY_CLASS The process preempts even important system tasks.

Environment and Directory Settings Each process has its own set of environment variables and its own current directory setting. If the `lpzEnvironment` and `lpzCurrentDir` parameters are NULL, the new process copies the block of environment variables and the directory setting from its parent. Environment variables are those defined with the SET command at the command prompt, typically in an autoexec file. Programmers usually define variables, such as BIN, LIB, and INCLUDE, telling the compiler and linker where to find particular kinds of files.

More generally, environment variables are a convenient way to customize programs by putting information where it is always available. A parent can send information to its children by defining environment variables. To give the child an entirely new environment, the parent should create a buffer and fill it with null-terminated strings of the form `<variable>=<setting>`.

TIP:

You should be careful to ensure that no spaces appear on either side of the equals sign. For example, the form `<variable> = <setting>` is invalid and will not be accepted.

The last string must be followed by two null characters. To give the child a slightly modified version of the parent's existing environment, the parent can temporarily modify its own settings with `GetEnvironmentVariable` and `SetEnvironmentVariable`, create the child, and then restore the old settings.

Pointing to Structures The last two parameters of `CreateProcess` point to structures. The parent fills out the `STARTUPINFO` structure before calling `CreateProcess` and receives information about the new process in the `PROCESS_INFORMATION` structure.

```
// You fill this out to describe a new process in advance
typedef struct _STARTUPINFO /* si */
{
    DWORD    cb;                // size of(STARTUPINFO)
    LPTSTR   lpReserved;        // should be NULL
    LPTSTR   lpDesktop;         // name of desktop object to run in
    LPSTR    lpTitle;           // caption for console title bar
    DWORD    dwX;               // left corner for new window
```



```

DWORD    dwY;                // top corner for new window
DWORD    dwXSize;            // width of new window
DWORD    dwYSize;            // height of new window
DWORD    dwXCountChars;      // width of new console window
DWORD    dwYCountChars;      // height of new console window
DWORD    dwFillAttribute;     // text/background colors for console
DWORD    dwFlags;            // activates fields in this structure
WORD     wShowWindow;         // iCmdShow parameter value
WORD     cbReserved2;         // zero
LPBYTE   lpReserved2;        // NULL
HANDLE    hStdInput;          // handles for the
HANDLE    hStdOutput;         //   child's standard
HANDLE    hStdError;          //   I/O devices
} STARTUPINFO, *LPSTARTUPINFO;

```

You must fill in the first field of the STARTUPINFO structure, but you can initialize all the rest to 0 or NULL to accept default values.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

The lpDesktop field is ignored under Windows 95/98, but under Windows NT, lpDesktop points to a zero-terminated string specifying either the name of the Desktop only or the name of both the window station and Desktop for this process. The Desktop is the background window on which a user's programs run. The current system allows only two Desktops: one for logging on and one for the current user.

A backslash in the string pointed to by lpDesktop indicates that the string includes both Desktop and window-station names. Otherwise, the lpDesktop string is interpreted as a Desktop name. If lpDesktop is NULL, the new process inherits the window station and Desktop of its parent process.

NOTE:

Several features of the API, such as this lpDesktop field, suggest that in later releases, NT/2000 may allow one user to work with multiple Desktop windows. Until then, Desktop-related fields and parameters serve little purpose. To hide the user's data while the machine is idle, screen savers run on the log-on Desktop for security. Under NT/2000, screen savers do not have access to the user's Desktop.

The lpTitle field is used only for console purposes, to supply a title in the title bar when a new console window is created. If lpTitle is NULL, the name of the executable is used instead. For GUI or console processes that do not create a new console window, this parameter should be NULL.

Many of the other fields in a STARTUPINFO structure matter only for non-graphics processes that will run in console windows instead of regular GUI windows. Most of the fields are ignored unless the values in dwFlags alert the system to use them. dwFlags may contain the values listed in Table 7.3, each value activating a different field or set of fields from the STARTUPINFO structure.

TABLE 7.3: dwCreationFlag Values in STARTUPINFO

Flag	Field(s) Activated
STARTF_USESHOWWINDOW	wShowWindow
STARTF_USESIZE	dwXSize, dwYSize
STARTF_USEPOSITION	dwX, dwY
STARTF_USECOUNTCHARS	dwXCountChars, dwYCountChars
STARTF_USEFILLATTRIBUTE	dwFillAttribute

You don't need to initialize any of these 11 fields unless you activate them with a flag in dwFlags.

Three additional values for dwFlags do not activate specific fields. The STARTF_FORCEONFEEDBACK and STARTF_FORCEOFFFEEDBACK values force the system to display or omit the waiting cursor that gives the user feedback as an application starts up.

I/O Handles The last three fields allow you to specify standard I/O handles for the child that differ from those of the parent. Normally, the child inherits whatever I/O devices the parent has. The I/O handles provide an easy way to pass the child any handles it needs to receive, such as one end of a pipe. If you use any of the fields, you should set values in all of them. The child receives an invalid handle for any device you leave NULL. Call GetStdHandle to copy any of the parent's standard device handles into these fields.

NOTE:

A process created with the DETACHED_PROCESS flag cannot inherit its parent's standard I/O devices. The console program initialization procedures do not correctly receive the devices when the process has no console. Microsoft identified this limitation in an early release of NT and added the handle fields as a workaround. While these work for any child, they are necessary only for detached children.

Return Value

CreateProcess returns TRUE if it succeeds in creating the new object and FALSE if an error occurs. If CreateProcess returns TRUE, it also returns information about the new process and its primary thread in the final parameter, the PROCESS_INFORMATION structure.

```
typedef struct _PROCESS_INFORMATION /* pi */
{
    HANDLE hProcess;           // handle to the new process
    HANDLE hThread;           // handle to its primary thread
    DWORD  dwProcessId;       // number identifying new process
    DWORD  dwThreadId;        // number identifying new thread
} PROCESS_INFORMATION;
```

If your program does not make use of the handles for the new process and its primary thread, you should close both right away. Otherwise, even if the PROCESS_INFORMATION structure is a local variable and goes out of scope, abandoning whatever it contained, the two object entries remain in your process's object table and the system counts them as open handles.

Because the size of physical system memory limits the total number of processes that can be created, be sure to check for error returns. As an example, one of the early beta versions of Windows NT allowed a 16MB machine to create about 40 processes before failing for lack of memory. While a 16MB machine is almost unheard of today, this limit will vary from version to version and machine to machine, but it is finite. Memory shortage can also cause CreateThread to fail, though threads consume significantly fewer resources than processes.

C Runtime Equivalents

The spawn and exec functions in the C runtime library also create new processes. Internally, however, they map to the same Win32 CreateProcess call. Through its parameters, CreateProcess offers more ways to customize the new process than do the C functions. The C runtime functions _getenv and _putenv, which work with a process's environment variables, also duplicate the Win32 API functions GetEnvironmentVariable and PutEnvironmentVariable.

Getting Information about a Process

Once a process starts up, it can call several commands to find out about itself.

Getting the Handle and ID

Like threads, a handle and an ID number identify processes with the parent receiving both from the CreateProcess call. The child, however, receives these by calling GetCurrentProcess and GetCurrentProcessId. Like

GetCurrentThread, GetCurrentProcess returns a pseudohandle valid only in the current process. The two functions are called as:

```
HANDLE GetCurrentProcess( void );  
DWORD GetCurrentProcessId( void );
```

TIP:

Pseudohandles may not be passed to other processes. To convert a pseudohandle to a real handle, use DuplicateHandle.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Getting the Environment Settings

The `GetEnvironmentVariable` function retrieves the environment settings inherited from the parent.

```
DWORD GetEnvironmentVariable(  
    LPTSTR lpszName,      // name of environment variable  
    LPTSTR lpszValue,     // address of buffer for variable value  
    DWORD  dwValue );    // size of lpszValue buffer in characters
```

When the `lpszName` buffer contains a variable name, such as `PATH`, the function looks up the corresponding value and copies it into `lpszValue`. The `DWORD` return value tells how many characters it copied into the `lpszValue` buffer, or 0 if the variable was not found.

Getting the Command Line

The `GetCommandLine` function retrieves a pointer to the command line, but the same information is usually available by other means as well. Console programs written in C can read the command line using `argc` and `argv`; GUI programs can retrieve it through the `lpszCmdLine` parameter of `WinMain`.

```
LPTSTR GetCommandLine( void );
```

Changing Process Priority

With the creation flags in `CreateProcess`, the parent can assign one of six base priority classes to a new process:

- `IDLE_PRIORITY_CLASS`
- `NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `REALTIME_PRIORITY_CLASS`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`

WARNING:

The default priority class is NORMAL. Applications should not use the HIGH class, and especially not the REALTIME class, unless absolutely necessary. Both levels impair the performance of lower-priority processes. Programs that do run at higher priorities should do so only for short periods of time.

Here are the functions to find out and modify a process's priority class:

```
DWORD GetPriorityClass( HANDLE hProcess );

BOOL SetPriorityClass(
    HANDLE hProcess,          // process to modify
    DWORD fdwPriority );      // new priority class
```

The DWORD values in both functions should be one of the four `PRIORITY_CLASS` flags listed earlier. The priority class of a process becomes the base priority for all of its threads. Refer to the discussion of thread priorities in Chapter 5 to see how the base priority influences a thread's dynamic priority.

Synchronizing Processes

Chapter 5 also explained how threads coordinate their separate tasks by waiting for signals from synchronization objects. Besides the four standard synchronization objects—mutexes, semaphores, events, and critical sections—threads can wait for other threads and for files, timers, and processes. Waiting for a thread or a process means waiting for it to stop execution. A thread waits for a process by passing the process handle to `WaitForSingleObject` or `WaitForMultipleObjects`. When the process terminates, it enters its signaled state and all threads waiting for it resume execution.

One other synchronization command works only when waiting for processes. Instead of waiting for the process to terminate, you can wait for it to be idle. For this purpose, a process is considered idle when it has finished initializing and no user input is waiting to reach it.

```
DWORD WaitForInputIdle(
    HANDLE hProcess,          // process to wait for
    DWORD dwTimeout );        // time-out time in milliseconds
```

NOTE:

The process handle must have `PROCESS_SET_INFORMATION` access rights before a new priority can be assigned.

Parents frequently call `WaitForInputIdle` immediately after creating a new process to allow the child time to establish itself. When the new process initializes and reaches its idle state, the parent can try to communicate with it.

The value returned by `WaitForInputIdle` depends on how it ends. It returns 0 for a successful wait when the process becomes idle. If the time-out period elapses before the process idles, `WaitForInputIdle` returns `WAIT_TIMEOUT`. To indicate an error, it returns `(HANDLE) 0xFFFFFFFF` (-1, a.k.a. `INVALID_HANDLE_VALUE`).

NOTE:

`WaitForInputIdle` tests the child's message queue for pending messages. It is intended only for GUI applications. Character-based console applications, lacking a message queue, are *always* idle by this definition.

Sharing Handles

Like handles to threads and synchronization objects, handles to processes can be shared. Several different processes might possess handles to any one process. As usual, you can't simply pass a handle directly from one process to another. You must instead rely on one of several transfer mechanisms to perform the conversion that makes a handle valid in a new address space.

One mechanism is inheritance. If you tell `CreateProcess` to let the child inherit handles, the child receives copies of all the inheritable handles in its own object table. Children cannot use the handles there

directly—they must still receive the handle on their command line or through a pipe—but inheritance makes the handle valid when it reaches the child process.

WARNING:

If a child process is passed a handle it has not already inherited, the handle will not work.

Making a Handle Copy

Inheritance helps in passing handles only between related processes. To create a handle that can be passed to any other process, related or unrelated, call `DuplicateHandle` as discussed in Chapter 6. Given the original handle and a source and destination process, `DuplicateHandle` creates a new handle valid in the destination process. `DuplicateHandle` also allows you to modify the attributes of a handle you want to keep. If, for example, you want only one of several children to inherit a particular handle, you create the first child, allowing inheritance, and then call `DuplicateHandle` to make a non-inheritable copy. Close the original inheritable handle and keep only the copy that will not be inherited by subsequent children.

One of the sample programs in this chapter uses `DuplicateHandle` to control inheritance. Look for the `StartProcess` procedure in the *AnonPipe* program described in the section “One-Way Communication: The Anonymous Pipe Version” later in this chapter and the complete program on the CD accompanying this book.

Opening a Process

The `OpenProcess` function allows any process to open a handle to any other process. The usual limits apply: Another process can be opened only if the security descriptor operating process has sufficient clearance.

TIP:

Security privileges apply only under Windows NT/2000 and not under Windows 95/98.

```
HANDLE OpenProcess(
    DWORD fdwAccess,          // desired access privileges
    BOOL bInherit,           // TRUE for children to inherit the handle
    DWORD dwProcessId );    // number identifying the process
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

OpenProcess requires the target process to be identified by its ID number. Normally, an application knows the process ID number only if it created the process, or if the process itself (or one of its relatives) passes the ID (through a pipe or a DDE conversation, for example) to the application.

TIP:

It is possible to generate a list of ID numbers for all the currently running processes, but the task is not trivial and involves enumerating information stored in the system registry under the HKEY_PERFORMANCE_DATA key. The online help file contains sample code showing how to search the registry with RegEnumKey, RegEnumValue, and RegQueryInfoKey. The structure of the HKEY_PERFORMANCE_DATA key is documented in *winperf.h*. The SDK comes with the source code for its Process Viewer utility (see Figure 6.4 in Chapter 6); it enumerates threads and processes.

Ending a Process

Normally you don't do anything special to end a process, just as you don't do anything to end a thread. When a thread reaches the return instruction at the end of its startup procedure, the system calls ExitThread and the thread terminates, leaving an exit code behind. When the primary thread in a process comes to the end of its starting procedure (usually WinMain), the system implicitly calls ExitProcess instead of ExitThread.

ExitProcess forces all the threads in a process to end, no matter what they may be doing at the time. Any thread, however, may call ExitProcess explicitly to end its process at any time.

```
void ExitProcess( UINT fuExitCode );
```

The process's exit code can be defined however the programmer chooses. Like threads, processes remain in memory even after they terminate, only dying when all the handles to them are closed. To determine a process's exit code, keep its handle and call GetExitCodeProcess.

```
BOOL GetExitCodeProcess(  
    HANDLE hProcess,           // handle to the process  
    LPDWORD lpdwExitCode );    // buffer to receive exit code
```

If the process has not ended, GetExitCodeProcess reports the exit code as STILL_ACTIVE.

Normally a process ends when its primary thread ends. The primary thread may, however, choose to quit without ending the process. If the primary thread ends with an explicit call to `ExitThread`, the system does not call `ExitProcess`. Other threads in the process continue to execute, and the process runs until any thread calls `ExitProcess` directly or until the last thread ends.

A number of things happen when a process ends:

- All of the process's handles are closed; all of its file handles, thread handles, event handles, and any other handles are destroyed. The objects they point to are also destroyed, but only if no other processes also possess handles to them.
- Any DLLs the process has called receive notification when the process terminates, giving them a chance to clean up and exit.
- The terminating process acquires an exit code. More specifically, the `Exit Status` attribute of the process object changes from `STILL_ACTIVE` to whatever value `ExitProcess` assigns.
- The process object enters its signaled state, and any threads waiting for it to end resume execution.
- Note that when a parent process dies, it does not take its children with it. The children, if any, continue to run independently.

Forcing a Process to Exit

Another command, `TerminateProcess`, also forces a process to exit. Actually, this command brings the process to an abrupt and crashing halt, preventing some of the usual clean up from happening. DLLs, for example, are not notified when `TerminateProcess` kills one of their clients. Like `TerminateThread`, `TerminateProcess` is abrupt, messy, and best avoided whenever possible.

```
BOOL TerminateProcess(  
    HANDLE hProcess,           // handle to the process  
    UINT fuExitCode );        // exit code for the process
```

Pipe-Related Commands

As explained earlier in the chapter, pipes allow two processes to communicate with each other. A pipe is a memory buffer where the system preserves data between the time that one process writes it and the time that another process reads it. The API commands ask you to think of the buffer as a pipe, or conduit, through which information flows from one place to another. A pipe has two ends. A one-way pipe allows writing only at one end and reading only at the other; all the information flows from one process to the other. A two-way pipe allows both processes to read and write, so the information flows both ways at once. When you create a pipe, you also decide whether it will be anonymous or named. Anonymous pipes are simpler, so we'll start with them.

Creating Anonymous Pipes

An anonymous pipe passes information in only one direction, and both ends of the pipe must be on the same machine. The process that creates an anonymous pipe receives two handles: one for reading and one for writing. To communicate with another process, the server must pass one of the handles to the other process.

```
BOOL CreatePipe(  
    PHANDLE phRead,           // read handle variable (inbound)  
    PHANDLE phWrite,          // write handle variable (outbound)  
    LPSECURITY_ATTRIBUTES lpsa, // access privileges  
    DWORD dwPipeSize );       // size of pipe buffer (0=default)
```

The size of the pipe buffer determines how much information the pipe can hold before it overflows. No one can deposit messages in a full pipe until someone makes room by reading the old information from the other end.

If all goes well, `CreatePipe` returns `TRUE` and deposits two new valid handles in the variables indicated by the `PHANDLE` parameters. Next, the creating process usually needs to pass one of the handles to another process. Which handle you give away depends on whether you want the other process to send (write) or receive (read) information through the pipe. You can pass the handle to a child process on its command line or through its standard I/O handles. An unrelated process would need to receive the handle by other means,

such as through a DDE conversation or a shared file. Connections through anonymous pipes are easier to arrange when the processes are related.

Creating Named Pipes

Many Windows NT system objects may be assigned name strings to identify them. While the named pipe operations can be created (compiled) only under Windows NT/2000, applications using named pipes can be run under Windows 95/98, where they perform essentially the same as anonymous pipe objects.

The advantage to using named pipe objects is that names allow other processes to locate objects more easily. Unnamed objects are known only by their handles, and handles are valid only in the process where they originate. In contrast, however, any process that knows the name of an object can ask the system to search its object name hierarchy. Thus, given a name, the system can find any object on any connected machine.

If a pipe has a name, the client program doesn't need to wait for the server to pass it a handle. Instead, the client can acquire a handle by calling CreateFile or CallNamedPipe. In either case, the client needs to know only the pipe's name string.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

This string might be passed by a parent to a child process on the command line, or any process might pass it to any other through a shared file or a DDE conversation. Most often, however, two processes sharing a named pipe have been written by the same developer, so they simply agree on a name string in advance.

The following commands, which are explained in the ensuing sections, work only with named pipes and cannot be used with an anonymous pipe:

- CallNamedPipe
- ConnectNamedPipe
- CreateFile
- CreateNamedPipe
- DisconnectNamedPipe
- GetNamedPipeHandleState
- GetNamedPipeInfo
- ImpersonateNamedPipeClient
- PeekNamedPipe
- RevertToSelf
- SetNamedPipeHandleState
- TransactNamedPipe
- WaitNamedPipe

To create named pipes, use the CreateNamedPipe command:

```
HANDLE CreateNamedPipe(
    LPTSTR lpszPipeName,           // string naming new pipe object
    DWORD fdwOpenMode,             // access, overlap, and write-through
    DWORD fdwPipeMode,             // type, read, and wait modes
    DWORD dwMaxInstances,          // maximum number of instances
    DWORD dwOutBuf,                // outbound buffer size in bytes
    DWORD dwInBuf,                 // inbound buffer size in bytes
    DWORD dwTimeout,               // time-out interval in milliseconds
```

```
LPSECURITY_ATTRIBUTES lpsa ); // access privileges
```

The Parameters

Because named pipes have more features, the `CreateNamedPipe` function takes more parameters.

Naming the Pipe The first parameter points to the string provided to name the new object and is stored by the system in its hierarchical tree of system object names. Pipe name strings should follow this form:

```
\\.\pipe\<pipename>
```

The first backslash designates the root node of the system's object name hierarchy. The other three backslashes separate the names of subsequent nodes. The dot (.) stands for the local machine.

TIP:

Because the C/C++ compiler interprets a single backslash (\) in a string as an escape character, pipe name strings must be written in the source code using the form `\\\\.\\pipe\\<pipename>`.

Although pipes can connect with clients on other network servers, a new pipe object always appears on the local server where it was created. Under the server name is a node called pipe, holding the names of all the pipe objects on the local machine. Within the name string, the substring `<pipename>` is the only section the programmer chooses. This substring may be as long as 256 characters and is not case-sensitive because object names are not sensitive to case.

Servers and clients both use the dot (.) to represent the local server, but a client wishing to open a pipe on a remote server must know the server's name. One way to learn remote server names is to enumerate them with the `WNetOpenEnum`, `WNetEnumResource`, and `WNetCloseEnum` functions, but enumeration is slow.

TIP:

To find out about a better method for enumerating remote server names, see "Distinguishing Pipes from Mailslots" later in this chapter.

The `CreateNamedPipe`, `CreateFile`, `WaitNamedPipe`, and `CallNamedPipe` functions require a pipe's name string as a parameter.

Access, Write-through, and Overlapping The next parameter after the name string, `fdwOpenMode`, combines flags to set several pipe attributes. The most important attribute is the access mode, which determines the direction information flows through the pipe. `fdwOpenMode` must include one of the following three access flags:

PIPE_ACCESS_OUTBOUND The server only writes, and the client only reads.

PIPE_ACCESS_INBOUND The server only reads, and the client only writes.

PIPE_ACCESS_DUPLEX Both the server and client may read and write.

The other two flags in this parameter are optional:

FILE_FLAG_WRITE_THROUGH Disables buffering over a network.

FILE_FLAG_OVERLAPPED Enables asynchronous read and write operations.

The `fdwOpenMode` can also include any combination of security flags as listed following. Different modes (and combinations) can be assigned to different instances of the same pipe and are independent of which other `fdwOpenMode` modes have been specified.

- **WRITE_DAC** Gives the caller write access to the named pipe's discretionary access control list (ACL).
- **WRITE_OWNER** Gives the caller write access to the named pipe's owner.
- **ACCESS_SYSTEM_SECURITY** Gives the caller write access to the named pipe's SACL.

For further information, refer to the on-line documentation for Access-Control Lists (ACLs) and SACL Access Rights.

For efficiency, when a pipe extends to a remote machine, the system normally does not send every message immediately. Instead, it tries to accumulate several short messages in a buffer and send them across the pipe

in a single operation. If too much time passes and the buffer remains only partly full, the system sends the buffer anyway. The `Write_Through` flag prevents the system from buffering; each new message is sent immediately, and write commands do not return until their output has been transmitted.

TIP:

Buffering should be disabled only if messages are expected to be sent infrequently.

Because they involve physical devices, read and write operations are usually slow. The second optional flag, `FILE_FLAG_OVERLAPPED`, allows read and write commands to return immediately while the action they initiate continues in the background.

When an NT program reads from a file, for example, it may choose simply to start the read process, name a procedure to be called when the read operation ends, and then continue executing while the system reads in the background. When the read operation finally ends, the system schedules an asynchronous procedure call and invokes the callback function named in the read command. The callback function then processes the newly retrieved information. Making the system do your reading and writing in the background is called *asynchronous I/O* or *overlapping I/O*. Pipes also support overlapping I/O. Overlapping I/O is more difficult to program because you need to write a callback function, but it's also more efficient.

Type, Read, and Wait The `fdwPipeMode` parameter combines flags to designate another set of pipe features: the read mode, the type, and the wait flag. The type and the read mode are closely related; they might be better named the write mode and the read mode. Together, they control how information in the pipe is organized and interpreted. Both offer a choice between byte and message modes.

These are the pipe-type (write mode) flags:

- `PIPE_TYPE_BYTE`
- `PIPE_TYPE_MESSAGE`

And these are the read-mode flags:

- `PIPE_READMODE_BYTE`
- `PIPE_READMODE_MESSAGE`

Either of two wait mode flags can be combined with the preceding flags, and different instances of the same pipe can be assigned different wait modes. (If no wait mode is specified (0), the parameter defaults to blocking mode.)

- `PIPE_WAIT` Enables blocking mode (default). When the `PIPE_WAIT` mode is used in the `ReadFile`, `WriteFile`, or `ConnectNamedPipe` function, operations are not completed until there is data to read, all data has been written, or a client has been connected—in some situations, this can mean waiting indefinitely for a client process to perform an action.
- `PIPE_NOWAIT` Enables non-blocking mode, and is enabled such that `ReadFile`, `WriteFile`, and `ConnectNamedPipe` always return immediately.

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

TIP:

The non-blocking mode is supported for compatibility with MS LAN Manager v2.0 but should not be used to support asynchronous I/O with named pipes.

The information in a byte-mode pipe is read and written in the normal binary manner and understood as being nothing more than a series of bytes.

Sometimes, however, it is more convenient to divide the information in a pipe into discrete messages, where the output from each separate write command constitutes a new message. A message-mode pipe automatically and transparently prefices each new message with an invisible header specifying the length of the message. The header enables a read command to stop automatically when it reaches the end of one message. The recipient recovers messages one at a time, exactly as they were written.

If one program sends to another a long series of integers, for example, it would probably use a byte-mode pipe, because the receiver doesn't care how many integers were written at one time. Everything it retrieves is simply another integer. But if a program were sending commands written in a script language, the receiver would need to retrieve the commands one at a time, exactly as written, in order to parse them. Further, because each command might be a different length, the two programs would use a message-mode pipe.

The write mode and read mode are designated independently, but not all combinations make sense. Specifically, you can't combine PIPE_TYPE_BYTE and PIPE_READMODE_MESSAGE. A byte-type pipe writes bytes without message headers, so the receiver can't recover message units. On the other hand, you can combine PIPE_TYPE_MESSAGE with PIPE_READMODE_BYTE. In this case, the sender includes message headers but the receiver chooses to ignore them, retrieving the data as a series of undifferentiated bytes. (The receiver still does not see the invisible message headers.)

Besides the flags to set the type and read mode for a pipe, the `fdwPipeMode` parameter accepts one other flag for the wait mode. The wait mode determines what happens when some condition temporarily prevents a pipe command from completing. For example, if you try to read from an empty pipe, some programs might want to forget about reading and move onto the next instruction, but other programs might need to wait for a new message before proceeding.

By default, pipes cause reading threads to block and wait, but you can prevent blocking by adding the

PIPE_NOWAIT flag to `fdwPipeMode`. (The default flag is `PIPE_WAIT`.) The wait mode affects write commands as well as read commands. A program that tries to write when the pipe buffer is full normally blocks until another program makes room by reading from the other end. The wait mode also affects a server trying to connect with a client. If the `ConnectNamedPipe` command finds no ready clients, the wait mode determines whether the command waits for a client to connect or returns immediately.

TIP:

The non-blocking mode is provided primarily for compatibility with LanMan 2.

Pipe Instances A server program may wish to open pipes for more than one client, but probably will not know in advance how many clients it will have. It would be inconvenient to invent a new pipe name for each new client. How would all the clients know in advance what name to use when they open their end of the pipe? To circumvent this problem, Win32 permits the server to create the same pipe over and over.

Each time you call `CreateNamedPipe` with the same name, you get a new instance of the same pipe, with each new instance providing an independent communication channel for another client. Thus, the server might begin by creating the same pipe four times, receiving four different handles, and waiting for a different client to connect to each one. All the clients would use the same pipe name to request their own handles, but each would receive a handle to a different instance. If a fifth client tried to connect, it would block until the server disconnected one of the first four instances (or created a fifth pipe instance).

The `dwMaxInstances` parameter of the `CreateNamedPipe` command sets an upper limit on the number of instances one pipe will support before `CreateNamedPipe` returns an error. The `PIPE_UNLIMITED_INSTANCES` flag indicates no upper limit. In that case, the maximum number of instances is limited only by system resources. The value of `dwMaxInstances` may not exceed the value of `PIPE_UNLIMITED_INSTANCES`.

NOTE:

winbase.h defines `PIPE_UNLIMITED_INSTANCES` as 255.

Buffer Sizes The `dwOutBuf` and `dwInBuf` parameters set the initial size of the buffers that store anything written to the pipe from either end. For an outbound pipe (`PIPE_ACCESS_OUTBOUND`), only the output buffer matters; for an inbound pipe, only the input buffer size is significant.

The limits set by the buffer size parameters are flexible. Internally, every read or write operation on a pipe causes the system to allocate buffer space from the kernel's pool of system memory. The buffer size value is interpreted as a quota limiting these allocations. When the system allocates buffer space for a write operation, it charges the space consumed to the write buffer quota. If the new buffer size fits within the quota, all is well. If it does not, the system allocates the space anyway and charges it to the *process's* resource quota. To avoid excessive charges to the process quota, every `WriteFile` operation that causes the buffer to exceed its quota blocks. The writing thread suspends operation until the receiving thread reduces the buffer by reading from it.

In estimating buffer sizes, you'll need to take into account the fact that each buffer allocation is slightly larger than expected because, in addition to the message contents, it includes an internal data structure of about 28 bytes. The exact size is undocumented and may vary from version to version.

To summarize, the system allocates buffer space dynamically as needed, but threads that frequently exceed their buffer size may block excessively. The sample programs at the end of this chapter leave the buffer size at 0 and suffer no apparent harm. Programs that send frequent messages or that expect the buffers to back up occasionally will benefit from increased buffer sizes.

Time-out Period The `dwTimeout` value matters only when a client calls `WaitNamedPipe` to make a connection, and it matters then only if the client accepts the default time-out period. The default period is the number the server sets in the `dwTimeout` parameter of `CreateNamedPipe`, but the client may set a different period in `WaitNamedPipe`.

Security Attributes The final parameter, a pointer to a `SECURITY_ATTRIBUTES` structure, should look very familiar by now. The values in it determine which operations the new handle allows you to perform on its object, and they also determine whether child processes may inherit the new handle. As usual, if you leave the field `NULL`, the resulting handle has full access privileges and cannot be inherited.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Summary of Named Pipe Attributes

CreateNamedPipe allows you to set many potentially confusing characteristics of the new object. To add to the confusion, some of the characteristics must be exactly the same for every instance of a pipe, but some may vary with each instance. Table 7.4 summarizes these characteristics.

TABLE 7.4: Summary of Named Pipe Attributes

Characteristics	Parameter	Description	Possible Values
Access mode	fdwOpenMode	The direction that information flows	PIPE_ACCESS_OUTBOUND PIPE_ACCESS_INBOUND PIPE_ACCESS_DUPLEX
Type (write mode)	fdwPipeMode	Whether to add a header to each new message	PIPE_TYPE_BYTE PIPE_TYPE_MESSAGE
Wait mode	fdwPipeMode	Whether to block when a command can't work immediately	PIPE_WAIT PIPE_NOWAIT
Overlapped I/O	fdwOpenMode	Whether to permit asynchronous I/O operations	FILE_FLAG_OVERLAPPED
Write-through		Whether to buffer network transmissions	FILE_FLAG_WRITE_THROUGH
Read mode		Whether to end read operations on message boundaries	PIPE_READMODE_BYTE PIPE_READMODE_MESSAGE
Maximum instances	dwMaxInstances	Maximum copies of one pipe allowed simultaneously	Number from 1 to PIPE_UNLIMITED_INSTANCES
Buffer sizes	dwOutBuf & dwInBuf	Size for buffers that hold messages during transit	Size in bytes (0 for defaults)
Time-out period	dwTimeOut	Maximum waiting period for commands to succeed	Period in milliseconds
Security attributes	lpSa	Access privileges and inheritability	SECURITY_ATTRIBUTES

NOTE:

Anonymous pipes always have the characteristics that are the default state for named pipes: PIPE_TYPE_BYTE, PIPE_READMODE_BYTE, PIPE_WAIT, no overlapping I/O, and network buffering enabled.

Return Value

CreateNamedPipe returns a valid handle if it succeeds. If an error occurs, it returns the value (HANDLE)0xFFFFFFFF (otherwise known as INVALID_HANDLE_VALUE).

WARNING:

Applications compiled under Windows 95/98 that use CreateNamedPipe will always return failure (INVALID_HANDLE_VALUE).

Catching Errors

You may have noticed that many of the functions we've described seem to have rudimentary error returns. CreateThread, CreateMutex, CreateProcess, CreatePipe, and CreateNamedPipe, for example, all might fail for a variety of reasons. The system might be low on memory, or a particular mutex might already exist, or a network connection might fail, or a parameter might be invalid. Yet all of these creation functions indicate errors only by returning either FALSE or an invalid handle.

Better diagnostics are available. The system keeps a large set of error messages in a single collective message table and identifies each message with a different number. Whenever a command fails, it stores an error message number for the active thread. Immediately after a function fails, call GetLastError to retrieve the message number. To translate the number into an explanatory string suitable for showing in a message box, call FormatMessage.

Even functions from the Windows 3.1 API sometimes set error codes under Win32. Microsoft's online help file regularly identifies error-setting commands in the descriptions of their return values: "To get extended error information, use the GetLastError function." The demo programs later in this chapter construct a procedure for displaying the appropriate message after any error. Look for ShowErrorMsg in the listings.

Connecting to an Existing Pipe

After a server opens a named pipe, it must wait for a client to open the other end. A client may open its end in any of several ways, but the most common is the CreateFile function. This also works with named pipes, communications devices, and the I/O buffers of a character-based console window. The ReadFile and WriteFile commands also work with the same set of objects. Using a single unified API for several different objects makes programming easier.

```
HANDLE CreateFile(
    LPCTSTR lpzName,           // name of the pipe (or file)
    DWORD fdwAccess,           // read/write access (must match pipe)
    DWORD fdwShareMode,        // usually 0 (no share) for pipes
    LPSECURITY_ATTRIBUTES lpsa, // access privileges
    DWORD fdwCreate,           // must be OPEN_EXISTING for pipes
    DWORD fdwAttrsAndFlags,    // write-through and overlapping modes
    HANDLE hTemplateFile );    // ignored with OPEN_EXISTING
```

The pipe name must match the string the server passed to CreateNamedPipe. If the server and client programs are connecting over a network, the string must name the network server machine instead of using a dot (.).

The fdwAccess parameter tells whether you want to read or write to the pipe. If the pipe was created with the PIPE_ACCESS_OUTBOUND flag, you should specify GENERIC_READ in CreateFile. For an inbound pipe, the client needs GENERIC_WRITE privileges or, for a duplex pipe, both the GENERIC_READ and GENERIC_WRITE privileges.

The fdwShareMode should generally be 0 to prohibit sharing the pipe with other processes. Occasionally,

however, a client might use the share mode to duplicate the pipe handle for another client. In that case, both clients have handles to the same instance of the same pipe, and you might need to worry about synchronizing their read and write operations.

The security attributes in the `lpSa` parameter should be familiar to you by now. The `fdwCreate` flag must be set to `OPEN_EXISTING` because `CreateFile` will not create a new pipe; it simply opens existing pipes. Other flags allow `CreateFile` to create new file objects where none existed before, but those flags produce errors when `lpzName` designates a pipe object.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

The last two parameters normally govern file attributes, such as hidden, read-only, and archive settings, but CreateFile uses the attributes only to create new files. When you open an existing object (with OPEN_EXISTING), the object keeps whatever attributes it already has. However, there are two exceptions. Two flags in the fdwAttrsAndFlags parameters do work when opening an existing named pipe: FILE_FLAG_WRITE_THROUGH and FILE_FLAG_OVERLAPPED. The client may set flags that differ from the server, enabling or disabling network buffering and asynchronous I/O to suit its own preferences.

Modifying an Existing Pipe

Two ends of the same pipe may have different read or wait modes, but CreateFile always copies the original attributes when it opens a handle for a client. Any process, however, can modify its pipe handle with SetNamedPipeHandleState.

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,           // handle of a named pipe
    LPDWORD lpdwModes,           // read and wait mode flags
    LPDWORD lpdwMaxCollect,      // transmission buffer size
    LPDWORD lpdwCollectDataTimeout ); // max time before transmission
```

The first parameter is a handle returned by CreateNamedPipe or CreateFile.

The second parameter, like the fdwPipeMode parameter of CreateNamedPipe, combines flags to set several attributes at once. The lpdwModes parameter controls whether read operations use the byte or message mode and whether certain commands will block while they wait for the pipe to become available. The read mode may be PIPE_READMODE_BYTE or PIPE_READMODE_MESSAGE. (Specifying the message read mode for a pipe created with PIPE_TYPE_BYTE causes an error.) The read-mode pipe may be combined with either PIPE_WAIT or PIPE_NOWAIT.

The last two parameters matter only for pipes that connect with a remote machine. They control how the system buffers network transmissions, but they have no effect on pipes with the PIPE_FLAG_WRITE_THROUGH attribute, which disables network buffering. Buffering allows the system to combine several messages into a single transmission. It holds outgoing messages in a buffer until either the buffer fills or a set time period elapses. lpdwMaxCollect sets the size of the collection buffer, and

lpdwCollectDataTimeout sets the time period in milliseconds.

Getting Information about an Existing Pipe

Three functions retrieve information about a pipe without changing any of its attributes.

Getting State Information

The GetNamedPipeHandleState function is the counterpart of SetNamedPipeHandleState, but retrieves more information than its partner sets:

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,           // handle of named pipe
    LPDWORD lpdwModes,           // read and wait modes
    LPDWORD lpdwCurInstances,    // number of current pipe instances
    LPDWORD lpcbMaxCollect,       // max bytes before remote transmission
    LPDWORD lpdwCollectTimeout,   // max time before remote transmission
    LPTSTR lpszUserName,         // user name of client process
    DWORD dwMaxUserNameBuff );   // size in chars of user name buffer
```

The lpdwModes parameter may contain the PIPE_READMODE_MESSAGE and PIPE_NOWAIT flags. To indicate the byte mode or wait mode, which are the default states, no flags are set.

lpdwCurInstances counts the number of instances that currently exist for a pipe. In other words, it tells how many times the server has called CreateNamedPipe with the same name string.

The collect and time-out parameters retrieve the same network buffering information that SetNamedPipeHandleState controls.

The last two parameters help a server learn about its client. They return the null-terminated string naming the user who is running the client application. Usernames are the names users give to log in. They are associated with particular configuration and security privileges. The server might want the name for a log or a report, but this parameter probably existed for compatibility with OS/2, which also made the username available. The lpszUserName parameter must be NULL if hNamedPipe belongs to a client (in other words, if it was created with CreateFile rather than CreateNamedPipe).

Any of the pointer parameters may be set to NULL to ignore the value normally returned in that place.

Getting Fixed Attributes

The GetNamedPipeInfo function returns information about pipe attributes that may not be changed, while GetNamedPipeHandleState returns attributes that may change during the life of a pipe.

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe,           // handle of named pipe
    LPDWORD lpdwType,            // type and server flags
    LPDWORD lpdwOutBuf,          // size in bytes of pipe's output buffer
    LPDWORD lpdwInBuf,           // size in bytes of pipe's input buffer
    LPDWORD lpdwMaxInstances );  // maximum number of pipe instances
```

The lpdwType parameter may contain either or both of two flags: PIPE_TYPE_MESSAGE and PIPE_SERVER_END. If no flags are set, the handle connects to the client end of a pipe that writes in bytes. The input and output buffer sizes are set in CreateNamedPipe.

The lpdwMaxInstances parameter returns the value set by CreateNamedPipe as an upper limit for the number of simultaneous instances allowed to exist for one pipe.

Retrieving a Message

Normally, when you read from a pipe, the read operation removes from the buffer the message it retrieves. With PeekNamedPipe, however, it is possible to retrieve a message without clearing it from the buffer.

TIP:


```
BOOL PeekNamedPipe(  
    HANDLE hPipe,           // handle of named or anonymous pipe  
    LPVOID lpvBuffer,       // address of buffer to receive data  
    DWORD dwBufferSize,     // size in bytes of data buffer  
    LPDWORD lpdwBytesRead,   // returns number of bytes read  
    LPDWORD lpdwAvailable,   // returns total number of bytes available  
    LPDWORD lpdwMessage );  // returns unread bytes in this message
```

The `lpvBuffer` parameter points to a place where the command can store whatever information it copies from the pipe. Keep in mind that `PeekNamedPipe` cannot retrieve more than `dwBufferSize` bytes, even if more information remains in the pipe.

`lpdwBytesRead` returns the number of bytes the function actually did read, and `lpdwMessage` returns the number of bytes remaining in the current message, if any. `lpdwMessage` is ignored if the pipe's read mode is `PIPE_READMODE_BYTE`. In that case, there are no message units to measure. (All anonymous pipes use the byte read mode.)

The total number of bytes returned in `lpdwAvailable` includes all bytes in all messages. If the buffer currently holds several messages, `*lpdwAvailable` may be greater than the sum of `*lpdwBytesRead` and `*lpdwMessage`.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

It is legal to retrieve only partial information by leaving some parameters set to NULL. If all you want to know is how many bytes are waiting in the buffer, you may, for example, set everything to 0 or NULL except the handle and lpdwAvailable.

When reading from a pipe set to the message read mode, PeekNamedPipe always stops after reading the first message—even if the data buffer has room to hold several messages. Also, PeekNamedPipe never blocks an empty pipe the way ReadFile does if PIPE_WAIT is set. The wait mode has no effect on PeekNamedPipe, which always returns immediately.

Reading and Writing through a Pipe

All the choices you make to create a pipe—named or anonymous, byte or message, blocking or non-blocking—prepare for the moment when you actually send a message through the pipe. One program writes to its handle, and the other program reads from its handle. This most basic transaction typically involves two functions: WriteFile and ReadFile.

```
BOOL WriteFile(  
    HANDLE hFile,                // place to write (pipe or file)  
    CONST VOID *lpBuffer,        // points to data to put in file  
    DWORD dwBytesToWrite,        // number of bytes to write  
    LPDWORD lpdwBytesWritten,    // returns number of bytes written  
    LPOVERLAPPED lpOverlapped ); // needed for asynchronous I/O
```

```
BOOL ReadFile(  
    HANDLE hFile;                // source for reading (pipe or file)  
    LPVOID lpBuffer;             // buffer to hold data retrieved  
    DWORD dwBytesToRead;         // number of bytes to read  
    LPDWORD lpdwBytesRead;       // returns number of bytes read  
    LPOVERLAPPED lpOverlapped ); // needed for asynchronous I/O
```

Bytes to Read or Write

The number of bytes to read or write need not be as large as the size of the buffer, but should not be larger.

If you call `ReadFile` on a message-mode pipe, however, and give `dwBytesToRead` a value smaller than the size of the next message, `ReadFile` reads only part of the message and returns `FALSE`. A subsequent call to `GetLastError` discovers an error code of `ERROR_MORE_DATA`. Call `ReadFile` again (or `PeekNamedPipe`) to read the rest of the message. When `WriteFile` writes to a non-blocking byte-mode pipe and finds the buffer nearly full, it still returns `TRUE`, but the value of `*lpdwBytesWritten` will be less than `dwBytesToWrite`.

Blocking

Depending on the pipe's wait mode, both `WriteFile` and `ReadFile` may block. `WriteFile` might have to wait for a full pipe to empty out from the other end; an empty pipe causes `ReadFile` to block waiting for a new message.

Asynchronous I/O

The final parameter of both commands points to an `OVERLAPPED` structure that provides extra information for performing asynchronous or overlapping I/O. Asynchronous I/O allows the command to return immediately, even before the read or write operation is finished. Asynchronous commands do not automatically modify the position of the file pointer, so the `OVERLAPPED` structure includes an offset pointing to the place in the file where the operation should begin.

The structure also contains a handle to an event object. A thread in the reading program can wait for the event's signal before examining what `ReadFile` placed in the retrieval buffer. You must supply an `OVERLAPPED` structure when using file handles that were created with the `FILE_FLAG_OVERLAPPED` attribute.

Another method of performing asynchronous I/O involves the `ReadFileEx` and `WriteFileEx` commands. Instead of signaling completion with an event, these commands invoke a procedure you provide to be called at the end of each operation.

With respect to pipes, overlapping I/O is a useful strategy for dealing with multiple clients connected to different instances of a single pipe. Synchronous I/O is easier to program, but slow read and write commands might hold up other waiting clients. A server can create a separate thread for each client, as our sample program does for simplicity, but that involves more overhead than the situation actually requires.

A single thread can read and write simultaneously to different pipe instances with asynchronous I/O because each command always returns immediately, leaving the thread free while the system finishes in the background. With `WaitForMultipleObjects`, a thread can arrange to block until any pending operation is completed. The efficiency of asynchronous I/O can make a big difference over slow network connections. Also, it is easier to protect program resources when you have fewer threads to synchronize.

Synchronizing Connections

At any time a client may call `CreateFile` to open its end of a named pipe. However, two problems may arise:

- The server often needs to know when a client has connected to a pipe. Writing to an unconnected pipe serves little purpose, and `CreateFile` does not tell the server when a connection occurs.
- If all the instances of a pipe are busy, `CreateFile` returns immediately reporting `INVALID_HANDLE_VALUE` without establishing a connection. The client may prefer to wait for a pipe instance to become available when another client finishes.

In short, both server and client must be able to block while waiting for conditions that permit a connection to occur. To do so, the server calls `ConnectNamedPipe` and the client calls `WaitNamedPipe`.

```
BOOL ConnectNamedPipe(  
    HANDLE hNamedPipe,           // handle of an available named pipe  
    LPOVERLAPPED lpOverlapped ); // info for asynchronous operation  
  
BOOL WaitNamedPipe(  
    LPTSTR lpszPipeName,         // points to string naming pipe object  
    DWORD dwTimeout );          // maximum wait time in milliseconds
```

NOTE:

These coordinating functions work only with named pipes. For anonymous pipes, a client cannot create its own handle but must receive a handle directly from the server. In that case, the connection is already made.

Signaling a Connection

Just like ReadFile and WriteFile, ConnectNamedPipe can respond asynchronously. The lpOverlapped parameter contains an event handle, and the event object signals when a client connects.

How ConnectNamedPipe behaves depends on whether the pipe was created with, or subsequently modified to include, the FILE_FLAG_OVERLAPPING flag and the PIPE_WAIT mode. Its operation is most intuitive on pipes that allow waiting. Table 7.5 describes the possibilities.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



ITKNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)
[Advanced Search](#)

PUBLICATION LOOKUP

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 7.5: ConnectNamedPipe Behaviors

Wait Mode	Operation Mode	Results
PIPE_WAIT	Synchronous operation	Does not return until a client connects or an error occurs. Returns TRUE if a client connects after the call begins. Returns FALSE if a client is already connected or an error occurs.
PIPE_WAIT	Asynchronous operation	Returns immediately. Always returns FALSE. GetLastError signals ERROR_IO_PENDING if the wait is in progress.
PIPE_NOWAIT	Synchronous operation	Always returns immediately, returning TRUE the first time it is called after a client disconnects. TRUE indicates the pipe is available. Otherwise, the command always returns FALSE. GetLastError indicates ERROR_PIPE_LISTENING if no connection was made or ERROR_PIPE_CONNECTED if a connection has already been made.
PIPE_NOWAIT	Asynchronous mode	Should not be used with a no-wait mode pipe. The no-wait mode exists only for compatibility with LanMan 2.

The unintuitive use of TRUE and FALSE returns results from the fact that ConnectNamedPipe returns TRUE only if the pipe begins in a listening state and a client connects after the connect command begins and before it returns. If the pipe is already connected and the command responds asynchronously (returns without waiting) or is called for a pipe that does not allow waiting, the command generally returns FALSE.

Client Waiting

The client's wait command, WaitNamedPipe, does not actually create a connection. It returns TRUE when a pipe is or becomes available, but it does not return a handle to the available pipe.

It is common for a client to repeat the wait-then-create cycle until it acquires a valid handle. Normally, WaitNamedPipe considers a pipe available only when the server calls ConnectNamedPipe to wait for a link. The two commands work together to synchronize server and client. If, however, the server creates a new pipe and has

never connected to any client, `WaitNamedPipe` returns `TRUE` even without a matching `ConnectNamedPipe`.

The purpose behind the apparent inconsistency is to guarantee that `WaitNamedPipe` connects only at times when the server knows its pipe is available. If a client breaks a connection, the server may not realize it right away; and if another client connected immediately, the server could not know it had a new partner. By recognizing only new pipes and pipes made available through `ConnectNamedPipe`, `WaitNamedPipe` prevents clients from sneaking in on the middle of a running conversation.

NOTE:

`WaitForSingleObject` does not work with pipes because pipes do not have signal states.

Closing a Connection

A client breaks its connection by calling `CloseHandle`. A server may do the same, but sometimes it may prefer to disconnect without destroying the pipe, saving it for later reuse. By calling `DisconnectNamedPipe`, the server forces the conversation to end and invalidates the client's handle.

```
BOOL DisconnectNamedPipe( HANDLE hNamedPipe );
```

If the client tries to read or write with its handle after the server disconnects, the client receives an error result. The client must still call `CloseHandle`.

Any data lingering unread in the pipe is lost when the connection ends. A friendly server can protect the last messages by calling `FlushFileBuffers` first.

```
BOOL FlushFileBuffers( HANDLE hFile );
```

When `FlushFileBuffers` receives a handle to a named pipe, it blocks until the pipe's buffers are empty.

Disconnecting a pipe from its client does not destroy the pipe object. After breaking a connection, the server should call `ConnectNamedPipe` to await a new connection on the freed pipe or else call `CloseHandle` to destroy that instance. Clients blocked on `WaitNamedPipe` do not unblock when a client closes its pipe handle. The server must disconnect its end and call `ConnectNamedPipe` to listen for a new client.

Making Transactions

The `TransactNamedPipe` and `CallNamedPipe` functions facilitate conversations through duplex pipes, combining read and write operations into single transactions. Such combined transactions are particularly efficient over networks because they minimize the number of transmissions.

To support reciprocal transactions, a pipe must be:

- Set up as a named pipe
- Set to use the `PIPE_ACCESS_DUPLEX` flag
- Set as the message type
- Set to message-read mode

The server sets all those attributes with `CreateNamedPipe`. The client can adjust the attributes, if necessary, with `SetNamedPipeHandleState`. The blocking mode has no effect on transaction commands.

Sending a Request

The first command, `TransactNamedPipe`, sends a request and waits for a response. Clients and servers both may use the command, although clients tend to find it more useful.

```
BOOL TransactNamedPipe(
    HANDLE hNamedPipe,           // handle of named pipe
    LPVOID lpvWriteBuf,          // buffer holding information to send
    DWORD dwWriteBufSize,        // size of the write buffer in bytes
    LPVOID lpvReadBuf,           // buffer for information received
    DWORD dwReadBufSize,         // size of the read buffer in bytes
    LPDWORD lpdwBytesRead,        // bytes actually read (value // returned)
    LPOVERLAPPED lpOverlapped ); // info for asynchronous I/O
```

In spite of its many parameters, the function is straightforward. It writes the contents of `lpvWriteBuf` into the pipe, waits for the next response, and copies the message it receives into the `lpvReadBuf` buffer.

The function fails if the pipe has the wrong attributes or if the read buffer is too small to accommodate the entire message. In that case, `GetLastError` returns `ERROR_MORE_DATA`, and you should finish reading with `ReadFile` or `PeekNamedPipe`.

`TransactNamedPipe` handles one exchange through a pipe. After establishing a connection, a program might call `TransactNamedPipe` many times before disconnecting.

Using a Pipe for a Single Transaction

Another command, `CallNamedPipe`, works for clients that need a pipe for only a single transaction. `CallNamedPipe` connects to, reads from, writes to, and closes the pipe handle.

```
BOOL CallNamedPipe(
    LPTSTR lpzPipeName,    // points to string naming a pipe object
    LPVOID lpvWriteBuf,    // buffer holding information to send
    DWORD dwWriteBuf,      // size of the write buffer in bytes
    LPVOID lpvReadBuf,     // buffer for information received
    DWORD dwReadBuf,       // size of the read buffer in bytes
    LPDWORD lpdwRead,      // bytes actually read (value returned)
    DWORD dwTimeout );    // maximum wait time in milliseconds
```

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

For its first parameter, `CallNamedPipe` expects the name of a pipe that already exists. Only clients call this function. Most of the other parameters supply the buffers needed to perform both halves of a transaction.

`CallNamedPipe` condenses into a single call a whole series of commands: `WaitNamedPipe`, `CreateFile`, `WriteFile`, `ReadFile`, and `CloseHandle`. The final parameter, `dwTimeout`, sets the maximum waiting period for the `WaitNamedPipe` part of this transaction.

If the read buffer is too small to hold an entire message, the command reads what it can and returns `FALSE`. `GetLastError` reports `ERROR_MORE_DATA`, but because the pipe has already been closed, the extra data is lost.

Disguising the Server

Often, a client sends commands or requests through a pipe for the server to perform some action on its behalf. The client might, for example, ask the server to retrieve information from a file. Because the client and the server are different processes, they may also have different security clearances. A server might want to refuse to perform commands for which the client lacks adequate clearance. The server may temporarily assume the client's security attributes before complying with a request and then restore its own attributes after responding, using these commands:

```
BOOL ImpersonateNamedPipeClient( HANDLE hNamedPipe );  
BOOL RevertToSelf( void );
```

The impersonation command fails on anonymous pipes, and it will not allow the server to impersonate the client on remote machines. The command temporarily modifies the security context of the thread that calls it.

The `RevertToSelf` command ends the masquerade and restores the original security context.

Destroying a Pipe

Like most of the objects we've discussed so far, a pipe object remains in memory until all the handles to it are closed. Whenever any process finishes with its end of a pipe, it should call `CloseHandle`. If you forget, the `ExitProcess` command closes all your remaining handles for you.

Distinguishing Pipes and Mailslots

Pipes are similar to another object called a “mailslot.” Like a pipe, a mailslot is a buffer where processes leave messages for each other. Mailslots, however, always work in only one direction, and many applications may open the receiving end of the same mailslot.

A program opening a handle to receive messages in a mailslot is a *mailslot server* and a program opening a handle to broadcast messages through a mailslot is the *mailslot client*. When a client writes to a mailslot, copies of the message are posted to every server that has a handle to the same mailslot. Where a pipe takes information from one end and delivers it to the other end, a mailslot takes information from one end and delivers it to many other ends. Mailslots have string names just as named pipes do and the commands for mailslot operations resemble the pipe API functions:

- To create a server’s read-only mailslot handle, call `CreateMailslot`.
- To retrieve or modify its attributes, call `GetMailslotInfo` and `SetMailslotInfo`.
- To create the client’s write-only mailslot handle, call `CreateFile`.
- To send and receive mailslot messages, call `WriteFile` and `ReadFile`.
- To destroy a mailslot, close its handles with `CloseHandle`.

Besides the advantage of broadcasting a message to multiple recipients, mailslots make it easier to connect with processes over a network. For a pipe client to connect with a remote server, the client must first ascertain the name of the server’s machine, which may require a slow enumeration of network servers and repeated attempts to connect with each until the pipe is found. But in naming a mailslot for `CreateFile`, the mailslot client may use an asterisk (*) to represent the current network domain. (A “domain” is any group of linked workstations and network servers to which an administrator has assigned a common name. One network system may contain a single all-inclusive domain, or it may be divided into several associated domains.)

`\\*\mailslot\<mailslotname>`

Using a name in this form, the handle the client receives broadcasts messages to all processes in the current domain that have opened a mailslot using the `<mailslotname>` string. This ability to broadcast across a domain suggests one of the ways mailslots might be used. Processes that want to connect through a remote pipe link may prefer to find each other through a mailslot first, using this procedure:

1. Both processes open a mailslot with the same name.
2. The client broadcasts the name of its pipe, including in the name string the name of its own network server.
3. The recipient uses the broadcast string to connect with the pipe and avoids laboriously enumerating all the available servers.
4. The client and server find each other through a one-way mailslot, and then establish a private two-way pipe to continue the conversation.

Processes Communicating through a Pipe

This chapter presents two versions of a sample program called *AnonPipe* and *NamePipe*. In both versions, a parent process creates a child process and communicates with it through a pipe. The difference is that the first version uses an anonymous pipe, and the second uses a named pipe.

A command on the parent’s menu lets the user launch the child process. Once both are running, the user selects shapes, colors, and sizes from the parent program’s menu. The parent sends the command information through a pipe, and the client draws the requested shape. In the second version, the user may launch several children, and the parent creates multiple instances of its pipe.

One-Way Communication: The *Anonymous Pipe* Version

An anonymous pipe is the easiest way to establish single-instance one-way communication between related processes on the same machine. (With two anonymous pipes you can communicate in both directions.)

The parent and child program windows of the *AnonPipe* program appear in Figure 7.1. In this example, the user has selected a small red rectangle from the parent’s menu, and the resulting shape appears in the child’s window.



FIGURE 7.1 Parent and child windows of the AnonPipe demo program

The Parent Process

The files that comprise the parent process are parent.h, parent.rc, and parent.c. The child process files are child.h, child.rc, and child.c. The two programs together share an additional header, global.h, which defines structures and values both processes use to communicate with each other.

When the parent puts command information into the pipe, it uses a descriptive structure called `FIGURE`. A `FIGURE` variable holds values that represent commands from the parent's menu. The commands determine the shape, size, and color of the figure the child should draw.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Initializing The global variables at the top of *parent.c* include a handle for the child process, a handle for the pipe, and a `FIGURE` variable. These three pieces of information are what the parent needs to communicate with its child. `WinMain` initializes `figure` to describe a small red rectangle.

```

/*-----

PARENT.C [anonymous pipe version]

Contains the parent process for the PROCESS demo program.
In this version, the two processes communicate through
an anonymous pipe.
-----*/

int WINAPI WinMain( HINSTANCE hinstThis, HINSTANCE hinstPrev,
                   LPSTR lpszCmdLine, int iCmdShow )
{
    ...
    // This global FIGURE structure records whatever choices
    // the user makes to choose the shape, size, and color
    // of the figure drawn in the client's window. Here
    // we initialize it to the program's startup defaults.

    figure.iShape = IDM_RECTANGLE; // draw a rectangle
    figure.iSize  = IDM_SMALL;     // don't fill the whole window
    figure.iColor = IDM_RED;       // make the rectangle red
    ...
}

```

Responding to System Messages The window procedure looks for only three messages. When the user begins to make a choice from a menu, the parent program intercepts `WM_INITMENU` to update the appearance of the menu. If the child process does not exist, the parent disables the commands that work only with the child present, including `Terminate` and all the shape options. Two images of the parent program's

Options menu are shown in Figure 7.2.

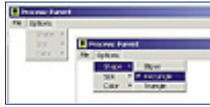


FIGURE 7.2 Two views of the Options menu

In the first instance (upper-left), no child process has been created and, therefore, all of the selections under Options are disabled. In the second instance (lower right), after the child process is active, the menu options are enabled and, on the Shape submenu, the current selection is indicated by a checkmark.

The first message handler, WM_INITMENU, also places checkmarks by all the options currently selected.

```
/*-----
PARENT_WNDPROC
This is where the messages for the main window are processed.
-----*/

LRESULT WINAPI Parent_WndProc(
    HWND    hWnd,           // message address
    UINT     uMessage,       // message type
    WPARAM  wParam,         // message contents
    LPARAM  lParam )        // more contents
{
    switch (uMessage)
    {
        HANDLE_MSG( hWnd, WM_INITMENU, Parent_OnInitMenu );
        HANDLE_MSG( hWnd, WM_COMMAND, Parent_OnCommand );
        HANDLE_MSG( hWnd, WM_DESTROY, Parent_OnDestroy );
        default:
            return( DefWindowProc( hWnd, uMessage, wParam, lParam ) );
    }
    return( 0L );
}
```

The second message handler responds to WM_COMMAND messages from the menu. The user gives commands to start or terminate the child, to modify the shape the child draws, and to close the parent program. If the user makes any selection from the Options menu, SendCommand writes the updated figure variable into the pipe.

The third message handler ends the parent program in response to WM_DESTROY.

The Parent_OnInitMenu procedure provides the handling to check and uncheck several menu options.

```
/*-----
PARENT_ONINITMENU
Check whether the child process exists and enable or
disable the Start, Terminate, and Options commands
accordingly. Also put checkmarks on the option commands
that reflect the user's most recent choices.
-----*/

void Parent_OnInitMenu( HWND hWnd, HMENU hMenu )
{
    // While the child process does not exist, some of our
    // menu commands make no sense and should be disabled.
    // These include the Terminate command and the figure
    // options commands.

    // get a handle to the options popup menu
    HMENU hmenuOptions = GetSubMenu( hMenu, 1 );
```

```

if( hProcess )
{ // child process exists; enable Terminate and shape options
  EnableMenuItem( hMenu, IDM_START, MF_GRAYED );
  EnableMenuItem( hMenu, IDM_TERMINATE, MF_ENABLED );
  EnableMenuItem( hmenuOptions, 0, MF_ENABLED | MF_BYPOSITION );
  EnableMenuItem( hmenuOptions, 1, MF_ENABLED | MF_BYPOSITION );
  EnableMenuItem( hmenuOptions, 2, MF_ENABLED | MF_BYPOSITION );
}
else
{ // child process does not exist;
  // disable Terminate and shape options
  EnableMenuItem( hMenu, IDM_START, MF_ENABLED );
  EnableMenuItem( hMenu, IDM_TERMINATE, MF_GRAYED );
  EnableMenuItem( hmenuOptions, 0, MF_GRAYED | MF_BYPOSITION );
  EnableMenuItem( hmenuOptions, 1, MF_GRAYED | MF_BYPOSITION );
  EnableMenuItem( hmenuOptions, 2, MF_GRAYED | MF_BYPOSITION );
}

// set a checkmark on one of the three shape commands
CheckMenuItem( hMenu, IDM_ELLIPSE,
  ( ( figure.iShape == IDM_ELLIPSE ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );
CheckMenuItem( hMenu, IDM_RECTANGLE,
  ( ( figure.iShape == IDM_RECTANGLE ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );
CheckMenuItem( hMenu, IDM_TRIANGLE,
  ( ( figure.iShape == IDM_TRIANGLE ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );

// set a checkmark on one of the two size commands
CheckMenuItem( hMenu, IDM_SMALL,
  ( ( figure.iSize == IDM_SMALL ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );
CheckMenuItem( hMenu, IDM_LARGE,
  ( ( figure.iSize == IDM_LARGE ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );

// set a checkmark on one of the three color commands
CheckMenuItem( hMenu, IDM_RED,
  ( ( figure.iColor == IDM_RED ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );
CheckMenuItem( hMenu, IDM_GREEN,
  ( ( figure.iColor == IDM_GREEN ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );
CheckMenuItem( hMenu, IDM_BLUE,
  ( ( figure.iColor == IDM_BLUE ) ? (int)MF_CHECKED :
    (int)MF_UNCHECKED ) );

return;
UNREFERENCED_PARAMETER( hwnd );
}

```

Creating the Pipe and the Child When the user chooses Start from the File menu, the program calls its StartProcess procedure to create the pipe, launch the child, and send the child its first command. Some complications arise in arranging for the child to inherit one end of the pipe.

The CreatePipe command must not simply accept the default security attributes because, by default, the new handles cannot be inherited. StartProcess begins by filling out a SECURITY_ATTRIBUTES structure to set the bInheritHandle field to TRUE. If the next command were CreateProcess, the new child would automatically inherit copies of both handles—the reading end and the writing end of the pipe.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



► **KEYWORD SEARCH**

GO!

► [Search Tips](#)

► [Advanced Search](#)

► **PUBLICATION LOOKUP**

GO!

Title

► **JUMP TO TOPIC**



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

Unfortunately, that's still not quite what we want to happen. The child needs only one handle. Anonymous pipes work only one way, and our child needs only to read from the pipe. It should not inherit hPipeWrite. The next command, DuplicateHandle, modifies the handle by changing its inheritance attribute. Because the parameter for the copied handle is NULL, the command does not actually make a copy; instead, it modifies the original. Now we have a reading handle that can be inherited and a writing handle that cannot.

Generally, the child should not inherit handles it does not need. Most objects stay open as long as any process holds an open handle to them. If a child inherits an assortment of extraneous handles, many objects may be forced to linger in memory even after the parent ends. Furthermore, in this particular case, if the child owned handles for both ends of the pipe, it would not know when the parent destroyed its end of the pipe. From the child's point of view, the pipe would remain open because someone (itself) still had a handle to the other end.

When one process closes its end of a pipe, the other process normally notices because of errors resulting from the read or write commands. That's why we went to the trouble of making sure the child inherits only one of the two pipe handles.

Inheriting a handle is not enough, however, for the handle to be useful. The child still needs to receive the handle explicitly

from the parent. In effect, the child inherits only the *right* to use the handle, not the handle itself. More accurately, it receives an entry in its object table but no handle to the entry. The parent must still find a way to pass the handle to the new process. The system considers the inherited object table entry to be an open handle even though the child has no direct access to it. The handle that the parent later passes directly to the child connects with the inherited object table entry and does not count as a separate new handle.

Rather than passing a handle on the command line, we set it in one of the child's standard I/O device channels. The STARTUPINFO procedure passed to CreateProcess contains three standard I/O handles. Two are the default handles returned by GetStdHandle, but the third is the pipe handle. The child will inherit all three devices. (It will use only the pipe handle, but it's best to pass all three handles through STARTUPINFO.)

To review, the StartProcess procedure performs these steps:

1. Loads the string that names the child program's executable file (*child.exe*).
2. Creates the pipe with inheritable handles.
3. Modifies the write-only handle with DuplicateHandle so it cannot be inherited.
4. Puts the read-only handle in a STARTUPINFO variable.
5. Creates the child process, which both inherits the read-only handle and receives a copy of the handle as its own stdin device. (The child inherits no other handles from this parent.)
6. Closes the parent's read-only handle to the pipe. The child has its own copy now, and the parent doesn't need it.

The STARTUPINFO structure allows the parent to decide how and where the child's window will appear. Many of the fields, however, apply only to character-based console windows. This child program uses a graphics window, not a character window. Furthermore, because you have set only one activation flag in the dwFlags field, CreateProcess will ignore most of the values anyway. At a minimum, however, you should initialize the cb field, the lpDesktop field, the dwFlags field, and the three reserved fields.

StartProcess checks for errors after almost every command. If any command fails, StartProcess closes all the handles created up to that point. The ShowErrorMsg procedure, which comes at the end of *parent.c*, displays a message box describing the last error that occurred.

```
/*-----  
    START PROCESS  
    In response to the IDM_START command, launch the child  
    process and create the pipe for talking to it.  
-----*/
```

```
void StartProcess ( void )  
{  
    char szProcess[MAX_BUFFER];          // name of child process image  
    SECURITY_ATTRIBUTES sa;               // security privileges for handles  
    STARTUPINFO sui;                    // info for starting a process  
    PROCESS_INFORMATION pi;              // info returned about a process  
    int iLen;                             // return value
```

```

BOOL bTest;                                // return value
HANDLE hPipeRead;                          // inbound end of pipe (for client)

    // load name of process image file from resources
iLen = LoadString( hInst, IDS_PROCESS, szProcess,
                  sizeof(szProcess) );
if( ! iLen )
{
    return;
}

    // fill out a SECURITY_ATTRIBUTES struct so handles are inherited
sa.nLength = sizeof(SEcurity_ATTRIBUTES); // structure size
sa.lpSecurityDescriptor = NULL;           // default descriptor
sa.bInheritHandle = TRUE;                 // inheritable

    // create the pipe
bTest = CreatePipe( &hPipeRead,           // reading handle
                   &hPipeWrite,          // writing handle
                   &sa,                  // lets handles be inherited
                   0 );                   // default buffer size

if( ! bTest )                             // error during pipe creation
{
    ShowErrorMsg( );
    return;
}

    // make an uninheritable duplicate of the outbound (write) handle
bTest = DuplicateHandle( GetCurrentProcess( ),
                       hPipeWrite,         // original handle
                       GetCurrentProcess( ),
                       NULL,               // don't create new handle
                       0,
                       FALSE,             // not inheritable
                       DUPLICATE_SAME_ACCESS );

if( ! bTest )                             // duplication failed
{
    ShowErrorMsg( );
    CloseHandle( hPipeRead );
    CloseHandle( hPipeWrite );
    return;
}

```

```

    }

    // fill in the process's startup information
    memset( &ampsui, 0, sizeof(STARTUPINFO) );
    sui.cb      = sizeof(STARTUPINFO);
    sui.dwFlags  = STARTF_USESTDHANDLES;
    sui.hStdInput  = hPipeRead;
    sui.hStdOutput = GetStdHandle( STD_OUTPUT_HANDLE );
    sui.hStdError  = GetStdHandle( STD_ERROR_HANDLE );

    // create the drawing process
    bTest = CreateProcess( szProcess, // .EXE image
                          NULL,       // command line
                          NULL,       // process security
                          NULL,       // thread security
                          TRUE,       // inherit handles=yes
                          0,           // creation flags
                          NULL,       // current directory
                          &ampsui,    // startup info
                          &ampi );   // process info (returned)
    block

    // did we succeed in launching the process?
    if( ! bTest )
    {
        ShowErrorMsg( );           // creation failed
        CloseHandle( hPipeWrite );
    }
    else
    {
        // creation succeeded
        hProcess = pi.hProcess;    // save new process handle
        CloseHandle( pi.hThread ); // discard new thread handle
        figure.iShape = IDM_RECTANGLE; // reset to default shape
        SendCommand( );           // tell child what to draw
    }
    CloseHandle( hPipeRead );      // discard receiving end of pipe
    return;
}

```

If the `CreateProcess` command succeeds, `StartProcess` performs several final actions. First, it looks at the two handles returned in the `PROCESS_INFORMATION` structure. It moves the new process's handle to a global variable. The `pi` variable also holds a handle to the primary thread of the new child process. Having no use for that handle, the program closes it immediately. Then it closes the handle to the child's end of the pipe and calls `SendCommand` to tell the child what to draw first.

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Resetting the figure variable before SendCommand is very important. The parent process can open and close a child many times in one session. To close the child, the parent puts IDM_TERMINATE in the figure variable and writes that to the pipe. The line that resets figure.iShape to IDM_RECTANGLE ensures that the child will not receive a leftover IDM_TERMINATE command as its first message.

Writing to the Pipe The parent program sends the child a message immediately after launching the child when the user chooses a new shape attribute and when the user chooses Terminate from the Process menu. At each point, the program calls SendCommand to put information in the pipe.

```

/*-----
SEND COMMAND
    Tell the child program what to draw. Write the current
    contents of the global FIGURE variable into the pipe.

Return
    TRUE indicates the write operation succeeded. FALSE means
    an error occurred and we have lost contact with the child.
-----*/

BOOL SendCommand()
{
    BOOL bTest;                // return value
    DWORD dwWritten;           // number of bytes written to pipe

    // pass the choices to the child through the pipe
    bTest = WriteFile( hPipeWrite,    // anonymous pipe (outbound)
                      &figure,        // buffer to write
                      sizeof(FIGURE), // size of buffer
                      &pdwWritten,    // bytes written
                      NULL );         // overlapping i/o structure

    if( ! bTest )              // did writing succeed?
    {

```

```

        // If the write operation failed because the user has
        // already closed the child program, then tell the user
        // the connection was broken. If some less predictable
        // error caused the failure, call ShowErrorMsg as usual
        // to display the system's error message.

        DWORD dwResult = GetLastError();

        if( ( dwResult == ERROR_BROKEN_PIPE ) || // pipe has ended
            ( dwResult == ERROR_NO_DAT ) )      // close in progress
        {
            // presumably the user closed the child
            MessageBox( hwndMain,
                        "Connection with child already broken.",
                        "Parent Message", MB_ICONEXCLAMATION | MB_OK );
        }
        else // an unpredictable error occurred
            ShowErrorMsg();
    }

    // If a write error occurred, or if we just sent an IDM_TERMINATE
    // command to make the child quit, then in either case we
    // break off communication with the child process.
    if( ( ! bTest ) || ( figure.iShape == IDM_TERMINATE ) )
    {
        CloseHandle( hProcess ); // forget about the child
        hProcess = NULL;
        CloseHandle( hPipeWrite ); // destroy the pipe
    }
    return( bTest );
}

```

The WriteFile command may fail for any of several reasons. One likely problem arises if the user closes the child program from the child's menu. The parent does not know the child is gone until it tries to write to the pipe and receives an error. In that case, the GetLastError function, which returns a number identifying what error last occurred in a given thread, indicates either ERROR_BROKEN_PIPE or ERROR_NO_DATA.

Instead of handling these results like any other error, SendCommand displays a message box explaining that the connection has been broken. If the pipe fails for any reason at all, however, the parent makes no effort to reestablish contact. It closes the pipe handle and the child process handle. The program also resets the process handle to NULL, which CloseHandle does not do. The Parent_OnInitMenu message handler relies on the value of hProcess to determine whether the child still exists.

A separate ShowErrorMsg procedure uses the FormatMessage function to present error messages.

```

/*-----
   SHOW ERROR MESSAGE
-----*/

void ShowErrorMsg ( void )
{
    LPVOID lpvMessage; // temporary message buffer

    // retrieve a message from the system message table
    FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM,
                  NULL, GetLastError(),
                  MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
                  (LPTSTR) &lpvMessage, 0, NULL );
    // display the message in a message box
    MessageBox( hwndMain, lpvMessage, "Parent Message",
                MB_ICONEXCLAMATION | MB_OK );
    // release the buffer FormatMessage allocated
}

```



```
LocalFree( lpvMessage );  
return;  
}
```

ShowErrorMsg is built around the FormatMessage command, which chooses a message from the system's internal message table to describe the most recent error. (The most recent error value is maintained separately for each thread.) Given the flags we've set in its first parameter, FormatMessage dynamically allocates a message buffer and puts the address in the lpvMessage variable. Note that FormatMessage wants to receive the *address* of the buffer pointer, not the pointer itself (the address of an address).

The Child

The CreateProcess command in the parent's StartProcess procedure launches a program called *Child*. The parent stores the string *child.exe* in its table of string resources.

Inheriting a Pipe Handle and Creating a Thread The *AnonPipe* version of *Child* dedicates a secondary thread to the task of waiting for data to arrive through the pipe. The primary thread processes system messages for the program's window. When the secondary thread receives a new command from the parent, it updates the global variables iShape, iSize, and iColor, and then invalidates the window. The primary thread receives a WM_PAINT message and redraws the display using the new shape values. The secondary thread runs in a loop that ends when it reads an IDM_TERMINATE command from the pipe.

The first part of the *Child.c* listing includes WinMain, the initialization procedures, and the message handlers. The child performs its most important initialization tasks in response to the WM_CREATE message.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

Child_OnCreate recovers the pipe handle inherited from the parent and creates a new thread to read from the pipe. If either action fails, the procedure returns FALSE and the process ends.

```
/*-----
CHILD_ONCREATE
On startup open the pipe and start the thread that will
read from it.
-----*/
BOOL Child_OnCreate( HWND hwnd, LPCREATESTRUCT lpCreateStruct )
{
    // open the pipe for reading
    hPipeRead = GetStdHandle( STD_INPUT_HANDLE );
    if( hPipeRead == INVALID_HANDLE_VALUE )
    {
        ShowErrorMsg();
        return( FALSE );
    }
    // Create the thread that will read from the pipe. It is
    // created suspended so its priority can be lowered before
    // it starts.
    hThread = CreateThread( NULL,                // security attributes
                           0,                    // initial stack size
                           (LPTHREAD_START_ROUTINE) PipeThread,
                           NULL,                 // argument
                           CREATE_SUSPENDED,    // creation flag
                           &pdwThreadId );      // new thread's ID

    if( ! hThread )
    {
        ShowErrorMsg();
        return( FALSE );
    }
    // lower the thread's priority and let it run
}
```

```

        SetThreadPriority( hThread, THREAD_PRIORITY_BELOW_NORMAL );
        ResumeThread( hThread );
        return( TRUE );
        UNREFERENCED_PARAMETER(hWnd);
        UNREFERENCED_PARAMETER(lpCreateStruct);
    }

```

To retrieve the pipe handle, the child calls `GetStdHandle`. This command finds the handle that the parent told `CreateProcess` to deposit in the child's stdin device slot. The child creates its secondary pipe-reading thread in a suspended state to adjust the thread's priority. Because the primary thread responds to user input, set the secondary thread to a lower priority. This difference ensures that the child will respond quickly to keyboard and menu input.

The paint procedure (not shown here) is long but straightforward. It simply reads the current values of `iShape`, `iSize`, and `iColor`; creates the pens and brushes it needs; and draws an ellipse, a rectangle, or a triangle.

Reading from the Pipe `Child_OnCreate` designates `PipeThread` as the main procedure for the secondary thread. The new thread immediately enters a while loop that ends when the global `bTerminate` flag becomes `TRUE`. The flag changes when the user chooses `Exit` from the child's menu, when the parent sends an `IDM_TERMINATE` command, or if the child encounters an error reading from the pipe. When the while loop finally does end, the thread posts a `WM_DESTROY` message to the program window. The secondary thread exits, the primary thread receives the destroy command, and the program ends.

```

/*-----
PIPE THREAD
    The WM_CREATE handler starts a thread with this procedure
    to manage the pipe connection. This thread waits for
    messages to arrive through the pipe and acts on them.
-----*/

LONG PipeThread( LPVOID lpThreadData )
{
    while( ! bTerminate )        // read from pipe until
                                // terminate flag = true

        DoRead();
        // when bTerminate is TRUE, time to end program
    FORWARD_WM_DESTROY( hWndMain, PostMessage );
    return( 0L );                // implicit ExitThread()
    UNREFERENCED_PARAMETER(lpThreadData);
}

```

The `DoRead` procedure is responsible for reading from the pipe and for making three decisions. The first is to determine if the read is successful or, if not, to report an error. Second, assuming a successful read, the `IDM_TERMINATE` command is checked to determine if the thread should terminate. And, third, if the read is successful and it is not a terminate message, the shape, color, and size variables are read from the piped message and the child window is updated so a new shape will be drawn.

```

/*-----
DO READ
    Read from the pipe and set the figure to be drawn.
-----*/

void DoRead( void )
{
    FIGURE figure;
    DWORD dwRead;
    BOOL bTest;

    // read from the pipe
    bTest = ReadFile( hPipeRead,          // place to read from
                    &figure,             // buffer to store input

```

```

        sizeof(figure), // bytes to read
        &pdwRead,        // bytes read
        NULL );

if( bTest )
{
    // the read command succeeded
    if( figure.iShape == IDM_TERMINATE )
        // is new command Terminate?
        bTerminate = TRUE;    // set flag to end this thread
    else
    {
        // thread continues; draw new shape
        // copy the new shape attributes to global variables
        iShape = figure.iShape;
        iColor = figure.iColor;
        iSize = figure.iSize;
        // force the parent window to repaint itself
        InvalidateRect( hwndMain, NULL, TRUE );
        UpdateWindow( hwndMain );
    }
}
else
    // the read command failed
{
    ShowErrorMsg( );    // tell user what happened
    bTerminate = TRUE;  // let the child end
}
return;
}

```

The while loop in the secondary thread does one thing: It calls DoRead over and over. The DoRead procedure performs one ReadFile command, interprets the message, and ends. Each call to DoRead retrieves one more message. Because all anonymous pipes use the waiting mode, the thread blocks on each call to ReadFile until data arrives. ReadFile may return immediately if the pipe is full or if an error occurs. For example, it will return immediately if the parent program has already exited and the pipe handle has become invalid.

If the read command succeeds, the child must determine whether it has received a command to terminate or to draw a new shape. If the command fails, the child notifies the user by displaying the system's error message. The child also assumes the connection has been broken and exits.

Although this pipe, like all anonymous pipes, writes and reads in byte mode, the child program still manages to retrieve one message at a time, even if several messages are waiting. Because the parent always writes exactly sizeof(Figure) bytes into the pipe, the child knows where one message ends and the next begins.

The *Child* program's AboutDlgProc and ShowErrorMsg procedures duplicate the corresponding procedures in *Parent* almost exactly. The full listing of *Child* appears on the CD accompanying this book.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Multiple Communications: The *Named Pipe* Version

An anonymous pipe serves the needs of the *Parent* and *Child* programs perfectly well. They communicate in only one direction, use only one instance of the pipe at a time, and both run on the same machine. The second version of the *Process* demo program, however, allows the parent to create any number of children and communicate with several of them at once. Rather than creating a new anonymous pipe for each client, this version creates two instances of a single named pipe.

If the user launches many child processes, only the first two will connect with the parent. The others will block, waiting for one of the existing connections to break. Choosing *Terminate* from the parent's menu causes all the currently connected children to quit. When the link with one child breaks, that instance of the pipe becomes available for one of the waiting children. In Figure 7.3, the parent process has launched three children. Two are connected and have drawn the currently selected shape, while the third is waiting for an available pipe instance.

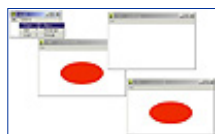


FIGURE 7.3 Parent process with three children, only two of which are connected to instances of the named pipe

WARNING:

Although the *Named Pipe* version of the program works fine under Windows NT/2000, this program will not compile correctly under Windows 95/98.

The listings that follow are partial and present only the most important changes for the new version, but the CD accompanying the book contains full listings for both versions.

The Parent Process

To use a named pipe, both processes must agree on a name string for the pipe they will share. The name

string belongs in the string table of both the parent and the child. We've set the string in a shared resource file, global.str, and modified both resource scripts to include it. The shared header file, global.h, adds a new constant to identify the common string resource. Notice that the name string in global.str contains double the expected number of backslashes:

```
/*-----  
    GLOBAL.STR  
        Contains a string that should be included in the string  
        tables of both the parent and child halves of the  
        PROCESS demo program.  
-----*/
```

```
IDS_PIPE,  "\\\\.\\pipe\\procdemo"    // name of pipe object
```

The resource compiler uses the backslash character to signal the beginning of an ASCII code sequence. Each pair of backslashes inserts a single literal backslash in the resource string.

The new Parent.h header defines the constant NUM_PIPE_INSTANCES, giving it the value of 2. To have the parent create more instances of its pipe and connect with more children simultaneously, modify the definition.

TIP:

In Figure 7.3, there are three child instances where only two have connected—due to the specified NUM_PIPE_INSTANCES limit. When the parent is instructed to close the child processes, the unconnected child will remain open and will assume one of the freed connections.

Creating the Named Pipe The anonymous pipe parent destroys its pipe each time the child process terminates. If the user launches a new child, the parent creates a new pipe. A named pipe, however, often lives through several connections with different clients. The named pipe parent creates its pipe instances only once, during initialization. The parent calls its MakePipeInstance procedure twice. Each successive call produces a handle to the program's pipe object and a new thread to support the new instance.

Parent_OnCreate also produces two other important objects, both events. Each event object broadcasts a signal to all threads that happen to be listening.

```
/*-----  
    PARENT_ONCREATE  
        Create all the pipe instances and the two event objects used  
        to synchronize the program's threads  
-----*/
```

```
BOOL Parent_OnCreate( HWND hwnd, LPCREATESTRUCT lpcs )  
{  
    int i;  
    int iNumInstances = 0;           // counts instances created  
  
    // Create all the instances of the named pipe. The  
    // MakePipeInstance command also starts up a new  
    // thread to service each pipe instance.  
  
    for( i = 1; i <= NUM_PIPE_INSTANCES; i++ )  
        if( MakePipeInstance() ) // make one instance  
            iNumInstances++;      // if successful, increment counter  
  
    if( iNumInstances != NUM_PIPE_INSTANCES )  
    {                               // did we make all of them?  
        char szBuffer[128];  
        wsprintf( szBuffer, "Created only %i instances\n\r",  
                  iNumInstances );  
        MessageBox( hwndMain, szBuffer, "Parent Message",  
                    MB_ICONEXCLAMATION | MB_OK );  
    }
```

```

        return( FALSE );                // creation failed
    }
    // Create the event object used for signaling the pipe
    // instance threads when the user makes a command.
    hCmdEvent = CreateEvent( NULL,      // default security attributes
                            TRUE,      // manual reset event
                            FALSE,     // initially not signaled
                            NULL );    // no name

    if( hCmdEvent == NULL )
    {
        ShowErrorMsg();                // event creation failed
        return( FALSE );
    }

    // Create the event that coordinates the pipe threads when
    // the program terminates all linked children. The threads
    // block on this event until all the clients have received
    // the IDM_TERMINATE message.
    hNotTerminatingEvent = CreateEvent(
                                    NULL, // default security attributes
                                    TRUE,  // manual reset event
                                    TRUE,   // initially signaled
                                    NULL ); // no name

    if( hNotTerminatingEvent == NULL )
    {
        ShowErrorMsg( );                // event creation failed
        return( FALSE );
    }
    return( TRUE );
    UNREFERENCED_PARAMETER(hWnd);
    UNREFERENCED_PARAMETER(lpcs);
}

```

The *Parent* program uses one event to notify its pipe threads whenever the user makes a new choice from the menu. In response, the pipe threads send the new command to their clients. This is a manual reset event, so all listening pipes will unblock when the signal arrives. (Automatic reset events unblock only one thread on each signal.)

The other event coordinates the threads while they are sending termination commands to multiple clients. It, too, is a manual reset event. This one, however, begins life already in its signaled state. You'll see why in the code for the pipe instance threads.

Because no other processes have any reason to use either event object, the events need no names.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

Creating a New Named Pipe Instance To create a new instance of a named pipe, use the MakePipeInstance procedure, which begins by loading the resource string that names the pipe object. The subsequent call to CreateNamedPipe sets the pipe's attributes.

```
/*-----
MAKE PIPE INSTANCE
    Create a new instance of the named pipe.
    Return    TRUE if the procedure creates a new instance;
             FALSE if an error prevents creation.
-----*/
```

```
BOOL MakePipeInstance ( void )
{
    char    szPipe[MAX_BUFFER];        // name of pipe
    int     iLen;                      // return value
    HANDLE hPipe;                     // handle to new pipe
    HANDLE hThread;                   // handle to new thread
    DWORD  dwThreadId;                // ID of new thread

    // get name to use for sharing pipe
    if( ! LoadString( hInst, IDS_PIPE, szPipe, sizeof(szPipe) ) )
        return( FALSE );
    // Create a new instance of the named pipe. This command will
    // fail if two instances already exist.
    hPipe = CreateNamedPipe( szPipe,                // name
                            PIPE_ACCESS_OUTBOUND, // open mode
                            PIPE_TYPE_BYTE | PIPE_READMODE_BYTE |
                            PIPE_WAIT,
                            NUM_PIPE_INSTANCES,    // max instances
                            0,                      // out buffer size
                            0,                      // in buffer size
                            0,                      // time out value
```

```

        NULL );           // security attributes
if (hPipe == INVALID_HANDLE_VALUE)
{
    ShowErrorMsg( );           // creation failed
    return( FALSE );
}
hThread = CreateThread( NULL,           // security attributes
                        0,           // initial stack size
(LPTHREAD_START_ROUTINE) PipeInstanceThread,
                        (LPVOID) hPipe,           // thread proc argument
                        CREATE_SUSPENDED,           // creation flag
                        &pdwThreadID );           // new thread's ID

if( ! hThread )
{
    ShowErrorMsg( );
    CloseHandle( hPipe );
    return( FALSE );           // thread creation failed
}

    // lower the thread's priority and let it run
SetThreadPriority( hThread, THREAD_PRIORITY_BELOW_NORMAL );
ResumeThread( hThread );
    // let go of the handle, for which we have no further use
CloseHandle( hThread );
return( TRUE );
}

```

Because the parent and child send information in only one direction, the program uses a one-way outbound pipe. For clarity, specify all three mode flags even though the pipe's particular characteristics—byte mode and wait mode—are the default values. We don't need the message mode because the parent's messages are always the same length. NUM_PIPE_INSTANCES prevents CreateNamedPipe from producing more than two handles to this pipe.

The first call to CreateNamedPipe sets the pipe's maximum number of instances, and subsequent creation commands will fail if they specify a different number. The zero values for the buffer sizes instruct the system to allocate message space dynamically as needed.

For each pipe handle, MakePipeInstance also creates a new thread. The pipe instance thread waits for the user to choose commands from the parent menu and writes the new command into its pipe. We might equally well have stored the pipe handles in an array and created a single thread to write to all the instances on each new command. Again, as with the anonymous pipe child, the program sets secondary threads to a lower priority, reserving normal priority for only the primary thread—the one thread that responds directly to the user.

This time, we pass a parameter to the thread's starting function. A thread function always receives a 32-bit parameter when it starts up, but until now the programs have not used it. Each instance thread, however, requires a different pipe handle, so hPipe becomes the fourth parameter of CreateThread.

Launching the Child When the user chooses the Start command from the parent's File menu, the program calls StartProcess. In this version, the Start command is always enabled, permitting the user to launch any number of children.

As an exercise in using the command line, the parent passes to each child an identifying number. The first child is 1, the second is 2, and so on. To pass arguments to a child on its command line, ignore the first parameter of CreateProcess and pass the entire command line, including the program name, in the second parameter. The command-line parameter string looks like this:

```
child.exe 1
```

Use the first parameter, lpszImage, when you have no arguments to pass or do not want the system to search for the child's .EXE file along the system PATH.

Because children can acquire their own pipe handles with CreateFile, there is no need to arrange for inheritance. Even if the *Parent* program had created inheritable handles, the children would not inherit any of

them because you pass FALSE as the fifth parameter to CreateProcess.

```
/*-----  
    START PROCESS  
    In response to the IDM_START command, create a new  
    child process. The user may create any number of children.  
-----*/  
  
void StartProcess()  
{  
    STARTUPINFO      sui;      // info for starting a process  
    PROCESS_INFORMATION pi;     // info returned about a process  
    static int iChildNum = 1;   // counts child processes  
    char szProcess[MAX_BUFFER]; // name of child process image  
    char szCmdLine[MAX_BUFFER]; // child's command line  
    BOOL bTest;               // return value  
  
    // load name of process image file from resources  
    if( ! LoadString( hInst, IDS_PROCESS, szProcess,  
                      sizeof(szProcess) ) )  
        return; // loading string failed  
    // fill in the process's startup information  
    sui.cb = sizeof(STARTUPINFO);  
    sui.lpReserved = NULL; // must be NULL  
    sui.lpDesktop = NULL; // starting desktop  
    sui.lpTitle = NULL; // title for new console window  
    sui.dwX = 0; // window starting offsets  
    sui.dwY = 0;  
    sui.dwXSize = 0; // window starting size  
    sui.dwYSize = 0;  
    sui.dwXCountChars = 0; // console screen buffer size  
    sui.dwYCountChars = 0;  
    sui.dwFillAttribute = 0; // console text colors  
    sui.dwFlags = 0; // flags to activate startup fields  
    sui.wShowWindow = 0; // iCmdShow parameter  
    sui.cbReserved2 = 0;  
    sui.lpReserved2 = NULL;  
  
    // prepare child's command-line argument,  
    // and a window caption string  
    wsprintf( szCmdLine, "%s %i", (LPSTR)szProcess, iChildNum++ );  
    // create the drawing process  
    bTest = CreateProcess( NULL, // .EXE image  
                          szCmdLine, // command line  
                          NULL, // process security  
                          NULL, // thread security  
                          FALSE, // inherit handles  
                          0, // creation flags  
                          NULL, // environment block  
                          NULL, // current directory  
                          &sui, // startup info  
                          &ppi ); // process info (returned)  
  
    if( ! bTest )  
    {  
        ShowErrorMsg(); // creation failed  
        return;  
    }  
    WaitForInputIdle( pi.hProcess, 5000 ); // wait for child to start  
    CloseHandle( pi.hProcess ); // we don't need handles  
    CloseHandle( pi.hThread );
```

```
} return;
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

Synchronizing the Threads Whenever the user changes the shape options by choosing a new figure, size, or color, the program must write new commands in all its pipes. The primary thread receives and processes the user's command. It needs a way to make all the pipe instance threads transmit the new information. One of the event objects created during initialization serves this purpose.

When the user picks a new option, the procedures that store the command also pulse the event. The PulseEvent command combines the SetEvent and ResetEvent commands into one operation. The event remains signaled just long enough to unblock all waiting threads and then immediately returns to its unsignaled state.

The ChangeShape procedure incorporates one other modification. When it receives the IDM_TERMINATE command, it saves the old shape value in a global variable, iPrevShape. After all the connected children quit, the parent restores the iPrevShape value to the figure.iShape field. If children are waiting, they will connect immediately to the newly released pipes; if figure.iShape still held IDM_TERMINATE, the children's first command would shut them down. The iPrevShape variable allows newly connected children to draw immediately whatever shape the user last selected.

```
/*-----
CHANGE SHAPE
    Record a shape command from the user. If the user has chosen
    a new shape, send the updated FIGURE structure to the child.
-----*/
```

```
void ChangeShape ( int iCmd )
{
    if( iCmd != figure.iShape )        // new shape?
    {
        // After sending a terminate command, we need to
        // restore the last shape drawn so that newly
        // connected clients can still draw whatever the
        // user last chose.

        if( iCmd == IDM_TERMINATE )
            iPrevShape = figure.iShape; // save old shape command
    }
}
```

```

        figure.iShape = iCmd;           // record new shape command
        PulseEvent( hCmdEvent );       // tell threads shape has
                                        // changed
    }
    return;
}

```

The ChangeSize procedure doesn't have much to do except for recording the new size and then, like the ChangeShape procedure, calling the PulseEvent function to tell the thread that a change has occurred.

```

/*-----
CHANGE SIZE
    Record a size command from the user. If the user has chosen
    a new size, send the updated FIGURE structure to the child.
-----*/

void ChangeSize( int iCmd )
{
    if( iCmd != figure.iSize )    // new size?
    {
        figure.iSize = iCmd;      // record it
        PulseEvent( hCmdEvent );  // tell threads shape has changed
    }
    return;
}

```

Again, the ChangeColor procedure simply records the new color, and then calls the PulseEvent function to tell the thread that a change has occurred.

```

/*-----
CHANGE COLOR
    Record a color command from the user. If the user has chosen
    a new color, send the updated FIGURE structure to the child.
-----*/

void ChangeColor ( int iCmd )
{
    if( iCmd != figure.iColor )    // new color?
    {
        figure.iColor = iCmd;      // record it
        PulseEvent( hCmdEvent );  // tell threads shape has changed
    }
    return;
}

```

Connecting with Clients The threads that run each pipe instance begin life at the PipeInstanceThread procedure. Each new thread enters an endless loop, waiting for clients to connect with its instance of the pipe. When a client does connect, a smaller nested loop begins.

While the connection lasts, the thread waits for command signals from the event object. Each time the event pulses, the thread unblocks, copies the current contents of the global figure variable into its pipe, and resumes its wait for a new command.

If, for any reason, the write operation fails, the thread assumes its client process has terminated. The thread calls DisconnectNamedPipe, returns to the top of its outer loop, and issues ConnectNamedPipe to wait for a new client. The loop also maintains a connection count in the global variable iNumConnections. Each time any thread succeeds in connecting, it increments the counter. When the connection breaks, it decrements the counter. The Parent_OnInitMenu procedure reads the counter to decide which menu options should be enabled. If the parent has no listening clients, all the shape option commands are disabled.

The outer loop of PipeInstanceThread begins with a while (TRUE) command, so the loop can never break. The

pipe threads stop running when the primary thread reaches the end of WinMain and the system calls ExitProcess. At the customary W4 warning level, using a constant for a conditional expression causes the compiler to complain. The #pragma commands surrounding the procedure suppress the warning.

```
// Tell the compiler not to complain about the "while (TRUE)" loop
#pragma warning (disable :4127)
```

Adding to the complexity of this procedure is the task of coordinating the threads when they all disconnect their clients in response to a Terminate command from the user. Several potential problems arise along the way. First, the parent should not write its IDM_TERMINATE command to the pipe and then disconnect immediately because DisconnectNamedPipe destroys any data still lingering in the pipe's buffer. The command could be lost before the child had a chance to read it. When passed a pipe handle, FlushFileBuffers blocks until the receiving program clears the pipe by reading all its contents. Only a program with write access to its pipe can call FlushFileBuffers. The command fails when passed a read-only handle.

As each thread disconnects from its terminated client, it returns to the top of the loop and waits for a new connection. As soon as it connects, it sends the client an initial message to make it draw something right away. But if other threads are still terminating their clients, the global figure variable still contains the IDM_TERMINATE command. The thread will terminate its newly connected client by mistake. We need a way to prevent any thread from sending that initial message to a new client until after all the old clients have been disconnected. The hNotTerminatingEvent object solves the problem.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Near the top of the outer loop you'll find a WaitForSingleObject command that every thread passes before writing its first message to a new client. Most of the time, the event remains signaled and all threads pass quickly by. As soon as one thread sends a termination command, however, it calls ResetEvent to turn off the signal, indicating that a termination sequence has started. Now when any thread finds a new client, it will block before sending the first message. The last thread to terminate its client resets the figure.iShape command to iPrevShape, the value it last held before the termination began. The last thread also calls SetEvent to restore the event signal and unblock the other waiting threads. A thread knows when it is the last to terminate its client because the iNumConnections counter reaches 0.

```
LONG PipeInstanceThread ( HANDLE hPipe )
{
    BOOL bConnected;    // true when a client connects to the pipe

    // This loop runs endlessly. When a client disappears, the
    // loop simply waits for a new client to connect. This thread
    // is terminated automatically when the program's primary
    // thread exits.
    while( TRUE )
    {
        // wait for a connection with some client
        ConnectNamedPipe( hPipe, NULL );
```

If other threads are terminating their clients, then figure.iShape still holds the IDM_TERMINATE command. The thread blocks here until the last client is terminated. The last terminating thread resets figure.iShape to its previous value.

```
        WaitForSingleObject( hNotTerminatingEvent, INFINITE );
        // now the connection is made and a command message is ready
        iNumConnections++;           // update global variable
        SendCommand( hPipe );       // give client its first command
        // send another message each time the Command event signals
```

```

bConnected = TRUE;
while( bConnected )
{
    WaitForSingleObject( hCmdEvent, INFINITE );
    // wait for signal
    if( ! SendCommand( hPipe ) ) // send new shape command
        bConnected = FALSE;
    // The connection failed-probably we just sent
    // IDM_TERMINATE or the user exited from the client.
    // Show no error message.
}
FlushFileBuffers( hPipe ); // wait for child to read message
DisconnectNamedPipe( hPipe ); // break connection
iNumConnections--; // update global variable

```

The following if condition coordinates threads when they are all terminating their clients. When a thread discovers it has just sent the IDM_TERMINATE command, it sets the hNotTerminatingEvent object to the non-signaled state. Other threads will block until the last thread to disconnect restores the signal. The last thread also replaces the IDM_TERMINATE command with IDM_RECTANGLE so all the threads will have a useful command to send in the first message to their new clients.

```

if( figure.iShape == IDM_TERMINATE ) // did we just terminate?
{
    // have all connections
    if( iNumConnections > 0 ) // been terminated?
    {
        // NO; block other threads
        // while terminating proceeds
        ResetEvent( hNotTerminatingEvent );
    }
    else // YES
    {
        figure.iShape = iPrevShape;
        // restore previous command
        SetEvent( hNotTerminatingEvent ); // unblock threads
    }
}
}
return( 0L );
}

```

Last, the conditional expression warning is re-enabled.

```

// allow the "conditional expression constant" warning again
#pragma warning (default :4127)

```

Writing to the Pipe Within the PipeInstanceThread loops, the program repeatedly calls SendCommand to write the current values from figure into the pipe. If the connection with the client breaks, the procedure returns FALSE. The connection may break in either of two ways, and each has different consequences for the program's behavior.

The program disconnects most gracefully when the user chooses Terminate and the parent breaks the links itself. In that case, SendCommand returns FALSE immediately after sending the message, and the program seeks new replacement clients immediately. But if a client closes its handle to the pipe's other end, the server does not know. If the user chooses Exit from the menu of a connected child process, the parent discovers the break only when it later tries to send a new message and WriteFile returns an error value. Furthermore, the broken pipe remains technically connected until the server calls DisconnectNamedPipe. As a consequence, if you end one of the connected child programs while a third child is blocked on WaitNamedPipe, the waiting child remains blocked until you issue another command.

Using a two-way duplex pipe would smooth this transition because the client could send the server a termination message before it exits and the server could disconnect immediately. To accommodate that arrangement, the server would probably dedicate two threads to each pipe instance: one to send and one to

receive, or perhaps one thread to do all the writing for all instances and another to do the reading.

```
/*-----
SEND COMMAND
    Tell the child program what to draw. Write the current
    contents of the global FIGURE variable into the pipe.

Return
    TRUE indicates the write operation succeeded. FALSE means
    an error occurred and we have lost contact with the child.
-----*/

BOOL SendCommand ( HANDLE hPipe )
{
    BOOL bTest;                // return value
    DWORD dwWritten;           // number of bytes written to pipe

    // pass the choices to the child through the pipe
    bTest = WriteFile( hPipe,    // named pipe (outbound)
                      &figure,   // buffer to write
                      sizeof(FIGURE), // size of buffer
                      &dwWritten, // bytes written
                      NULL );     // overlapping i/o structure
    if( ! bTest )                // did writing succeed?
    {
```

If the write operation failed because the user has already closed the child program, then we don't need to do anything special about the error. If, however, some less predictable error caused the failure, call `ShowErrorMsg` as usual to display the system's error message.

```
        DWORD dwResult = GetLastError();

        if( ( dwResult != ERROR_BROKEN_PIPE ) && // pipe has ended
            ( dwResult != ERROR_NO_DATA ) )      // close in progress
        {
            ShowErrorMsg();                      // unpredictable error
        }
    }
```

`SendCommand` returns `FALSE` on errors to indicate that the connection has failed; `SendCommand` also returns `FALSE` after it tells a child to quit because that too makes a connection fail.

```
        return( ( bTest ) && ( figure.iShape != IDM_TERMINATE ) );
    }
```

The Child Process

The child process requires fewer changes. One change is visible in Figure 7.3. Because the parent now creates multiple children, we distinguish each with a different number in its window caption. The parent passes each child its own number through the command-line parameter of `CreateProcess`. The child's window caption also states whether the child is connected to a pipe or waiting for a connection.

```
BOOL CreateMainWindow ( void )
{
    char szAppName[MAX_BUFFER];
    char szBuffer[MAX_BUFFER];
    char *pToken;

    // load the relevant strings
    LoadString( hInst, IDS_APPNAME, szAppName, sizeof(szAppName) );
```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

You use the command-line string from the parent to create the window's caption. The basic caption has the form Child 1 %. The identifying number comes from the parent through the command line. We'll use `wsprintf` to insert the phrase *waiting* or *connected* into the title as appropriate.

```
strcpy( szTitle, "Child " );           // begin with "Child "
strtok( GetCommandLine(), " " );       // move past first word
pToken = strtok( NULL, " " );          // get first argument
if( pToken )                           // is there one?
{
    strcat( szTitle, pToken );          // append it
}
strcat( szTitle, " %s" );               // append a wsprintf format mark
```

During initialization, each child retrieves its assigned number with `GetCommandLine`. This command returns a pointer to a string containing all the command-line arguments separated by spaces. The first item on the command line should be the name of the program (*child.exe*), so calling `strtok` twice extracts the child's sequence number, the second item on the command line. The program constructs a base string of the form "Child 1 %" and calls `wsprintf` to replace the %s marker with a string describing the program's current status, which is initially "waiting" but may change to "connected."

```
// The global szTitle now contains the base caption.
// Insert a current status marker in it.
wsprintf( szBuffer, szTitle, (LPSTR)" (waiting)" );
// create the parent window
hwndMain = CreateWindow( szAppName, szBuffer,
                        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInst, NULL );

// return FALSE for an error
return( hwndMain != NULL );
}
```

Creating a Thread and Connecting to the Pipe~CreateMainWindow has already set the window caption, the new Child_OnCreate procedure no longer loads a caption from the resource string table. Because the child has not inherited a pipe handle, this version also omits the original call to GetStdHandle. Instead, the new thread function, PipeThread, now coordinates all the pipe actions.

PipeThread first loads the pipe name string both programs share in the global.str file, and then it passes the name to CreateFile. The system searches its object name tree and finds the pipe with this name. If an instance of the pipe is available, CreateFile returns a handle to it. Because the original pipe was created with the PIPE_ACCESS_OUTBOUND flag, the client must request GENERIC_READ access rights.

```
LONG PipeThread ( LPVOID lpThreadData )
{
    char    szBuffer[MAX_BUFFER];    // used for pipe name and window
                                        // caption
    BOOL    bConnected;              // TRUE when we have a pipe handle
    HANDLE  hPipeRead;               // receiving handle to 1-way pipe

    // load the string that contains the name of the named pipe
    if( ! LoadString( hInst, IDS_PIPE, szBuffer, sizeof(szBuffer) ) )
        return( FALSE );
    // This while loop continues until an instance of the named
    // pipe becomes available. bConnected is a global variable,
    // and while it is FALSE the primary thread paints "Waiting..."
    // in the window's client area.
    // If an unpredictable error occurs, the loop sets the
    // bTerminate flag, this procedure ends quickly, and it kills
    // the program on its way out.
    bConnected = FALSE;
```

PipeThread embeds the CreateFile command in a while loop that runs until CreateFile returns a valid handle. The loop begins with a WaitNamedPipe command, causing the thread to block waiting for an available instance. WaitNamedPipe does not, however, initiate the connection; CreateFile does that.

```
while( ! bConnected && ! bTerminate )
{
    // wait for a pipe instance to become available
    WaitNamedPipe( szBuffer, NMPWAIT_WAIT_FOREVER );
```

Between the execution of the WaitNamedPipe command and the execution of CreateFile, the system can schedule another thread that grabs the pipe for itself. If that happens, CreateFile will fail even though WaitNamedPipe returned TRUE. If CreateFile fails during PipeThread, the while loop notices the error and tries again. If CreateFile produces an error message indicating any problem other than busy pipes, the loop sets the bTerminate flag, and the process ends a few lines later.

```
    // open the named pipe for reading
    hPipeRead = CreateFile( szBuffer,          // name of pipe
                           GENERIC_READ,      // access mode
                           0,                 // share mode
                           NULL,              // security descriptor
                           OPEN_EXISTING,
                           // don't create new object
                           FILE_ATTRIBUTE_NORMAL,
                           // file attributes
                           NULL );           // file from which
                                              // to copy attributes
    // check that the pipe's handle is valid
    if( hPipeRead == INVALID_HANDLE_VALUE )
    {
        // If CreateFile failed simply because other waiting
        // threads grabbed pipe, don't bother user with error
        // message.
```

```

        if( GetLastError() != ERROR_PIPE_BUSY )
        {
            // an unpredictable error occurred; show message
            ShowErrorMsg();
            bTerminate = TRUE;                // break loop; end program
        }
    }
    else
        bConnected = TRUE;                // succeeded in connecting
}

```

When the client child successfully links to the server parent, the procedure updates the window caption and enters another loop to wait for messages to arrive through the pipe. The receiving loop ends when the user chooses Exit from the child's menu or when the child receives an IDM_TERMINATE command from the parent.

```

        // change window caption to show this window is connected
        wsprintf( szBuffer, szTitle, (LPSTR)" (connected)" );
        SetWindowText( hwndMain, szBuffer );

```

To read messages from the parent, the DoRead procedure calls ReadFile. It needs no revisions to work with a named pipe.

```

        // read messages from the pipe until
        // we receive a terminate command
        while( ! bTerminate )
            DoRead( hPipeRead );
        // when bTerminate is TRUE, end the program
        FORWARD_WM_DESTROY( hwndMain, PostMessage );
        return( 0L );                // implicit ExitThread()
        UNREFERENCED_PARAMETER( lpThreadData );
}

```

Summary

The programs discussed here demonstrate how parent and child processes interact and how pipes allow processes to communicate.

The sample programs in this chapter also introduced the GetLastError and FormatMessage functions for signaling to the user when something goes wrong. In Chapter 9, however, you'll learn about structured exception handling, a more advanced mechanism for dealing with unexpected failures.

First, however, the next chapter will introduce the system registry and registry access operations.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 8

Using the Registry

- The structure of the Windows 2000 registry
- Registry entries and data types
- Registry API functions
- Registry operations

Under Windows 3.x and DOS, applications commonly created initialization files to store configuration parameters, user preferences, status flags, and various other application settings. For example, the WIN.INI file held system configuration data, the REG.DAT file stored file associations and object linking and embedding data, and the SYSTEM.INI file recorded the hardware configuration. Other individual .INI files contained settings for specific applications.

The shortcomings of these numerous initialization files were various. All of these were in ASCII format, easily accessible with any text editor but also generally impossible to understand, alter, or correct. They were also easily damaged, erased, or misplaced. And, in the end, they tended to be a general nuisance. Of course, Windows and application designers did not intend for users to access or edit these initialization files. Ideally, these files would be altered only by the applications themselves and would not be subject to tampering by the uninformed—neither Joe Public nor Jane Engineer. (And ideally, Lil' Abner's schmoo laid both eggs and bottles of grade A milk, tasted better than anything you could imagine, and dropped dead at a hungry glance.)

The reality is a very different animal. Not only are these initialization files untrustworthy and subject to damage, but they are also easily misplaced or erased, with often disastrous consequences. In short, initialization files are a pain.

With the introduction of Windows NT (version 3.1), the system registry appeared as a replacement for a scattered multitude of individual initialization files, including the SYSTEM.INI, WIN.INI, PROTOCOL.INI, CONFIG.SYS, and AUTOEXEC.BAT files.

NOTE:

Even today, a few applications continue to employ .INI files to contain application parameters. Many of these .INI settings are read into the registry on startup, and thus become enshrined in two formats—the registry and the .INI file. Hence, a registry change may simply be undone on reboot unless the offending .INI file is located and also changed.

The reality of the registry is that it is also a pain, just not quite as bad as the initialization files... in some ways. As an application developer, you will need to know how to work with the registry. In this chapter, we will look at methods for creating registry keys, writing data to the registry, and—perhaps most important—retrieving information from the registry.

Welcome to the System Registry

The Windows system registry is a structured database containing all of the settings, paths, variables, and application parameters that were previously parceled among a host of individual initialization files.

The database format offers several advantages. The registry entries are protected from both casual tampering and accidental erasure. Also, because all of the information is in a single place, you can access the information you need without searching through a multitude of sources to find specific data.

The Registry Editor

The registry data is not directly accessible; that is, you cannot view or edit it in the same fashion as you can with an .INI file. To view or edit the registry, you use the Windows 2000 Registry Editor (RegEdt32.exe) utility. The Registry Editor window is shown in Figure 8.1.



FIGURE 8.1 Using the Registry Editor to examine the registry

Two Registry Editors

Windows 2000—like Windows NT—provides two versions of the Registry Editor—RegEdit.exe and RegEdt32.exe. While both offer similar services, for both NT and 2000, the preferred editor is the RegEdt32 utility since this version provides access to registry elements which RegEdit does not view. But on the other hand, RegEdit provides editing capabilities which are not found in RegEdt32.

Briefly, the two registry editors are compared as following:

RegEdit

Search Allows searching through both keys and values in the registry for a given string. RegEdt32 only provides the ability to search for a key, not a value.

Single tree display Displays the registry as a single hierarchical tree, bringing all the keys together in one structure. RegEdit32 displays the registry contents by separating primary (hive) key entries.

RegEdt32

Hives RegEdt32 has the ability to load and unload individual hives.

Read-only mode RegEdt32's Read-only mode protects registry data from potentially damaging, accidental changes. When the Read-only mode is selected, the Registry Editor does not save any changes made.

Default Data Types RegEdt32 supports five (REG_BINARY, REG_DWORD, REG_SZ, REG_EXPAND_SZ, REG_MULTI_SZ) of the six default data types, while RegEdit supports only three. (The new data type REG_FULL_RESOURCE_DESCRIPTOR is not directly supported by RegEdt32.)

SAM and Security Hives RegEdt32 permits direct manipulation of the SAM and Security Hives. This does require RegEdt32 to be launched using AT or equivalent in /Interactive mode in order to gain System level permissions. At this point, the knowledgeable (or foolhardy) gain the ability to edit the contents of these two hives directly. (Of course, ideally, the title bar would change to offer the caution "All hope abandon, ye who enter here...")

Security Menu RegEdt32's Security Menu provides access to view (and edit) permissions and ACLs and to set, audit, and change ownership for registry keys.

In addition, RegEdt32 understands how to display NT/2000-specific key types such as REG_FULL_RESOURCE_DESCRIPTOR, which would be invisible or mangled under RegEdit. Ideally, it would be nice if Microsoft would modify RegEdit to incorporate the features found in RegEdt32 (or modify RegEdt32 to offer RegEdit's features).

As an alternative, you might choose to employ the Norton Registry Editor (which is Windows 2000 compatible), which offers access to permissions but also offers a host of features that are not provided by the RegEdit or RegEdt32 utilities. The Norton Registry Editor is part of both Norton Utilities for Windows 98 and Norton Utilities for Windows NT, and either version automatically detects whether it is running under Windows 9x or under Windows NT/2000 where it adds the security menu.

Although it is easy to make changes using the Registry Editor, caution is **strongly** suggested! Before altering or editing the registry, making a second backup is an excellent idea. From the root of any hive in the Registry Editor, select Registry > Save Subtree As...to write the entire hive to a backup file. Pick a convenient location (such as the root directory) and a recognizable name (such as Current user on Timanog.reg). Note that the saved file will be in ASCII format and may easily occupy 4MB or 5MB of space. Then, if your changes result in any serious problems, you can restore the original registry, either by using the Registry Editor to import the saved version or by simply double-clicking on the file you exported.

More important than editing the registry, from a developer's standpoint, is how your own applications can use the registry to contain important information by creating and writing registry entries and by later retrieving the information from the registry. For this purpose, you must first understand the structure of the registry.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The Registry Structure

The registry is structured as keys and subkeys. For convenience, it's easy to visualize the registry structure as a directory tree where the keys and subkeys are directories and subdirectories (and in the Registry Editor, this is exactly how the registry database is presented). Continuing the parallel, at any level, a key may contain one or more pieces of data—just as a directory may contain one or more files—or it may simply contain links to subkeys.

The Windows 2000 registry structure begins with the five top-level keys listed in Table 8.1. These top-level keys (a.k.a. hives) are predefined system keys, and you can use them for access to subkeys within the registry. For example, the demo presented in this chapter, Reg_Ops, uses the HKEY_CURRENT_USER key for user-specific information and the HKEY_LOCAL_MACHINE key for system-specific data.

TABLE 8.1: Top-Level Registry Keys

Key	Contents
HKEY_CLASSES_ROOT	Information about registered classes, application extensions, and so on. This key is actually a link to the HKEY_LOCAL_MACHINE\SOFTWARE\Classes subkey.
HKEY_CURRENT_USER	Information about the current user's configuration, Desktop layout, preferences, network and dial-up settings, and so on. This key is actually a link to the HKEY_USERS subkey for the current user.
HKEY_LOCAL_MACHINE	Information about the system (hardware) configuration, including global application settings, supported services, device maps, etc.
HKEY_USERS	Information sets for all users registered on the local system (see HKEY_CURRENT_USER).
HKEY_CURRENT_CONFIG	This key is actually a link to the HKEY_LOCAL_MACHINE\ Config\xxxxxx subkey where xxxxxx is the numeric value of the hardware configuration currently used.

NOTE:

For complete and comprehensive details on the Windows registry, we highly recommend Ron Petrusha's *Inside the Windows 95 Registry* (published by O'Reilly and Associates). Although the title refers specifically to Windows 95, the contents are comprehensive and apply to the Windows 98/NT/2000 registries as well. (While dated, this volume is still the definitive guide to the Windows registry.)

Registry entries (named fields) can hold various types of data, as shown in Table 8.2. Of these, the REG_DWORD and REG_SZ formats are most commonly used. However, because of the handling format of the RegQueryValueEx and RegSetValueEx APIs, described in the next section, the data format being used makes little difference in reading or writing to the registry.

TABLE 8.2: Registry Data Types

Value	Meaning
REG_BINARY	Binary data in any format
REG_DWORD	DWORD (32-bit) value
REG_DWORD_LITTLE_ENDIAN	DWORD (32-bit) value using the little-endian format, where the most significant word is in the high-order byte. This is the same as REG_DWORD and is standard for Windows 98/95/NT systems.
REG_DWORD_BIG_ENDIAN	DWORD (32-bit) value using big-endian format, where the most significant value byte appears in the low-order (low-word, low-byte) byte.
REG_EXPAND_SZ	Null-terminated string containing unexpanded references to variables (such as “%PATH%”). May be either Unicode or ANSI, depending on whether Unicode or ANSI functions are being used.
REG_LINK	Unicode symbolic link.
REG_MULTI_SZ	An array of null-terminated (ASCIIZ/UnicodeZ) strings. The array is terminated by two null characters.
REG_NONE	No value type defined.
REG_RESOURCE_LIST	Device-driver resource list.
REG_SZ	Null-terminated (ASCIIZ/UnicodeZ) string. May be either Unicode or ANSI, depending on whether Unicode or ANSI functions are being used.

As an example of the structure of registry keys and entries, take a look at Figure 8.2. This figure illustrates the data used for the Reg_Ops demo, where five subkeys have been created and each subkey has two named fields: Password and Status. The Password field contains a string entry—here, a string validation check for a password, not the actual password. The Status field contains a DWORD value, shown here in hex format.

NOTE:

The (default) field, showing (*value not set*) in Figure 8.2, is a holdover from Windows 3.x, where key fields could contain only one unnamed value and additional subkeys were required to hold further values. While the default field can still be accessed using Windows 3.x functions, this is not recommended. Instead, all registry entries should rely on named fields to hold values.

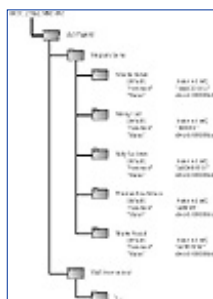


FIGURE 8.2 The structure of registry entries

Registry Access Methods

Windows supplies 40 registry API functions (or 35 if we exclude the five functions provided for Win3.x compatibility), but most applications are likely to use only three or four of these functions. The most commonly required functions appear in bold in the following table. (Additional COM+ registry classes and Java registry functions are also defined but not included here.)

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 8.3: Commonly Used Registry API Functions

Registry API	Use
RegCloseKey	Closes (releases) the handle of a specified key but does not update the registry. (See also RegFlushKey)
RegConnectRegistry	Establishes a connection to a predefined registry handle on another (remote or network) computer. Not required for local registry operations.
RegCreateKey	Creates a specified key or, if the key exists, opens the key. (Win3.x)
RegCreateKeyEx	Creates a specified key or, if the key exists, opens the key. (Win98/95/NT/2000)
RegDeleteKey	Win98/95: Deletes key and all descendents. WinNT/2000: Deletes key but does not delete descendent keys.
RegDeleteValue	Removes a named value from a specified registry key.
RegEnumKey	Enumerates subkeys of a specified opened key. (Win3.x)
RegEnumKeyEx	Enumerates subkeys of a specified opened key. (Win98/95/NT/2000)
RegEnumValue	Enumerates the values contained in an opened key. (Win98/95/NT/2000)
RegFlushKey	Writes all attributes of the specified key to the registry.
RegGetKeySecurity	Retrieves the security descriptor for a specified open key. (WinNT/2000 only)
RegLoadKey	Creates a subkey, under HKEY_USER or HKEY_LOCAL_MACHINE, before copying registry information from a specified file to the subkey.
RegNotifyChangeKeyValue	Notifies caller when the attributes or contents of an open key have changed. Does not notify if a key is deleted.

RegOpenKey	Opens a specified key but does not create the key if it does not exist. (<i>Win3.x</i>)
RegOpenKeyEx	Opens a specified key but does not create the key if it does not exist. (<i>Win98/95/NT/2000</i>)
RegQueryInfoKey	Retrieves subkey information, including size, number, class, security, and so on, about an open key. (<i>Win98/95/NT/2000</i>)
RegQueryMultipleValues	Retrieves type and data for a list of value names associated with an open key.
RegQueryValue	Retrieves a value associated with an unnamed value in a specified open key. (<i>Win3.x</i>)
RegQueryValueEx	Retrieves type and data for a specified value name in an open key. (<i>Win98/95/NT/2000</i>)
RegReplaceKey	Replaces the file backing a key and subkeys with a new file; when the system is restarted, the key and subkeys will assume the values stored in the new file. (<i>Win98/95/NT/2000</i>)
RegRestoreKey	Reads registry information from a specified file, copying it to a specified key and subkeys.
RegSaveKey	Saves a specified key, subkeys, and values to a file.
RegSetKeySecurity	Sets the security of an open key. (<i>WinNT/2000 only</i>)
RegSetValue	Associates an unnamed value (text) with a key. (<i>Win3.x</i>)
RegSetValueEx	Stores data in the value field of an open key, and may set additional value and type information for a key. (<i>Win98/95/NT/2000</i>)
RegUnLoadKey	Unloads (removes) the specified key, subkeys, and values from the registry. (<i>Win98/95/NT/2000</i>)

NOTE:

The `RegSetKeySecurity` and `RegGetKeySecurity` APIs and the security fields in several other functions are relevant only under Windows NT/2000. Windows 95/98 does not support registry security; security settings are simply ignored or returned as null.

In many ways, these API functions seem rather archaic, because they are not MFC-based and do not support or recognize MFC classes. For example, they do not support the `CString` data type, so string information can be supplied or retrieved only as an array of `char`.

The *Regs_Ops* Demo: Registry Operation

The *Reg_Ops* demo, included on the CD accompanying this book, demonstrates how the registry operates. As you can see in Figure 8.3, the dialog-box-based application shows a combo list box and a group of buttons.



FIGURE 8.3 The *Reg_Ops* demo

You may select an existing name from the drop-down list (you will be prompted for a password) or you may enter a new name in the edit box, assign a status, and enter a password for the entry. The name will become a new subkey in the registry with the status and password validation code as named value fields.

TIP:

Be sure to execute the `Special.reg` registry script before using the *Reg_Ops* demo program.

Preparing for the *Reg_Ops* Demo

Before you run the *Reg_Ops* demo, open the *Special.reg* file included on the CD accompanying this book (in the Chapter 8 subdirectory), using Windows Explorer, File Manager, or the file browser of your choice. When you open *Special.reg*, RegEdit will be invoked to write several entries to your system registry under the heading `HKEY_LOCAL_MACHINE\SOFTWARE\`. The following are the plain-text contents of the *Special.reg* file:

REGEDIT4

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo]
[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo\Charlie Brown]
"Password"="@ABCDGCGJ"
"Status"=dword:000003e9

[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo\Henry Ford]
"Password"="B@@EC"
"Status"=dword:000003ea

[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo\Billy Sol Estes]
"Password"="ABDABBCGE"
"Status"=dword:000003eb

[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo\Thomas Alva Edison]
"Password"="AA@@F"
"Status"=dword:000003ec

[HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo\Blaise Pascal]
"Password"="ACBBDEGE"
"Status"=dword:000003ed

[HKEY_LOCAL_MACHINE\SOFTWARE\Not Very Secret]
"Secret Key"="Rumplestiltskin"
```

The *Reg_Ops* demo uses these registry entries to provide preliminary data for illustrating operations.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Opening a Key

The *Reg_Ops* demo begins in *OnInitDialog* when the application opens, by calling *RegCreateKey* to open a handle to the application's private key under the *HKEY_LOCAL_MACHINE* root key.

```
BOOL CReg_OpsDlg::OnInitDialog()  
{  
    CDialog::OnInitDialog();  
  
    ...  
    // begin by opening keys to your registry entries  
    RegCreateKey( HKEY_LOCAL_MACHINE, "SOFTWARE\\Registry Demo",  
                &hRegKey );  
    ResetButtons();  
    // then initialize the combo list box  
    InitializeListBox();  
    return TRUE; // return TRUE unless you set the focus  
                // to a control  
}
```

NOTE:

Calling this a "private" key is something of a misnomer since privacy does not apply to registry keys. All keys are accessible to all applications if the application knows that the key exists or goes to the trouble of finding it. Still, since there is no reason for any other application to want to use this key, it is private in the sense that it has meaning only within the *Reg_Ops* application.

Because this is a list of user names, status flags, and password validation keys, the demo stores this information under *HKEY_LOCAL_MACHINE* rather than under *HKEY_USER*. If you were registering information about a specific user's preferences, you would place this information under *HKEY_CURRENT_USER*, so that that user's settings would be immediately available when he or she logged on to the system. This way, you would not need to query to find out who the user is but simply assume that he or she is the same person who logged on.

The RegCreateKey function opens the key if it already exists or creates the key if it doesn't exist, returning a handle to the key in m_hRegKey. By using a member variable (type HKEY), m_hRegKey is available to other functions in the application. When you expect multiple or frequent accesses to a key, it's convenient to open a handle to the key once and then to use this handle to access subkeys later. In other cases, when you need only occasional access to a key, you can wait and open it if and when needed.

Querying Registry Keys and Values

Next, the demo uses m_hRegKey to initialize the contents of a combo list box by reading entries from subkeys in the registry (see Reg_OpsDlg.cpp).

```

BOOL CReg_OpsDlg::InitializeListBox()
{
    DWORD    dwName, dwSubkeys, dwIndex = 0;
    long      lResult;
    TCHAR     szBuff[MAX_PATH+1];
    CString   csBuff;

    EnableButtons( FALSE );
    lResult = RegQueryInfoKey( m_hRegKey, NULL, NULL, NULL,
                              &dwSubkeys, NULL, NULL, NULL,
                              NULL, NULL, NULL, NULL );

    if( lResult == ERROR_SUCCESS )
        dwIndex = dwSubkeys;
    else ReportError( lResult );
}

```

In this case, before reading the subkeys, we're using the RegQueryInfoKey API to find out how many subkeys HKEY_LOCAL_MACHINE\SOFTWARE\Registry Demo contains. The RegQueryInfoKey contains a number of arguments to return different values, but you may also simply set any undesired arguments to NULL, as done here. This way, we can query a single value and not worry about items that we don't need to know.

The RegQueryInfoKey is defined as:

```

LONG RegQueryInfoKey
(
    HKEY      hKey,                // handle of key to query
    LPTSTR     lpClass,            // class string
    LPDWORD    lpcbClass,          // size of class string
    LPDWORD    lpReserved,         // reserved
    LPDWORD    lpcSubKeys,         // number of subkeys
    LPDWORD    lpcbMaxSubKeyLen,   // longest subkey name
    LPDWORD    lpcbMaxClassLen,    // longest class string
    LPDWORD    lpcValues,          // number of value entries
    LPDWORD    lpcbMaxValueNameLen, // longest value name
    LPDWORD    lpcbMaxValueLen,    // longest value data
    LPDWORD    lpcbSecurityDescriptor, // security descriptor
    PFILETIME  lpftLastWriteTime   // last write time
);

```

The only argument that is absolutely required is hKey, which specifies the registry key we want information

about.

TABLE 8.4: RegQueryInfoKey Arguments

Data Type	Argument	Notes
HKEY	hKey	Handle of key being queried (required)
LPTSTR	lpClass	Class string, description of class associated with key
LPDWORD	lpcbClass	Size of class string buffer, must be supplied with lpClass
LPDWORD	lpReserved	Reserved, always NULL
LPDWORD	lpcSubKeys	Number of subkeys belonging to the reference key
LPDWORD	lpcbMaxSubKeyLen	Longest name length for subkeys
LPDWORD	lpcbMaxClassLen	Longest class string length for classes associated with subkeys
LPDWORD	lpcValues	Number of value entries in reference key
LPDWORD	lpcbMaxValueNameLen	Longest value name length in reference key
LPDWORD	lpcbMaxValueLen	Longest value data length in reference key
LPDWORD	lpcbSecurityDescriptor	Security descriptor length (WinNT only)
PFILETIME	lpftLastWriteTime	Last write time (WinNT only)

Assuming that you have used Special.reg to initialize the registry with the sample data, RegQueryInfoKey should report five subkeys.

Continuing with the InitializeListBox function, now that we know how many subkeys our selected key has, the first step is to set dwName to the size of the szBuff array. Since the size of szBuff is not going to change, we can get away with doing this once and leaving it at that.

```
dwName = sizeof( szBuff );
do
{
    lResult = RegEnumKey( m_hRegKey, -dwIndex,
        (LPTSTR)szBuff, dwName );
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

Since the subkey entries are zero-indexed, before we use `dwIndex`, the value needs to be decremented; that is, for five subentries, we ask for index numbers 4, 3, 2, 1, and 0.

NOTE:

Rather than asking for index numbers, we could have initialized `dwIndex` as zero and then incremented it after each call. We don't really need to know how many subkeys we're going to query because we have other means of discovering when the list of subkeys is exhausted, but we do need an index value for the retrieval.

Using `RegEnumKey`, we are able to execute a cyclic query to discover the names of the subkeys. `RegEnumKey` is defined as:

```
LONG RegEnumKey( HKEY    hKey,        // handle of key to query
                 DWORD    dwIndex,    // index of subkey to query
                 LPTSTR   lpName,     // address of subkey name
                                     // buffer
                 DWORD    cbName );   // size of subkey name buffer
```

Each time we call `RegEnumKey` with a different index value (`dwIndex`), the `lpName` argument returns with the name of another substring. If the substrings are exhausted, `RegEnumKey` returns an error value, `ERROR_NO_MORE_ITEMS`, telling us that we have no more items to find.

The order in which these items are retrieved really doesn't matter. Until we pass an index that does not correspond to a subkey entry, we can expect to return `ERROR_SUCCESS` and to find a new subkey name in the `lpName` array. If we receive any return value other than `ERROR_SUCCESS` or `ERROR_NO_MORE_ITEMS`, then we want to report the error.

```
if( lResult != ERROR_SUCCESS &&
    lResult != ERROR_NO_MORE_ITEMS )ReportError( lResult );
```

We'll cover the `ReportError` procedure, as well as turning system error messages into intelligence, later in "Interpreting System Error Messages." For the moment, we still need to use the subkey name returned by `RegEnumKey`.

```

else
if( lResult == ERROR_SUCCESS )
{
    HKEY    hKeyItem;
    ULONG   ulSize, ulType;
    int     nIndex;

    m_csNameEntry = szBuff;

```

Assuming the result was ERROR_SUCCESS, we want to transfer the string from szBuff to a CString member, m_csNameEntry. We'll reuse szBuff in a moment, but we don't want to lose the information returned a moment earlier.

Next, we call RegCreateKey again. This time, we use the m_hRegKey handle as the root key and the string value in szBuff to identify the subkey to open, and we'll get a new handle, hKeyItem, identifying the subkey.

```

RegCreateKey( m_hRegKey, szBuff, &hKeyItem );

```

Since we already know the key exists—RegEnumKey assured us of that—we simply assume that RegCreateKey has opened the key returning a handle, and now we can use this handle to query the values in the key. Notice, however, that we supply the size of the member receiving the data as part of the request.

```

ulSize = sizeof( m_dwNameStatus );
lResult = RegQueryValueEx( hKeyItem, "Status",
                           NULL, &ulType,
                           (LPBYTE) &m_dwNameStatus, &ulSize );

```

In this call to RegQueryValueEx, we're asking for the value contained in a named field, Status. Along with the value, the data type is also returned in the ulType member as an unsigned long while the size of the data is returned in the ulSize member. Regardless of what kind of data you are requesting—whether it is a DWORD value, a string, a binary array, or any other type—the address of the variable to receive the data is always cast as a pointer to byte. If you make a mistake, such as using a DWORD variable to retrieve a string, you would simply get back the first four bytes of the string, because that is all the space specified.

Next, we again check the lResult returned to be sure that the function performed correctly; if not, we'll report the error.

```

if( lResult != ERROR_SUCCESS ) ReportError( lResult );

```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

At this point, we've retrieved the status member from the subkey and we've retrieved a name from the key itself. Before the loop continues for the next subkey, we need to use these values. We do this by placing them in the ComboBox member as a string entry and as an associated value.

```
nIndex = m_cbNamesList.AddString( m_csNameEntry );
m_cbNamesList.SetItemData( nIndex, m_dwNameStatus );
}
```

Then, until we reach the end of the subkeys, we allow the loop to continue for the next subkey.

```
}
while( lResult != ERROR_NO_MORE_ITEMS );
return TRUE;
}
```

TIP:

Using the same basic techniques, you could be reading much more complex data and even recording the data as structures rather than individual keys. For example, I know of an application that used the registry to record default operating parameters for a dozen different types of motorized microscope stages, along with specific parameters for a each installation plus operator preferences. Altogether, these comprised several hundred settings, some binary and others as string values. This may sound like a lot, but compared to the array of parameters governing how Windows 2000 performs in your custom configuration, these several hundred elements were a very small entry.

The ConfirmPassword subroutine demonstrates reading a registry as a string value. In this subroutine, a name entry retrieved from the combo list box is used to open a handle to a key and then to retrieve the password verification entry.

```
BOOL CReg_OpsDlg::ConfirmPassword()
{
    HKEY    hKeyItem;
    ULONG   ulSize, ulType;
```

```

long    lResult;
TCHAR   szBuff[MAX_PATH+1];
CString csBuff, csPassword;

RegCreateKey( m_hRegKey, m_csNameEntry, &amphKeyItem );
ulSize = sizeof( szBuff );
lResult = RegQueryValueEx( hKeyItem, "Password", NULL, &ampulType,
                          (LPBYTE) szBuff, &ampulSize );
if( lResult != ERROR_SUCCESS )
{
    ReportError( lResult );
    return FALSE;
}

CEnterPassword *pDlg = new CEnterPassword();
if( pDlg->DoModal() == IDOK )
    csPassword = theApp.Encrypt( pDlg->m_csPassword );
else
    return FALSE;
return( csPassword == szBuff );
}

```

Again, the demo includes verification for error results. Then the password returned from the CEnterPassword dialog box is passed to the Encrypt routine, in CReg_OpsApp, before comparing the password entries for verification.

Setting Registry Keys and Values

In some ways, writing a registry key is even simpler than reading one. We begin, as before, by using RegCreateKey to open (create) a key.

NOTE:

If you want to ensure that you are not overwriting an existing key, you can begin by using RegOpenKey as a test. If the key already exists, you would then abort the operation. However, in the demo, we really don't care whether there is an existing key or not—if a key exists, we're prepared to write new information; if not, we'll create it.

After opening or creating the appropriate registry key, we call RegSetValueEx to write the registry information. RegSetValueEx is defined as:

```

LONG RegSetValueEx( HKEY    hKey        // handle of key to set value for
                   LPCTSTR lpValueName, // address of value to set
                   DWORD    Reserved,   // reserved
                   DWORD    dwType,     // flag for value type
                   CONST BYTE * lpData,  // address of value data
                   DWORD    cbData );   // size of value data

```

The RegSetValueEx parameters are used as follows:

hKey Identifies a currently open key or any of the following predefined reserved handle values: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS.

lpValueName Points to a string containing the name of the value to set. If the named value is not already present, it is added to the key. If NULL or a pointer to an empty string and dwType is REG_SZ type, the value is set to the (default) unnamed value (per RegSetValue for Windows 3.x).

Reserved Reserved, must be NULL (zero).

dwType Identifies the information type to be stored. The type identifiers are those listed in Table 8.2.

lpData Points to a buffer containing the data to be stored.

cbData Identifies the size in bytes of the information pointed to by lpData. If the type is REG_SZ, REG_EXPAND_SZ, or REG_MULTI_SZ, cbData must allow for the terminating null character.

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

The size of a value is limited by available memory. To improve performance, values larger than 2KB should be stored as external files (store the filename in the registry). Likewise, elements such as icons, bitmaps, and executables should also be stored as files and not included in the registry.

Also, the opened registry key must be opened using KEY_SET_VALUE access using RegCreateKeyEx or RegOpenKeyEx. (KEY_SET_VALUE is automatic using RegCreateKeyEx.)

The AddToRegistry procedure offers two examples using RegSetValueEx to write registry data:

```
BOOL CReg_OpsDlg::AddToRegistry()
{
    HKEY    hKeyNew;

    RegCreateKey( m_hRegKey, m_csNameEntry, &hKeyNew );
    RegSetValueEx( hKeyNew, "Password", NULL, REG_SZ,
        (LPBYTE)( (LPCTSTR) m_csPassword,
            m_csPassword.GetLength() + 1 ) );
}
```

In the first RegSetValueEx operation, because we are writing a string using the REG_SZ specification, we need to include one extra character in the final size argument to allow for the terminating null character, which the GetLength function does not include.

Earlier, we mentioned that the registry functions do not recognize MFC classes such as CString, but here we used two CString references as arguments. Notice, however, that we have typecast the first use as LPCTSTR, making it appear to be an array of char, and the second instance uses an integer value returned by a member function.

In the second RegSetValueEx operation, we are writing a DWORD value. This operation is just as simple as the first RegSetValueEx, except that we specify REG_DWORD as the data type. Again, the value to be written is passed by address and followed with the size of the data (in bytes).

```
RegSetValueEx( hKeyNew, "Status", NULL, REG_DWORD,
    (LPBYTE)&m_dwNameStatus, sizeof( m_dwNameStatus ) );
return TRUE;
```

```
}
```

Notice also that the two `RegSetValueEx` operations each typecast the data argument as `LPBYTE`, a pointer to byte, and the data must always be passed by address.

And that's about it for writing data to the registry. All in all, it's really very simple.

Interpreting System Error Messages

Earlier, we discussed using the `ReportError` procedure to translate error results returned by the `RegQueryInfoKey`, `RegEnumKey`, and `RegQueryValueEx` functions. In the `ReportError` procedure, which is called with the error value returned by any of the registry functions, the `FormatMessage` function is used with the `FORMAT_MESSAGE_FROM_SYSTEM` flag to use an error value to retrieve an explanatory string from the system.

The `FORMAT_MESSAGE_ALLOCATE_BUFFER` flag allows you to use a void pointer to a buffer without actually allocating the buffer. Instead, the `FormatMessage` function handles the memory allocation according to the size of the message it finds. If you use this flag, you must call the `LocalFree` function to free the memory allocated when you're finished with it.

```
void CReg_OpsDlg::ReportError( long lError )
{
#ifdef _DEBUG
    LPVOID lpMsgBuf;
    FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM,
                  NULL, lError,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  // Default language
                  (LPTSTR) &lpMsgBuf, 0, NULL );
    MessageBox( (char *)lpMsgBuf );
    LocalFree( lpMsgBuf );
#endif
}
```

If you substitute the `GetLastError()` function for the `lError` argument, this routine can be used to report other errors where an error value has not already been explicitly returned.

This error report function is not intended for release applications. The error messages returned are not very useful for presenting to users; they are strictly suitable for programmers during debugging—hence the `#ifdef _DEBUG / #endif` provisions. You should make other provisions for error reports in finished applications.

TIP:

For more on error messages and how to report errors in a manner acceptable to the users, please refer to my book: *Windows Error Messages* (published by O'Reilly and Associates, 1997). This is a shameless plug, granted, but the real shame is in the error messages common in most applications today.

Running the *Reg_Ops* Demo

The real topics of interest in the *Reg_Ops* demo are not in the presentation but inside the code. Still, simply for amusement, each of the status buttons (except for Computer Dweeb) requires a small test. Also, the System Guru button demands that you exhibit a small bit of knowledge about the registry (it really isn't that hard) before you can be accorded this status.

TIP:

To achieve System Guru status, there are a few hints elsewhere in this chapter that should help you, as well as in the registry and on the CD.

The password encryption accepts a string and produces a validation code, which can be used subsequently to compare with a new password entry without actually storing the password. The mechanism used is a very simple one and not particularly foolproof, but it is sufficient for the purpose of this demo. If you are

interested, you can look at the function to see how it works, but keep in mind that there are much better processes for encrypting passwords for real security.

The following are the sample names, supplied by the Special.reg file, and the passwords for each (these are not matched, but you shouldn't find it too hard to figure out which password goes with which name).

Charlie Brown	Countess
Henry Ford	Menlo
Billy Sol Estes	fefifofum
Thomas Alva Edison	edsel
Blaise Pascal	anhydrous

Summary

In some ways, we've probably told you more than it's really safe to know about the registry—for the average user, we would certainly not suggest any access to the registry. Of course, being professionals, you're expected to understand how to handle such potentially dangerous territory. But, in any case, do be careful because even the most casual changes to the registry can have far reaching consequences.

Next, in Chapter 9, we'll look at how to deal with consequences—which hopefully will be unrelated to registry errors—by trapping and handling exceptions during application execution.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 9

Exception Handling

- Exception-handling basics
- Structured exception handling
- Exception-handling functions, classes, and macros

As a program runs, various conditions may disturb its normal flow of execution. The CPU may complain of an improper memory address; the user may interrupt by pressing Ctrl+C; a debugger may halt and resume a program arbitrarily; or an unexpected value may produce an overflow or underflow in a floating-point calculation. Exceptional conditions such as these may arise in user mode or kernel mode, or on a RISC or an Intel chip; either the hardware or the software may signal their occurrence. Furthermore, each programming language must find a way to cope with exceptional conditions. To unify the processing required for all these different situations, Windows NT/2000 (as well as Windows 95/98) builds structured exception-handling mechanisms into the system at a low level. *Exceptions* closely resemble *interrupts*. These are signals that arise within the system and interrupt the processor to handle some new condition. Both signals cause the CPU to transfer control to some other part of the system; however, interrupts and exceptions are not the same. An interrupt occurs asynchronously, often as a result of some hardware event such as a key press or serial port input. A program has no control over such interruptions, and they may occur at any time. An exception, on the other hand, arises synchronously, as a direct result of executing a particular program instruction.

Often, exceptions indicate error conditions, and they can usually be reproduced by running the same code again within the same context. These generalizations are useful guidelines, but in practice the distinction between errors and exceptions is not quite as firm as it might at first appear. The decision to signal some API failures with error returns and others by raising exceptions must sometimes be arbitrary. This chapter explains exception handling and illustrates some examples of using exception-handling routines.

Traps and the Trap Handler

Besides switching from thread to thread, the kernel must respond to interrupts and exceptions. When the

kernel detects an interrupt or an exception, it preempts the current thread and diverts control to a different part of the system—which part depends on what condition the signal indicates. The *trap handler*, the part of the kernel invoked to answer interrupts and exceptions, interprets the signal and transfers control to some procedure previously designated to handle the indicated condition.

Although the system's handling of interrupts will sound familiar to DOS programmers, there are two important differences. First, DOS uses only interrupts, not exceptions. Interrupts are asynchronous, meaning they may occur at any time, and their causes have nothing to do with any code the processor may be executing. Hardware devices, such as a mouse, a keyboard, and a network card, often generate interrupts to feed their input into the processor. Sometimes software generates interrupts, too. The kernel, for example, initiates a context switch by causing an interrupt.

Exceptions, on the other hand, are synchronous, meaning they arise within a sequence of code as the result of executing a particular instruction. Often, exceptions arise when some piece of code encounters an error it cannot handle. Divide-by-zero errors and memory-access violations, for example, cause the system to raise an exception. But not all exceptions are errors. Windows 2000 also raises an exception when it encounters a call for a system service. In handling the exception, the kernel yields control to the part of the system that provides the requested service.

When it receives an interruption signal, the trap handler first records the machine's current state so it can be restored after the signal is processed. Then it determines whether the signal is an interrupt, a service call, or an exception, and passes the signal accordingly to the interrupt dispatcher, the system service dispatcher, or the exception dispatcher. These subsidiary dispatchers locate the appropriate handler routine and transfer control there.

In addition to the trapping of exceptions as well as interrupts, Windows 2000 differs from DOS in assigning priority levels for each interrupt. The priority assigned to an interrupt is called its *interrupt request level (IRQL)*. Do not confuse this with a thread's dynamic priority, which is assigned to a sequence of code; IRQLs are assigned to interrupt sources. The mouse has an IRQL, and its input is processed at one level of priority. The system clock also generates interrupts, and its input is assigned another IRQL.

The CPU also has an IRQL, which changes as the system runs. Changing the CPU's IRQL allows the system to block out interrupts of lower priority. Only kernel-mode services, such as the trap handler, can alter the processor's IRQL. User-mode threads do not have that privilege. Blocked interrupts do not receive attention until some thread explicitly lowers the CPU's level. When the processor runs at the lowest IRQL, normal thread execution proceeds and all interrupts are permitted to occur. When the trap handler calls an *interrupt service routine (ISR)*, it first sets the CPU to that interrupt's IRQL. Traps of a lower level are masked while the ISR runs, in order to prevent relatively unimportant events, such as device-input signals, from interfering with critical operations, such as the power-loss routines. When the processor's IRQL level drops, any interrupts that were masked are drawn from their queue and duly processed. Eventually, the processor returns to the lowest IRQL and the interrupted thread resumes.

To process any interrupt, the trap handler must first locate an appropriate handler routine somewhere in the system. It keeps track of interrupt handlers in the *interrupt dispatch table (IDT)*. The IDT has 32 entries, one for each IRQ level. Each entry points to a handler, or possibly to a chain of handlers if several devices happen to use the same IRQL. When new device drivers are loaded into the system, they immediately add their own handlers to the appropriate IDT entry. They do this by creating and connecting an *interrupt object*; a structure containing all the information the kernel needs to augment the IDT. By using an interrupt object, drivers are able to register their interrupt handlers without knowing anything about the interrupt hardware or the structure of the interrupt dispatch table.

Structured Exception Handling

Any exception that arises must be handled—if not by the program, then by the system itself. Some code somewhere must respond and clear it. Exception handling, then, means providing blocks of code to respond if an exception occurs. A program is likely to have many small handler blocks guarding different portions of code against different exceptions.

In searching for a block of code prepared to deal with a particular condition, the system looks first in the current procedure, then backward through the stack to other active pending procedures in the same process, and finally to the system's own exception handlers. If the offending process happens to be under the scrutiny of a debugger, then the debugging program also gets a chance to handle the exception.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Provisions for Error Handling

The provisions for handling errors differ with the subsystem and the programming language. The WOW subsystem, for example, must handle all exceptions directly because its Win16 clients have no way to do it themselves. Also, different languages may choose to expose exception handling through different syntax. But the name *structured exception handling* implies that the language itself must include some control structure for dealing with exceptions.

C/C++ Exception Handling

C/C++ compilers (per ANSI standards) supply the keywords `try` and `catch`, each introducing a new block of code. A `try` block marks code that might raise exceptions, while a `catch` block contains code to run if an exception occurs. From the programmer's perspective, these syntax structures conveniently separate code that handles exceptional conditions from code that handles normal tasks.

MFC Exception Handling

MFC uses an exception-handling scheme that is modeled on—but not identical to—the system proposed by the ANSI standards committee for C++. Following either the ANSI-proposed standard or the MFC version requires an exception handler to be initiated before calling a function that may encounter an abnormal situation. If the function encounters an abnormal condition, an exception is thrown and control is passed to the exception handler.

Visual C++ Exception Handling

In Visual C++, exception handling is supported by several mechanisms:

- Exception-handling functions, which structure exception handling
- Exception-handling classes, which provide responses to specific exception types
- Exception macros, which structure application exception handlers
- Exception-throwing functions, which generate exceptions of specific types

All of these mechanisms are designed to throw exceptions or specialized exceptions and to terminate

programs if necessary.

The basic exception-handling statements in Visual C++ are try and catch. The try/catch structure separates the code for exceptional situations from the code for normal situations.

```
try                                // beginning of try block
{
    <guarded code statements>      // code that may produce exceptions
}
catch( <filter> )                  // beginning of exception handling block
{
    <exception handler>           // code to execute if an exception occurs
}
```

DOS Exception Handling

Under DOS, the try keyword becomes __try, catch becomes __except and a third keyword, __finally is also supported. The __finally block contains code to run when a __try block ends, even if the __try block fails or is interrupted.

For those interested in the DOS exception handling, several DOS-based examples are included on the CD accompanying this book. Look in the Chapter 9 directory for the Errors, Jumps, Nest, Threads and Unwind subdirectories, which contain DOS examples (originally published in *Mastering Windows NT Programming*, by Brian Myers and Eric Hamer). These examples have been revised for compatibility with Visual C/C++, and they will appear in a DOS window when compiled and executed. In addition, some notes have been added to explain current performance.

For more information on DOS exception handling, see the *NT4/Windows 95 Developer's Handbook*, by Ben Ezzell (published by Sybex).

Filtering Exceptions

In any implementation, structured exception handling associates a block of code for handling exceptions with a block of code it is said to guard. If an exception occurs while the guarded block executes, control transfers to the filter expression. Usually, the filter expression asks what the exception is and decides how to proceed. Exceptions that pass through the filter reach the exception-handling code. If the filter blocks an exception, then the handler is not invoked. The system continues to search elsewhere for a handler that will take the exception.

The filter expression may be complex. It may even call a separate function. Sometimes the filter does the real work of responding to an exception, leaving the catch block empty. In many cases, CException and CException-derived MFC classes may be used to handle exception events.

What Is Exceptional?

Programmers new to structured exception handling sometimes have the false impression that they no longer need to check for error returns after executing each command. An error, however, is not the same thing as an exception. A function can fail without raising an exception. For example, consider these lines of code:

```
hBrush = CreateSolidBrush( RGB(255, 0, 0) );
hOldBrush = SelectObject( hDC, hBrush );
Rectangle( hDC, 0, 0, 100, 100 );
```

If the first command fails and returns NULL for the brush, then SelectObject fails, too. The third command still draws a rectangle but does not color it correctly. No exceptions are raised. The only way to protect against those failures is to check the return values. Here's another example:

```
HANDLE hMemory;
char *pData;

hMemory = GlobalAlloc( GHND, 1000 );
pData = (char *)GlobalLock( hMemory );
```

If the allocation fails, then `hMemory` becomes `NULL`, `GlobalLock` fails, and `pData` becomes `NULL`, too. Neither failure, however, produces an exception. But the next line does produce an exception when it tries to write to an invalid address:

```
pData[0] = 'a';           // raises exception if pData = NULL
```

In the *Exceptions* demo, to create an exception, a similar process is used:

```
void CExceptionsView::ForceException()
{
    int    *p = 0x00000000;    // void pointer

    *p = 999;                  // invalid memory access
}
```

Here, assigning a void pointer is perfectly valid, but attempting to assign a value to the void pointer is not. This latter action is guaranteed to produce an exception, which is exactly what we want to do to test the exception-handler mechanisms.

TIP:

Using a memory-access violation to generate an exception is something of an extreme case; none of the `CException` classes provide explicit handling for this type of error. But, as you will see later in this chapter when we discuss the *Exception* demo routines, there is still a way to handle it!

An exception is a kind of error that a command cannot process. If `GlobalAlloc` cannot find enough room, it simply returns `NULL`. But if the assignment operator has no valid destination to place a value, it can do nothing, not even return an error. It must raise an exception; and if the process cannot handle the exception, the system must close down the process.

The line between exceptions and errors is sometimes difficult to draw. The difference between an error and an exception is sometimes a matter of implementation. Recognizing commands that might raise exceptions takes a little practice. You should learn which exceptions can arise and then imagine which operations might cause them. For example, a faulty assignment statement causes an *access-violation* exception. The list of possible exceptions varies on different machines, but here are some exceptions that the Windows NT/2000 kernel defines:

- Data-type misalignment
- Debugger breakpoint
- Debugger single-step
- Floating-point divide by zero
- Floating-point overflow and underflow
- Floating-point reserved operand
- Guard-page violation
- Illegal instruction
- Integer divide by zero
- Integer overflow
- Memory-access violation
- Page-read error
- Paging file quota exceeded
- Privileged instruction

Frame-Based Exception Handling

The exception-handling mechanisms in Visual C++ are *frame-based*, meaning that each catch block is associated with, or framed in, the procedure that contains it. The term *frame* describes a layer in the program stack. Each time a program calls a procedure, the program pushes a new set of information on the stack. The information includes, for example, parameters passed to the new procedure and an address showing where to return when the called procedure ends. If the second procedure calls a third and the third a fourth, each

successive call pushes a new frame onto the stack, as you see in Figure 9.1.

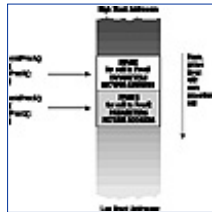


FIGURE 9.1 Frames in a program stack

Each frame represents an activated procedure waiting for its subroutines to finish before resuming. At any point, it is possible to trace back through the stack frames to discover which procedures have been called. When an exception occurs, the system traces back, looking for exception handlers in each pending procedure.

NOTE:

The internal mechanisms that support exception handling vary from system to system. A MIPS machine, for example, implements handlers through tables, not stacks.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

Order of Execution

When an exception arises, the system first saves the machine context of the interrupted thread, just as it does when it performs a context switch for normal multitasking. Depending on how the exception is eventually handled, the system may later use the saved context to resume execution at the beginning of the line where the exception occurred.

In response to an exception, the system generally tries to execute the exception filter first, and then the exception handler. However, you can provide for a variety of exceptions by stacking catch blocks, as illustrated by the following fragment:

```
try
{
    ...           // something that might cause an exception
}
catch( CMemoryException *e )      // out of memory exception
{
    // yadda, yadda, yadda
}
catch( CFileException *e )        // file exception
{
    // yadda, yadda, yadda
}
catch( CArchiveException *e )     // Archive/Serialization
                                  // exception
{
    // yadda, yadda, yadda
}
catch( CNotSupportedException *e ) // response to request for
                                  // unsupported service
{
    // yadda, yadda, yadda
}
catch( CResourceException *e )    // resource allocation
```



```

{                                     // exception
    // yadda, yadda, yadda
}
catch( CDaoException *e )           // database exceptions
{                                   // (DAO classes)
    // yadda, yadda, yadda
}
catch( CDBException *e )           // database exceptions
{                                   // (ODBC classes)
    // yadda, yadda, yadda
}
catch( COleException *e )          // OLE exceptions
{
    // yadda, yadda, yadda
}
catch( COleDispatchException *e ) // dispatch (automation)
{                                   // exceptions
    // yadda, yadda, yadda
}
catch( CUserException *e )         // user exception
{
    // yadda, yadda, yadda
}
catch( CException *e )             // must follow all derived
{                                   // classes or compiler
    // yadda, yadda, yadda         // will flag error
}
catch(...)                         // catch everything!!!
{
    // yadda, yadda, yadda
}

```

Here, all of the CException-derived classes are included in catch statements. An example that follows this structure is included in the *Exceptions* demo. However, in practice, you will probably not need to use such a thorough exception-handler routine. There are two items to note here:

- The catch(CException *e) block must follow all of the catch blocks using derived classes. If this sequence is not followed, the compiler will post an error warning that the derived blocks following catch(CException *e) will not be recognized because the CException block will have already trapped the exception.
- The catch(...) block follows everything else because this block catches exceptions that are not handled by the CException and CException-derived blocks (more on each of these in a moment).

Exception Handling and Debuggers

The task of finding a handler becomes more involved when a process is being debugged, because the system transfers control to the debugger *before* seeking the program's own exception handlers. Typically, the debugger uses this early alert to handle single-step and breakpoint exceptions, allowing the user to inspect the context before resuming.

If the debugger decides *not* to handle an exception during the early alert, the system returns to the process in search of a handler. If the process also refuses the exception, the system gives the debugger a second chance to handle what has now become a more serious situation. If the debugger refuses a second time, then the system finally gives up and settles for providing its own default response, which is usually to terminate the process.

Exception-Handling Macros

As an alternative to using the try/catch functions directly, you can structure your code with exception-handling macros. When using the macro approach, you'll begin with the TRY macro, which sets up a TRY block identifying a block of code that might throw exceptions.

Exceptions raised in the TRY block are handled in the following CATCH and AND_CATCH blocks. Also,

recursion is allowed; exceptions may be passed to an outer TRY block, either by ignoring them or by using the THROW_LAST macro.

All CATCH blocks end with an END_CATCH or END_CATCH_ALL macro. If no CATCH block is supplied, the TRY block must include the END_CATCH or END_CATCH_ALL macro.

The THROW and THROW_LAST macros throw the specified exception, interrupting program execution and passing control to the program's associated CATCH block. If a CATCH block is not provided, control is passed to an MFC library module, which displays an error message and then exits. The THROW_LAST macro throws the exception back to the next outer CATCH block.

Nested exception handling is discussed in the following section.

Exception Classes

MFC provides a group of exception-handler classes, based on the CException class, for handling specific types of exceptions. Table 9.1 lists the derived classes.

TABLE 9.1: CException-derived Classes

Class	Description
CMemoryException	Out-of-memory exception
CNotSupportedException	Request for an unsupported operation
CArchiveException	Archive-specific exception
CFileException	File-specific exception
CResourceException	Windows resource not found or cannot be created
COleException	OLE exception
CDBException	Database exception, such as exception conditions arising for MFC database classes based on Open Database Connectivity (ODBC)
COleDispatchException	OLE dispatch (automation) exception
CUserException	Exception that indicates that a resource could not be found
CDaoException	Data access object exception, such as exception conditions arising for DAO classes
CInternetException	Internet exception, such as exception conditions arising for Internet classes

These exceptions are intended to be used with the try/catch block functions or the TRY, CATCH, AND_CATCH, THROW, THROW_LAST, and END_CATCH macros.

Customarily, you use a derived class to catch a specific exception type, but you may also use CException to trap all exception types before calling CObject::IsKindOf to differentiate between the derived classes. Remember, however, that CObject::IsKindOf applies only to classes declared using the IMPLEMENT_DYNAMIC macro, taking advantage of dynamic type checking. Any CException-derived class that you create should use the IMPLEMENT_DYNAMIC macro, too.

You can use the GetErrorMessage and ReportError member functions with any of the CException-derived classes to report details about exceptions.

If an exception is caught by one of the macros, the CException object is deleted automatically; do not delete it yourself. If an exception is caught by using a catch keyword, it is not automatically deleted and you do need to delete it yourself.

NOTE:

For more information on exception macros, refer to Exception Processing or the “Exceptions” article in the online *Visual C++ Programmer's Guide*.

Because CException is an abstract base class, you cannot create CException-derived classes; you must create custom classes from derived classes. If you need to create your own CException type, use one of the derived classes listed as a model. Also, ensure that the derived class uses the IMPLEMENT_DYNAMIC macro to support dynamic type checking.

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Catching Everything

In addition to using explicit CException-derived classes for catch clauses, you can also supply an ellipsis as the argument, like this : catch (...). When you use an ellipsis as the exception declaration statement, the catch clause handles all types of exceptions, including C exceptions as well as system-generated and application-generated exceptions. These include exceptions such as memory protection, divide-by-zero, and floating-point violations.

TIP:

Remember, just as the CException handler must follow any CException-derived handler, an ellipsis catch handler must always be the last handler for its try block.

Plug-and-Play Event Messages

While Plug-and-Play messages are not precisely the same as exception messages, they do serve a similar purpose—warning an application that something has occurred that could potentially affect operations. Note particularly the word ‘potentially’ because here the report is not that an error event has occurred, but simply that a normal and permitted event is taking place and the notification is provided as a courtesy.

More specifically, a plug-and-play event message is a hardware-level event issued as notification that some element of the system hardware is being added or removed. Whether this occurrence may affect operations or not is up to the application to determine and to provide handling in response.

At present, Plug-and-Play events are most relevant to laptop machines, with or without docking stations. However, as ‘portable’ systems supplant conventional desktop systems and with the introduction of the Universal Serial Bus allowing system components to be removed and added without having to ‘power down and open up’ the main computer, Plug-and-Play events will become increasingly important to every application.

NOTE:

Just as Windows 98’s Plug-and-Play handling was an improvement over Windows 95’s, Windows 2000 has again improved support for PnP devices.

How Plug-and-Play Messages Function

In brief, the current Plug-and-Play standards under Windows make provisions for generating WM_COMMAND/WM_DEVICECHANGE messages to report event codes describing the type of change and, in general, the device whose availability is changing. Plug-and-Play messages can provide notification of a variety of events including:

- A request to dock or undock from a station, which may result in:
 - Availability of new resources such as a docking station drive or network drive.
 - Loss of resources currently in use, as discussed previously.
- A request to insert or disconnect a removable drive or other device such as a modem card, network card, SCSI controller, and so on.

While such changes might or might not be immediately relevant to your application, the ability to recognize such events would permit:

- Adapting operations to accommodate such changes.
- Warning the user that such a change has affected the available resources.

Recognizing Plug-and-Play messages

A Plug-and-Play message is received as a WM_COMMAND event message where the event identifier WM_DEVICECHANGE is found in the LOWORD value in the wParam argument. Accompanying this in the HIWORD value is an identifier for the type of the event. The event types are shown in Table 9.2.

TABLE 9.2: Plug-and-Play Event Messages

Value	Meaning
DBT_QUERYCHANGECONFIG	Requests permission to change the current configuration (dock or undock).
DBT_CONFIGCHANGED	Current configuration has been changed, due to docking or undocking.
DBT_CONFIGCHANGECANCELED	A request to change the current configuration (dock or undock) has been canceled.
DBT_CUSTOMEVENT	A custom event has occurred. (<i>Win98/Win2000 only</i>)
DBT_DEVICEARRIVAL	A new device has been inserted and is now available.
DBT_DEVICEQUERYREMOVE	Permission is requested to remove a device. This request can be denied by any application and the removal will be canceled.
DBT_DEVICEQUERYREMOVEFAILED	A request for removal has been canceled.
DBT_DEVICEREMOVEPENDING	A device is about to be removed. For advice only; cannot be denied.
DBT_DEVICEREMOVECOMPLETE	A device has been removed.
DBT_DEVICETYPESPECIFIC	A device-specific event has occurred. This is a generic interface allowing for media-specific information to be obtained from a hardware device.
DBT_USERDEFINED	The meaning of this message is user-defined.

In the same message the lParam argument provides a pointer to a PDEV_BROADCAST_HDR structure, where the dbch_devicetype field identifies the type of device. The device type is found as: ((PDEV_BROADCAST_HDR) lParam)->dbch_devicetype. The device types are listed in Table 9.3.

TABLE 9.3: Plug-and-Play Device Types

Value	Meaning
DBT_DEVTYP_OEM	OEM-defined device type
DBT_DEVTYP_DEVNODE	Devnode number (Win95/98 specific, not valid in WinNT/2000)
DBT_DEVTYP_VOLUME	Logical volume (drive, may be either local or remote)
DBT_DEVTYP_PORT	Serial or parallel port

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Paralleling Plug-and-Play events are two other event messages that deserve note here: the WM_DISPLAYCHANGE and WM_POWERBROADCAST messages.

The WM_DISPLAYCHANGE message is triggered by a change in the display resolution (mode) and permits applications to recognize and respond to changes in color depth and screen resolution during operation.

The WM_POWERBROADCAST message is a feature of the *Advanced Power Management (APM)* system (introduced by Windows 95) and allows applications to recognize situations where automatic power-down provisions have changed system behavior. While this is usually applied to the monitor, many laptops—to conserve battery power—also perform other types of shutdown including suspending drive and CPU operations even though they have not been turned off.

Incorporating any of these provisions in your application is, of course, completely dependent on the nature and needs of your program. Recognizing such events, however, is a trivial exercise.

The *Exceptions* Demo: Exception-Handling Examples

The *Exceptions* demo provides examples of several types of exception handling. From the program's menu, select the example to run:

Simple Handler Demonstrates a simple try/catch exception-handling structure

Nested Handler Demonstrates a nested try/catch exception-handling structure

Failed Catch Fails to catch a memory-access error; may require reboot

Resource Exception Fails to load a nonexistent resource

User Exception Throws a user exception

For convenience, each of the examples displays a series of text messages in the application window, tracing the execution of the exception handling. Ideally, you should compile the example in debug mode, place breakpoints in the source code, and watch the execution directly, but the displayed messages offer an overview of the execution.

NOTE:

The *Exceptions* demo is included on the CD accompanying this book, in the Chapter 9 folder.

A Simple Exception Handler

Beginning with the simplest example in the demo, a try/catch block is interrupted when the ForceException procedure is called to create a memory-access exception. The ForceException procedure was discussed briefly earlier in this chapter, in the “What Is Exceptional?” section. Figure 9.2 shows the display where the messages trace the execution of the procedure.

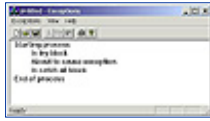


FIGURE 9.2 Displaying actions in a simple exception handler

NOTE:

Perhaps this is overly simple and too obvious to mention, but the message display would not be part of a conventional application. These messages are used here only to demonstrate the flow of events.

In this case, the exception has been trapped in the catch(...) block and has not produced a system fault.

```
void CExceptionsView::OnExceptionsSimplehandler()
{
    CDC      *pDC = GetDC();
    CSize     cSize = pDC->GetTextExtent( "A", 1 );
                                   // get vertical size for text
    int       vSize = cSize.cy,
             iLine = 0;
    CString  csText;
```

Each of the subroutines begins by getting a device context and then using the GetTextExtent function to determine the vertical size of a line of text so that we can use this value for spacing when displaying the messages. Also, the Invalidate and OnPaint calls are used to erase any text that may have been left behind from a previous example.

```
Invalidate();
OnPaint();           // erase anything previous
csText = "Starting process";
pDC->TextOut( 10, vSize*iLine++, csText );
try
{
    csText = "In try block";
    pDC->TextOut( 50, vSize*iLine++, csText );
    csText = "About to cause exception";
    pDC->TextOut( 50, vSize*iLine++, csText );
    ForceException(); // shown previously
```

Even though the exception occurs in a separate subroutine, execution of the try block is suspended and passed to the catch(...) block. Therefore, the next line of text will not be displayed in the view window (see Figure 9.2). Instead, execution resumes in the catch block, where a different message is presented.

```
    csText = "This line will not display because execution"
            " has passed to the catch block";
    pDC->TextOut( 50, vSize*iLine++, csText );
}
catch(...)           // catch everything!!!
{
    csText = "In catch all block";
    pDC->TextOut( 50, vSize*iLine++, csText );
}
csText = "End of process";
```

```

    pDC->TextOut( 10, vSize*iLine++, csText );
}

```

In a real application, the catch block might have any number of provisions, ranging from warning the user about the problem to executing some specific recovery operations. What exactly should occur, of course, depends on the exception and what operations were attempted.

A Nested Exception Handler

The second example in the demo, using nested exception handling, places one try/catch block inside an outer try block. Also, the inner catch block has a series of catch statements, illustrating how the various CException-derived classes might be employed. Last, a throw statement in the inner catch block throws the trapped exception to the outer catch block.

```

void CExceptionsView::OnExceptionsNestedhandlers()
{
    CDC      *pDC = GetDC();
    CSize     cSize = pDC->GetTextExtent( "A", 1 );
    int       vSize = cSize.cy, iLine = 0, *p = 0x00000000;
    CString   csText;

    Invalidate();
    OnPaint(); // erase anything previous
    csText = "Starting process";
    pDC->TextOut( 10, vSize*iLine++, csText );
    try
    {
        csText = "In outer try block";
        pDC->TextOut( 50, vSize*iLine++, csText );
        try
        {
            csText = "In inner try block";
            pDC->TextOut( 90, vSize*iLine++, csText );
            csText = "About to cause exception ...";
            pDC->TextOut( 90, vSize*iLine++, csText );

            ForceException();

```

At this point, we're in the inner try block, and ForceException is about to produce an exception event, throwing execution to the inner catch block and preventing the next instructions from executing.

```

            csText = "This line will not display because execution "
                    "has passed to the catch block";
            pDC->TextOut( 50, vSize*iLine++, csText );
        }
    }

```

Next, we have a series of catch blocks, each one explicitly trying to catch a particular type of exception defined by one of the CException-derived classes. However, this particular exception—a memory-access error—is not covered by any of these; therefore, the actual catch will not occur until a suitable handler is reached.

Because the body of each of these catch statements is essentially the same except for the display string, these have been truncated to save space.

```

    catch( CMemoryException *e ) // out of memory
    {
        TCHAR    szCause[255];

        csText = "CMemoryException cause: ";
        e->GetErrorMessage( szCause, 255 );
        csText += szCause;
        pDC->TextOut( 90, vSize*iLine++, csText );
    }

```

```

    }
    catch( CFileException *e )           // file exception
    {
        ...
    }
    catch( CArchiveException *e )       // Archive/Serialization
    {                                   // exception
        ...
    }
    catch( CNotSupportedException *e )  // response to request for
    {                                   // unsupported service
        ...
    }
    catch( CResourceException *e )      // resource allocation
    {                                   // exception
        ...
    }
}

```

Because the demo application was not created using database support, the CDaoException and CDBException classes are not recognized and are commented out.

```

/*           // no database support; ergo, CDaoException and
           // CDBException are not recognized in this example
    catch( CDaoException *e )           // database exceptions
    {                                   // (DAO classes)
        ...
    }
    catch( CDBException *e )           // database exceptions
    {                                   // (ODBC classes)
        ...
    }
*/
    catch( COleException *e )           // OLE exceptions
    {
        ...
    }
    catch( COleDispatchException *e )   // dispatch (automation)
    {                                   // exceptions
        ...
    }
    catch( CUserException *e )          // exception alerts the user
    {                                   // with message box, then
        ...                             // throws generic CException
    }

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Finally, the CException catch appears after the derived classes. Otherwise, as mentioned earlier, the compiler would flag the derived catch blocks as errors.

```
catch( CException *e )           // must follow all derived
{                                 // classes or compiler
    TCHAR    szCause[255];       // will flag error

    csText = "CException cause: ";
    e->GetErrorMessage( szCause, 255 );
    csText += szCause;
    pDC->TextOut( 90, vSize*iLine++, csText );
}
```

And, last, since we know that none of these CException classes will actually handle this particular exception, we include the catch(...) block, which catches everything but is not very informative about what has happened.

```
catch(...)                       // catch everything!!!
{
    csText = "In inner catch all block: ";
    pDC->TextOut( 90, vSize*iLine++, csText );
    csText = "Throwing exception to outer catch block: ";
    pDC->TextOut( 90, vSize*iLine++, csText );
    throw;    // will throw exception to outer catch block below
}
```

TIP:

While using the catch(...) block is valid for preventing a program fault, it's better to try the CException classes first. You want to use only those classes that are relevant to the exception(s) you actually want to trap, so that you can identify the exception and take appropriate recovery actions.

The exception occurred in the inner try block, then was captured in the inner catch block, and is now being

thrown to the outer catch block. The throw statement is simply a way to prevent the code following the try/catch blocks from being executed. This is essentially the same handling that occurred automatically in the inner try block when the original exception occurred. The only difference is that here, because of the exception entering the inner catch block, we have decided that we do not want the outer try block to continue either. In brief, this is simply an option to provide additional control over how the application executes, and it allows us to prevent the following program instructions from operating.

```

        csText = "This line will not display because execution has "
                "been thrown to the outer catch block";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
    catch(...) // catch everything!!!
    {
        csText = "In outer catch all block, "
                "catching thrown exception";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
    csText = "End of process";
    pDC->TextOut( 10, vSize*iLine++, csText );
}

```

The flow report generated by the nested exception example is shown in Figure 9.3. The report explains how the exception has been caught and redirected to the outer catch block.

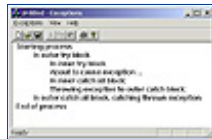


FIGURE 9.3 The flow report for the nested exception example

NOTE:

Perhaps some of you remember using GOTO and LABEL instructions for similar purposes. In one sense, the throw function is less flexible; in other respects, it is simply more rigorous in its structuring. But however you view it, the throw statement does provide a means of bypassing a block of instructions in response to a failed circumstance. In many circumstances, this may be exactly what you need to prevent further or more serious malfunctions.

A Failed Catch Example

The failed catch example illustrates what happens when a try/catch combination attempts to guard execution of an area of code but fails to correctly trap an exception; that is, the program encounters an exception that is not trapped. The code for this example is essentially the same as the code for the simple exception handler:

```

void CExceptionsView::OnExceptionsFailedcatch()
{
    CDC      *pDC = GetDC();
    CSize     cSize = pDC->GetTextExtent( "A", 1 );
    int       vSize = cSize.cy, iLine = 0, *p = 0x00000000;
    CString   csText;

    Invalidate();
    OnPaint(); // erase anything previous
    csText = "Starting process";
    pDC->TextOut( 10, vSize*iLine++, csText );
    try
    {
        csText = "In try block";
        pDC->TextOut( 50, vSize*iLine++, csText );
        csText = "About to cause exception ...";
        pDC->TextOut( 90, vSize*iLine++, csText );
    }
}

```

In addition to the text display, the routine also displays a modal dialog message with a warning, as shown in Figure 9.4, because this routine is intended to fail.

```

        MessageBox( "WARNING!\r\n"
                    "This exception may require a reboot\r\n"
                    "for full recovery!" );
        *p = 999;
    }
    catch( CException *e )           // CException will not catch the
    {                                // memory access error ...

        TCHAR    szCause[255];

        csText = "In catch block";
        pDC->TextOut( 50, vSize*iLine++, csText );
        csText = _T("CException cause: ");
        e->GetErrorMessage( szCause, 255 );
        csText += szCause;
    }

```

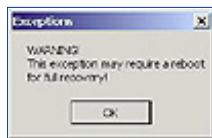


FIGURE 9.4 The warning for the failed catch example

NOTE:

In the event that the Failed Catch example is executed under Windows 98, the results may either terminate the demo application or corrupt system memory to the point of requiring a reboot to recover. We cannot be sure which of these events will occur because Windows 98 does not protect memory as rigorously as Windows NT/2000 does; but in either case, you are appropriately warned.

The catch(...) handler is commented out because this is the only handler that can catch the memory-access fault, and the point of this example is what happens when the exception is not handled.

```

/*          // enabling the catch(...) block will
           // prevent the system from trapping the error
    catch(...)           // catch everything ... including
    {                   // the memory access error!
        csText = "In catch all block";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
*/
    csText = "End of process";
    pDC->TextOut( 10, vSize*iLine++, csText );
}

```

You could enable the catch(...) block to prevent the failure.

A Resource Exception Example

Not all exceptions are automatically generated by the system. In many cases, you will want to raise an exception because you have detected a malfunction or error and need to branch execution or avoid executing subsequent instructions. This task is best accomplished by throwing an exception to a catch block, for reporting or for alternative handling.

For example, you might be translating a lengthy string table for internationalization and one (or more) strings might be lost in the process. (Accidents do happen.) As long as the resource identifier was not lost, the code will still compile, even though there are instructions to retrieve a string table entry that does not exist. This mishap does not in itself raise an exception. Instead, the LoadString instruction simply returns a zero length, reporting that the item was not found. In most cases, you will not have even bothered to check the return value. However, the resource exception example in the demo does check and uses the error to raise

an exception.

The resource exception example begins with both an inner and outer try block, to illustrate the throw operation again. In the inner try block, it attempts to load a string resource.

```
void CExceptionsView::OnExceptionsResourceexception()
{
    CDC      *pDC = GetDC();
    CSize     cSize = pDC->GetTextExtent( "A", 1 );
    int       vSize = cSize.cy, iLine = 0;
    CString   csText, csMsg;

    Invalidate();
    OnPaint(); // erase anything previous
    csText = "Starting process";
    pDC->TextOut( 10, vSize*iLine++, csText );
    try
    {
        csText = "In outer try block";
        pDC->TextOut( 50, vSize*iLine++, csText );
        try
        {
            csText = "In inner try block";
            pDC->TextOut( 90, vSize*iLine++, csText );
            csText = "About to cause exception ...";
            pDC->TextOut( 90, vSize*iLine++, csText );
            if( ! csText.LoadString( IDS_NON_STRING ) )
                AfxThrowResourceException();
            pDC->TextOut( 90, vSize*iLine++, csText );
        }
    }
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

If the LoadString operation succeeded, the application would have proceeded by displaying the string. However, since there is no corresponding string table entry, even though the Resource.h file has a definition for the resource identifier, AfxThrowResourceException is called and throws an exception, forcing execution into the following catch blocks.

```
catch( CResourceException *e ) // resource allocation exception
{
    TCHAR    szCause[255];

    csText = "CResourceException cause: ";
    e->GetErrorMessage( szCause, 255 );
    csText += szCause;
    pDC->TextOut( 90, vSize*iLine++, csText );
}
```

Here, the CResourceException class does catch the exception and reports the cause of the exception as "A required resource was unavailable," as you can see in Figure 9.5.

```
catch( CException *e )           // must follow all derived classes
{                                 // or compiler will flag error
    TCHAR    szCause[255];

    csText = "CException cause: ";
    e->GetErrorMessage( szCause, 255 );
    csText += szCause;
    pDC->TextOut( 90, vSize*iLine++, csText );
}
catch(...)                       // catch everything!!!
{
    csText = "In inner catch all block: ";
    pDC->TextOut( 90, vSize*iLine++, csText );
    csText = "Throwing exception to outer catch block: ";
}
```

```

        pDC->TextOut( 90, vSize*iLine++, csText );
        throw;    // will throw exception to outer catch block below
    }

```



FIGURE 9.5 Raising a resource exception

Notice that while the catch(...) handler does have a throw instruction, the CResourceException handler did not. Since the exception has been trapped (handled) in the CResourceException handler, the throw instruction is not executed. Therefore, the outer try block continues execution and displays the following message.

```

        csText = "Still in the outer try block. "
                "This line displays because";
        pDC->TextOut( 50, vSize*iLine++, csText );
        csText = "execution was not thrown to the outer catch block";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
    catch(...)                                // catch everything!!!
    {
        csText = "In outer catch all block, catching thrown exception";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
    csText = "End of process";
    pDC->TextOut( 10, vSize*iLine++, csText );
}

```

By using different exception handlers, depending on the exception, the application can respond in different fashions. And, always remember, the catch handlers may contain a variety of different responses to the exception, as appropriate.

TIP:

This example uses a string resource, but any type of missing resource could be justification for raising a resource exception. You might consider experimenting with other types, such as dialog boxes or custom resource types.

A User Exception Example

Like a resource exception, a user exception must almost always be raised explicitly. The system does not have any way of recognizing a user error because these errors occur within the application's context. It is up to you to define the user exception. Raising a user exception is not particularly different from raising a resource exception.

In most cases, when a user error occurs, you should simply have provisions in place to recognize them, to alert the user, and to suggest corrective actions. Raising an exception should not be the automatic response. Still, there may be circumstances where the only practical way to avoid more serious problems as a result of a user mistake is to raise an exception and to throw execution to a catch handler for resolution.

Because the exception does not necessarily need to occur in the same procedure as the try/catch blocks, the demo uses a subprocedure to illustrate this fact.

```

BOOL CExceptionsView::UserException()
{
    // assume something has gone wrong
    AfxMessageBox( "Drat! The XDR Veng operation failed!" );
    AfxThrowUserException();
    return TRUE;
}

```

Here, the UserException procedure displays a message box reporting a failure, as shown in Figure 9.6, and then uses AfxThrowUserException to raise an exception.

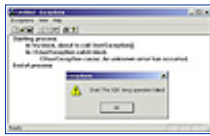


FIGURE 9.6 Raising a user exception

The UserException procedure is called from the following process, where the try/catch blocks appear.

```
void CExceptionsView::OnExceptionsUserexception()
{
    CDC          *pDC = GetDC();
    CSize        cSize = pDC->GetTextExtent( "A", 1 );
    int          vSize = cSize.cy, iLine = 0, *p = 0x00000000;
    CString      csText, csMsg;

    Invalidate();
    OnPaint();           // erase anything previous
    csText = "Starting process";
    pDC->TextOut( 10, vSize*iLine++, csText );
    try
    {
        csText = "In try block, about to call UserException()";
        pDC->TextOut( 50, vSize*iLine++, csText );
        if( UserException() )
        {
            csText = "In try block, the XDR Veng operation succeeded "
                    "(impossible)";
            pDC->TextOut( 50, vSize*iLine++, csText );
        }
        csText = "Continuing try block";
        pDC->TextOut( 50, vSize*iLine++, csText );
    }
}
```

Since we expect UserException to throw an exception, the message that reports the operation as successful will never be executed. If you trace this process, you will not see a return value from the UserException subprocessure, because execution has been thrown directly to the catch handlers.

```
catch( CUserException *e )
{
    TCHAR        szCause[255];

    csText = "In CUserException catch block";
    pDC->TextOut( 50, vSize*iLine++, csText );
    csText = "CUserException cause: ";
    e->GetErrorMessage( szCause, 255 );
    csText += szCause;
    pDC->TextOut( 90, vSize*iLine++, csText );
}
```

In the CUserException handler, the cause for the exception is reported as "unknown." After all, the CUserException class has no way to know why the exception was raised. However, by using a custom-derived class, you could provide explanations and include mechanisms for identifying different types of exceptions.

```
catch( CException *e )
{
    TCHAR        szCause[255];

    csText = "In CException catch block (unexpected)";
    pDC->TextOut( 50, vSize*iLine++, csText );
    csText = "CException cause: ";
    e->GetErrorMessage( szCause, 255 );
}
```

```
        csText += szCause;
        pDC->TextOut( 90, vSize*iLine++, csText );
    }
    csText = "End of process";
    pDC->TextOut( 10, vSize*iLine++, csText );
    return;    // No exception thrown
}
```

Summary

This concludes our introduction to using exception handling. Exceptions are a useful tool and, in addition to the *Exceptions* demo, a number of other examples are included on the CD accompanying this disk. Additional information is also available in the online documentation that is included on your Visual C++ CD.

Also note that although this chapter did not demonstrate the use of the TRY and CATCH macros (in favor of using the clearer try/catch block instructions), these macros are still worth consideration.

In the next chapter, we move to another aspect of Windows 2000 application development: memory management.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 10 Memory Management

- How the Virtual Memory Manager handles memory
- Virtual memory commands
- Heap management
- File-mapping objects for sharing memory

Besides the flashy interface and the multitasking, one of Windows' great attractions has always been memory management. From the 8086 target machine of version 1 all the way up through the 80486, 16-bit Windows has always helped programs make full use of whatever resources the system offered. But MS-DOS, the underlying trellis on which Windows grew, was inherently a 1MB system, and much of the potential of a 32-bit processor remained untapped.

The first crack in the 1MB barrier came with the protected addressing mode of the 80286 processor, which offered access to more memory but was incompatible with the DOS real mode. The result was a flurry of proposed memory management standards and DOS extender programs blazing new routes into the frontiers. The methods were ingenious. Temporarily unneeded blocks of code or data were swung up like a Murphy bed into higher memory, and clever programmers insinuated their code into the lowest levels of the system software to hijack allocation requests and smuggle goods across the 1MB border. The original 80286 was never designed for one program to run in two modes. When switching from one to the other, the CPU ground to a halt and reset itself. Like Janus, the two-faced god of doorways, the DOS extender must stand with one leg on each side of the abyss. 16-bit Windows never entirely freed itself from this schizophrenic existence; it just became better adjusted to its inherent neuroses.

Beginning with Windows NT, continuing with Windows 95/98 and now in Windows 2000, Windows has fully transcended the restrictions of DOS and Windows NT/2000 is a true protected-mode system, where attempts to address illegal memory space result in the errant application being terminated. (Unfortunately, Windows 95/98 is not so fully protected, and errant applications are more likely to crash the system than to be terminated by the system.)

NOTE:

You can expect to see—in the near future—a 64-bit version of NT/2000 which will finally catch up to the 64-bit capacities provided by such contemporary processors as the Pentium, Athlon, Celeron, etc. Also, even without a fully 64-bit OS, in Chapter 18, we will introduce contemporary 64-bit operations supported by the MMX instruction set.

This chapter begins by summarizing the hardware mechanisms and system policies that underlie the memory management in Windows 2000 and explaining the translation from virtual to physical memory and the paging scheme that supports the protected address spaces. A new set of virtual memory API routines gives you control over individual memory pages. Win32 also adds improved heap management while still preserving the old global and local memory functions from the Win16 API. After discussing these memory-management commands, we'll see how processes in protected address spaces can share blocks of memory by creating file-mapping objects.

Memory-Management Concepts

The Windows 2000 memory management API, working with an imagined logical address space of 4GB, is supported internally by the *Virtual Memory Manager* (VMM), which in turn bases its services on the features of advanced 32-bit processors. Windows 2000 requires its hardware to use 32-bit memory addresses, to map virtual addresses to physical addresses, and to perform memory paging. From these basic capabilities, the VMM constructs its own mechanisms and policies for managing virtual memory.

NOTE:

Windows NT/2000 memory management differs from Windows 95/98 memory management in one way: Under Windows NT/2000, the VMM works with an imagined logical address space of 4GB, while under Windows 95/98, the VMM is limited to a mere 2GB. Currently, both of these limits are well in excess of anything we expect to see physically installed in any system (in the near future, at least). No doubt, when RAM in excess of 1GB becomes a common feature, some extension will be introduced to allow addressing spaces well in excess of either limit. Functionally, this single issue aside, the memory management APIs perform essentially the same for both systems.

The Virtual Memory Manager (VMM)

Like virtual reality, virtual memory isn't the real thing but does a good job of pretending to be. The 4GB of memory addresses that every program commands is a virtual space. The system does not contain 4GB of physical memory, yet somehow a program is able to use any address in the full range. Obviously, a translator somewhere in the background must silently convert each memory reference into an existing physical address. From this silent conversion, the VMM draws much of its power.

NOTE:

Technically, under Windows NT/2000, the VMM creates a 4GB memory space for each process (application) but reserves the upper 2GB for system use and only makes the lower 2GB available for the process. Hereafter, we will refer only to the 2GB of memory space that is actually available to the process.

The VMM is the part of the Windows system responsible for mapping references to a program's virtual address space into physical memory. It decides where to put each memory object and which memory pages should be written to disk. It also isolates each process in a separate address space by refusing to translate virtual addresses from one process into the physical memory used by another process. It supports the illusion of an idealized, logical space of 2GB for each and every application, just as the GDI supports the illusion that every program draws to idealized logical display coordinates. The system translates logical addresses or coordinates into physical locations and prevents programs from colliding with each other by trying to use the same resource at the same time.

Pointers and Movable Memory

MS-DOS programmers customarily work with pointers to memory objects. If the object moves to another location, the pointer becomes invalid. On a machine that runs only one program at a time, pointers work fine. But multitasking systems need to manage system memory more actively, loading and discarding pieces on demand and moving pieces to make more room.

The first versions of Windows got around the problem by substituting *handles* for pointers wherever possible. A handle points to an entry in an object table, and the table remembers where each object actually is at any given moment. When the system moves an object, it updates the table. The handles pointing to the table remain valid even when the object moves, because they point to the place where the real pointer is kept, and the system keeps the real pointer current.

Processors running in a protected mode provide a similar layering mechanism for all memory addresses. Instead of working with pointers containing segments and offsets, Intel CPUs work with *selectors* and offsets. A selector is to memory what a handle is to Windows—a pointer to a pointer.

To give a simplified account, the selector, with its offset, refers to a system table that keeps track of memory addresses. In a sense, protected mode has built-in memory handles. The difference is that selectors live at a much lower level in the system than handles do. The hardware knows what a selector is—the CPU can decode them—but only Windows knows what a handle is. Windows can have handles only where Windows itself creates them. You must make an effort to use handles instead of pointers, but protected mode gives you selectors whether you want them or not.

Even when you lock down memory, the system lets you see only the selector, not the physical address. Your program doesn't care. The code that worked with pointers works just as well with selectors, only now it doesn't matter if the system moves what your selector "points" to. The operating system can move objects through memory with impunity as long as it remembers to update the corresponding memory table entry. The entry may change, but the selector does not. The selector always points to the same place in the table, and the table is guaranteed to point to your memory object. As a consequence, you can safely use and preserve a pointer without impeding any other program.

The conversion from a selector and its offset to a physical address involves two stages: a trivial conversion from a logical to a virtual address, and a complex conversion from a linear address to a physical address. The first conversion, from a logical segmented address to a virtual linear address, simply adds an offset to a base address. Win32 makes this conversion trivial by keeping the base address set always to 0. If the segment is constant, only the offset matters. The 32-bit offset may range from 0 to 2GB, exactly the range of the system's flat linear addresses. Under Win32, a 32-bit selector addresses all of virtual memory in much the same way that small-model programs address their entire segment with a single near pointer. But a linear address is still not a physical address. In the second and more complex conversion, the VMM parses the linear address into indices for the process's paging tables, which contain physical addresses.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Paging

The use of virtual addresses offers many benefits, among them the illusion of working in a very large space. Win32 allows the virtual addresses to range from 0 to 2GB, regardless of the physical memory actually installed on the current system. Obviously, a problem arises if all the running programs try to allocate all the virtual memory they think they see all at once. Like a bank, the system goes suddenly broke if all its customers cash in at the same time. Physical memory always imposes some kind of upper limit, but a paging system raises the limit considerably by setting off part of a hard disk to act as additional memory.

The CPUs on which Windows runs have built-in memory-paging capabilities. Small blocks of memory called *pages* can be saved to the disk when not in use to make more room. A *page interrupt* (or *page fault*) occurs whenever a program tries to read a part of memory that has been moved to the disk. The program doesn't ever know the memory is gone; the CPU generates an error, the operating system reloads the missing page, and the program resumes. If blocks of memory can be saved and restored as needed, nothing prevents the system from over-committing memory. If you have a thousand pages of memory, you could have two thousand pages' worth of code. It can't all be in memory at once, but any block can always be loaded when needed. Of course, disks are slower than RAM and virtual memory always works best if your hard drive is large, fast, and partly empty.

Much of the VMM's energy goes into moving and recovering pages of memory. Windows can work with pages of up to 64KB in size, but the Intel CPU enforces a size of 4KB. A 4KB block of memory aligned on a 4KB boundary is called a *page frame*. The term *page* refers to the data a program stores in a page frame. There are usually more pages than page frames; some of the pages have been saved in the paging file. Although the contents of a page remain the same, those contents may appear in different page frames at different times as the VMM adjusts to the demands of all the current programs.

Page Databases and Tables In order to satisfy requests for memory, the VMM maintains several data structures. It must be able, for example, to traverse a list of page frames to see which are free and which are full and also to determine which process currently owns any given page. (Obviously, this also requires each process to occupy some integral number of pages regardless of their actual requirements.)

The structure that holds this set of information is the *page frame database*. Through the database, the VMM, given any page, can find the process that owns it. Another data structure, called a *page table*, works in the other direction. Every process has at least one page table; given any process, the VMM can find all the pages

it owns.

When the VMM assigns physical memory to a process, it updates the database and also adds an entry to a page table. Whenever a process refers to a virtual address, the VMM looks up the address in a page table to find the associated memory. Besides a virtual address and its corresponding physical address, the page table entry records other information about each page, including whether the page is protected (read-only, for example) and whether it is in memory or swapped to the disk. A swapped page is marked *invalid* in the page table (meaning that it resides in the swap file rather than in memory).

For the convenience of the hardware, a single page table takes up exactly 4KB. One 4KB page table has room for 1024 different entries. Each entry points to a single page. If one page table can point to 1024 pages and each page is 4KB, then one page table can address 4MB of memory (1024×4096). A process that uses more memory receives more page tables.

Each process has a single master table, called a *page directory*, pointing to all its page tables. A page directory also holds up to 1024 entries. With 1024 page tables, each addressing 4MB, a process can reach up to 4GB ($4\text{MB} \times 1024$). Because each page directory and page table occupies exactly 4KB, or one page frame, the system can easily swap the directory and tables in and out of memory along with all the other pages as needed.

The diagram in Figure 10.1 should help clarify how page directories point to page tables that point to pages, and at the same time the page frame database keeps a separate list of each page's current status. Most memory operations occur more rapidly than this elaborate indexing scheme seems to allow, because the CPU caches frequently used virtual address translations in a fast hardware buffer called the *Translation Look-aside Buffer (TLB)*.

Page Frame States The page frame database records the current state of each page frame. A page frame is always in one of six states (or status markers):

reserved page A page of memory in a process's virtual address space which is reserved for future use. A reserved page is not accessible by a process and does not have any associated physical storage. The reserved page simply reserves a range of virtual addresses, preventing these from being subsequently used by other allocation operations.

committed page A page of memory in a process's virtual address space which has been allocated physical storage either in RAM or on the disk.

free page A page of memory in a process's virtual address space which is available for use. Before a process can use the page, the page must be assigned as reserved or committed; a free page can not be accessed.



FIGURE 10.1 How the system finds a process's physical memory through the page directory and page tables

WARNING:

One of the more obvious ways for a page fault to occur is for an application to attempt to perform its own memory operations and to write to an invalid address (an address that the application or process does not own). Under Windows NT/2000, where operations are very tightly controlled, the usual result is that the process will be terminated by the system. Under Windows 95/98, memory addressing is not so thoroughly policed and improper addressing may trash the system itself.

For security reasons, any page frame assigned to a process must be zeroed first. No process may receive memory that still contains information left behind by another program. When the list of available zeroed pages runs low, the memory manager reads through the page frame database following all the links to free pages and zeroes them. If memory is still low, the VMM wakes one of its threads to find modified pages and save them, slowly and one by one in the background, so as not to interfere with system performance.

Policies and Working Sets

It should be clear by now that the VMM keeps a constant watch over the status of all the system pages. If one process uses more physical memory than allowed in its security quota, the VMM steps in and invalidates some of its pages by marking them standby or modified, and eventually their page frames may be released. The group of page frames currently in use by one process is the process's *working set*. Like an embezzler, the VMM sneaks through the system and steals frames out from under processes as they run, hoping no one will notice. When a process begins to complain with a flurry of page faults, the VMM placates it by enlarging its working set. Busy programs end up with bigger sets.

The VMM follows defined policies in deciding when to retrieve swapped pages, where to place restored pages in memory, and what to swap out first when the system needs more room. This retrieval policy is called demand paging with clustering. *Demand paging* means the VMM loads pages only when a program asks for them, rather than trying to minimize delays by anticipating what a program will need. The system does try to anticipate to some extent by *clustering*; thus, in response to each demand, the system actually loads several adjacent pages, figuring that memory operations often center on a single region, so nearby pages may be needed soon.

The placement policy determines where reloaded pages are stored. The VMM tries to put them on the first zeroed page frames it finds in the page frame database. When it runs out of zeroed frames, it begins searching the lists of frames in other states. When deciding on pages to swap out, the VMM follows a local FIFO ("first in, first out") replacement policy. *Local* in this context means that the system makes room for one page in a process by dropping another page from the same process. By keeping the working set for each process to a roughly constant size, the system prevents one program from monopolizing resources. Within one process's working set, the pages that have been in memory longest are the first to go. Pages a program touches often may be marked invalid, but the process will probably recover them from their transitional standby or modified state before the VMM actually saves the contents and zeroes the frame.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Commitments and Reservations

Now that you know the VMM is sneaky because it steals pages, and that it's lazy because it often invalidates them without bothering to discard them, perhaps you won't be surprised to learn that it sometimes only pretends to allocate the memory you request.

To allocate memory, the VMM must construct page tables and search the page frame database for suitable zeroed areas. It may even need to find other invalidated pages and prepare them for use, possibly readjusting the working sets of other programs. If a program allocates a large virtual space but ends up using only parts of it, the VMM's efforts will have been largely wasted. Instead of allocating physical memory, the VMM often simply marks part of the process's virtual address space as being in use without securing physical memory to back it up. It *reserves* memory without *committing* it.

Reserving memory is like paying with a promissory note. When a program later tries to use some of the reserved memory, cashing in its IOU, the hardware notices because it can't find a page table entry for the given virtual address. It issues a page fault. The VMM steps in and finally pays its debt, committing physical pages to fulfill an allocation. Of course, even as it commits new page frames for one request, it may be invalidating other pages to make room. The entire sequence of reserving addresses, receiving a page fault, and committing new memory is intended to be invisible to the process.

Virtual Address Descriptors To support the illusion of a vast address space, the VMM requires yet another data structure. A tree of *virtual address descriptors* (VADs) records each allocation and sub-allocation that the process makes in its range of virtual addresses. Whenever a program allocates memory, the VMM creates a VAD and adds it to the tree, as shown in Figure 10.2. A VAD records the range of addresses an allocation claims, the protection status for all pages in the range, and whether child processes may inherit the object contained in the range. If a thread uses an address that falls outside the range of any of its VADs, the VMM knows the address was never reserved and recognizes an access violation.



FIGURE 10.2 Tree of VADs

Constructing a VAD is much simpler than constructing a page table and filling it with valid page frame addresses. Furthermore, the size of the allocation has no effect on the speed of the response. Reserving 2KB is no faster than reserving 2MB; each request produces a single VAD. When a thread actually uses the reserved memory, the VMM commits page frames by copying information from the descriptor into new page table entries.

Virtual Memory APIs Often, programs have no need to concern themselves with the differences between reserved and committed memory. Usually it's enough to know that memory will be available when you expect it to be. Among its new features, however, Win32 boasts a set of virtual memory APIs that give you precise control over reserving, committing, and protecting pages of memory. By using these APIs, you can allocate a very large memory object, fill it only partially, and not waste any memory. The usual example is a spreadsheet, because most of its cells are likely to be empty with data clustering together in a few areas. If you reserve a large range of addresses to represent the entire spreadsheet as an array, the VMM commits physical memory only for the areas actually in use, and it still allows convenient access to any part of the array through a full range of continuous addresses.

The Address Space

Most of what you've read about so far goes on behind the scenes. From the perspective of a running program, what matters most is not the page tables and working sets but the 2GB of virtual address space. Figure 10.3 shows how the system organizes the space.

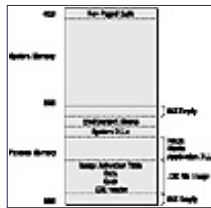


FIGURE 10.3 The virtual addresses the system uses for different entries in the address space of a single process (please note that this representation is not to scale)

Although a process does indeed run in a 4GB address space, the process gets to use only 2GB of that space. The system reserves the upper half of the space for itself. The high half of the address space is the same for every application, but only kernel-mode threads may use it. In other words, it is accessible only to the operating system. At the very highest addresses, the system keeps critical system code that cannot be paged out of memory; for example, the part of the VMM that performs paging operations is stored at the high addresses.

All the pieces of memory over which you have control are mapped into the lower 2GB of the address space. That's where the code and data for a process reside, along with a stack for each thread, at least one default heap, and the program's own DLLs. The system always loads a process's code near the bottom of the address space at the 64KB mark. The 64KB of space at the top of a process's 2GB space also remains permanently empty. The "no-man's land" at either end helps the system identify invalid pointers. For example, 0 is never a valid address.

At the lowest available address is an image of the process's .EXE file, including its header, code, data, and an *image activation table*. The activation table aids in dynamic linking. When the linker finds a `CreateWindow` command in your code, for example, it cannot know in advance where in memory to find the `CreateWindow` code. For all such unresolved dynamic links, the linker creates an entry in the image activation table. When the system loads an executable file, it searches for all the DLL entry points listed in the activation table, locates the appropriate DLL, and fixes the table to include the current entry addresses. The linker speeds loading by guessing where the DLLs will be and by providing tentative addresses. The Win32 subsystem always tries to map its system DLLs to the same addresses near the top of the 2GB space. If, upon loading a program, the system discovers that the DLLs are indeed in the expected place, then the activation table does not need fixing. (You can specify a preferred loading address for your own DLLs, too.)

The loader may not actually copy an entire .EXE file into memory. It is more likely to reserve virtual addresses for the .EXE file and let the VMM commit physical pages later if the code refers to those addresses. The Win32 subsystem DLLs have been structured to cluster related commands in adjacent addresses. As a result, common calling sequences usually require a minimum of new pages to be loaded.

An application's own DLLs, heaps, and stacks may be allocated anywhere in the address space between the .EXE file image and the system DLLs. Normally, the first allocation begins at 0x00010000, or 64KB.

Mapped Files

Alert readers may already have wondered what happens when a user initiates several instances of a single program. Given that every process has its own secure address space, must the system load a new copy of the .EXE file for each new instance? If not, how do two processes share the block of memory that contains the file image? The situation calls for a new strategy and one widely useful in many other situations as well.

Normally, the VMM protects programs from each other by ensuring that a virtual address from one process never translates to a page frame in use by another process. Because the VMM always translates every virtual address into a physical address, no program reaches memory directly, and the VMM easily routes every memory access to a safe location. The scheme also allows for the VMM to make a single page frame visible in the virtual addresses of different processes. At the operating-system level, a block of shared memory is called a *section object*.

The Win32 subsystem exposes the functionality of section objects to its clients in the form of *memory-mapped files*. Two programs cannot share memory directly, but they can share a single memory-mapped disk file. Of course, most of what the process perceives as physical memory is already in a disk-swapping file. In effect, the VMM lets you retrieve information from the swap file by reading from particular memory addresses. In fact, you can access any disk file the same way, using memory addresses, by creating a memory-mapped file. As an extension of this memory I/O capability, two processes may open the same block of memory as though it were a file, read from it, and write to it. To share memory without creating a new disk file, the programs link the shared object to the system's normal page-swapping file.

When the user launches multiple instances of a program, the system creates a mapped-file object to enable all instances to share a single copy of the .EXE file image.

Memory Management Commands

The Win32 memory management commands fall into three main categories: virtual memory functions, heap functions, and the familiar global and local allocation functions. Each set includes commands to allocate and free blocks of memory, but each set manages memory a little differently. We'll survey each group and compare their advantages, look briefly at a few related commands, and finish this section with the API for memory-mapped files.

Virtual Memory Commands

The virtual memory commands, which have names like VirtualAlloc and VirtualFree, expose some of the VMM's operations that the other two command sets hide. With the virtual memory commands, you can imitate the VMM by reserving addresses without committing memory to support them and by protecting ranges of memory with read-only, read/write, or no-access flags. You can also lock pages in memory to prevent them from being swapped to disk. The other command sets are built on top of the virtual memory commands; these are the basic operations from which the Win32 API builds its other memory services.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Allocating Memory

The VirtualAlloc command is the starting point for managing your own virtual address space. Its parameters tell how much memory to allocate, where in the address space to situate the new block, whether to commit physical memory, and what kind of protection to set.

LPVOID VirtualAlloc

```
(  
    LPVOID lpvAddress,           // desired address for new block  
    DWORD  dwSize,               // size of new memory block  
    DWORD  fdwAllocationType,    // reserve addresses or  
                                // commit memory  
    DWORD  fdwProtect            // no access, read-only,  
                                // or read/write  
);
```

VirtualAlloc tries first to find a range of free addresses marking a block of dwSize bytes beginning at lpvAddress. To do this, it searches the process's tree of VADs. If the requested range is free, the function returns lpvAddress. If some part of it is in use, the function searches the entire address space looking for any sufficiently large free block. If it finds one, it returns the starting address. Otherwise it returns NULL.

Most often, programs set the first parameter to NULL and allow VirtualAlloc to place the block anywhere. Controlling the placement of a block may occasionally be useful if, for example, you are debugging a DLL that usually loads at a particular address. By using VirtualAlloc to reserve that address before the DLL loads, you can force it to another location.

The fdwAllocationType parameter may be MEM_COMMIT, MEM_PHYSICAL, MEM_RESERVE, MEM_RESET, MEM_TOP_DOWN, or MEM_WRITE_WATCH.

MEM_COMMIT Physical storage is allocated in memory or in the paging file on disk for the specified region of pages.

MEM_PHYSICAL Physical memory is allocated—solely for use with Address Windowing Extensions (AWE) memory.

MEM_RESERVE Reserves a range of the process's virtual address space but does not allocate any

physical storage.

MEM_RESET Specifies that the pages within the range identified by the `lpAddress` and `dwSize` arguments will not be written to or read from the paging file. Declares that the address range is free for reuse and the contents do not need to be saved or maintained.

MEM_TOP_DOWN Memory is allocated at the highest possible address.

MEM_WRITE_WATCH System tracks pages written to in the allocated region—requires the `MEM_RESERVE` flag to be used.

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.

To reserve a range of addresses, `VirtualAlloc` simply makes a new VAD marking an area in use. It does not, however, allocate any physical memory, so the reserved addresses cannot yet be used for anything. Attempts to read or write reserved pages produce access-violation exceptions. On the other hand, no other allocation command may use previously reserved addresses either. `GlobalAlloc` and `malloc`, for example, cannot place new objects in a range that overlaps with reserved space. If you call `VirtualAlloc` to reserve your entire 2GB of address space, all subsequent allocations will fail, even though `VirtualAlloc` has not yet taken up any physical memory.

Reserving addresses has no effect on the system's physical memory. The VMM makes no guarantee that physical pages will be available when you begin to commit a reserved area. Only when you commit memory does the VMM find pages to support it. As a consequence, the system does not charge reserved memory against a process's system resource quotas. Only memory actually in use counts against the quota.

WARNING:

A problem arises when you try to write to pages that have been reserved but never committed. The system generates an access-violation exception. You might choose to call `VirtualQuery` before every read or write to be sure the page is committed, but that takes time. The usual practice, demonstrated in the *List* demo (discussed later in this chapter), is to let the exception arise and provide an exception handler to deal with it. The handler calls `VirtualAlloc` to commit the required page and lets the program continue.

Memory cannot be committed without being reserved. By combining the `MEM_RESERVE` and `MEM_COMMIT` flags, you can reserve and commit at the same time. More often, programs call `VirtualAlloc` once—using `MEM_RESERVE`—to reserve a large area, and then many times subsequently—using `MEM_COMMIT`—to commit parts of the area piece by piece.

The `fdwProtect` flag determines how a page or range of pages may be used. For memory that is only reserved, the flag must be `PAGE_NOACCESS`. When committing memory, you can optionally change the protection to `PAGE_READONLY` or `PAGE_READWRITE`. No other programs can read memory in your address space anyway, so read-only protection guards against bugs in your own program that might accidentally corrupt some important part of your data. Protection levels apply to individual pages. The pages in a single range of memory may have different protection flags. You can, for example, apply `PAGE_READONLY` to an entire block and temporarily change single pages to allow write access as needed. You cannot write-protect just part of a page. The protection flags apply to entire pages.

Any of the following flags can be combined with the `PAGE_GUARD` and `PAGE_NOCACHE` protection modifier flags:

PAGE_READONLY Read-only access to the committed region of pages is enabled with write access disabled. I.e., attempting to write to the committed region will result in an access violation. If the system differentiates between read-only access and execute access, any attempt to execute code in the committed region will also result in an access violation.

PAGE_READWRITE Both read and write access to the committed region of pages are enabled.

PAGE_EXECUTE Execute access to the committed region of pages is enabled while any attempt to read or write to the committed region will result in an access violation.

PAGE_EXECUTE_READ Execute and read access to the committed region of pages are enabled but any attempt to write to the committed region will result in an access violation.

PAGE_EXECUTE_READWRITE Execute, read, and write access to the committed region of pages are enabled.

PAGE_GUARD Pages in the committed region become guard pages. This means that any attempt to read

from or write to a guard page will cause the system to raise a STATUS_GUARD_PAGE exception and to turn off the guard page status. In this fashion, guard pages act as a one-shot access alarm.

The PAGE_GUARD flag is a page protection modifier and can be combined with any of the other page protection flags except for the PAGE_NOACCESS flag . When the guard page status is turned off (by an access attempt) 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 All access to the committed region of pages is disabled and any attempt to read from, write to, or execute in the committed region will result in an access violation exception (a general protection (GP) fault).

PAGE_NOCACHE The committed regions of pages are not permitted to be cached. Likewise, the hardware attributes for the physical memory should also be specified as *no cache*.

PAGE_NOCACHE is not recommended for general usage but is useful for device drivers as, for example, when mapping a video frame buffer with no caching. This flag is valid only as a page protection modifier when used with a page protection flag other than PAGE_NOACCESS.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

VirtualAlloc cannot reserve more than 2GB because a process has control over only the bottom half of its 4GB address space. In fact, the actual limit is slightly smaller because of the 64KB free area at either end of a process's 2GB space (see Figure 10.3, earlier in this chapter). Also, VirtualAlloc reserves memory in 64KB blocks and commits memory in one-page blocks. When reserving memory, VirtualAlloc rounds lpvAddress down to the nearest multiple of 64KB. When committing memory, if lpvAddress is NULL, VirtualAlloc rounds dwSize up to the nearest page size boundary. If lpvAddress is not NULL, VirtualAlloc commits every page containing any bytes in the range lpvAddress to lpvAddress + dwSize. A 2-byte allocation, if it crosses a page boundary, would require the commitment of two entire pages. On most Windows 2000 systems, a page is 4KB, but if you need to know the size, call GetSystemInfo.

Uses for Reserved Memory

The ability to reserve memory without committing it is useful for dynamically allocated structures and sparse arrays. A thread that expands some structure (perhaps a list) as the program runs can reserve room to prevent other threads from using up addresses the structure may need as it expands.

The reserved area does set an upper limit on the size of the structure because there is no such command as "VirtualReAlloc" to expand a reserved area. Resizing requires allocating a second block, copying the first block into it, and freeing the original. On the other hand, given a 2GB range of possible addresses, you can reasonably set the upper size limit of the original allocation quite high.

Reserved memory also makes it easy to create a sparse array (a large array) with only a few elements filled. For example, a spreadsheet is a sparse array of empty cells with only a few positions occupied, and those positions are usually clustered in adjacent areas. With the virtual memory commands, you can reserve a large address space for all the possible cells and commit memory only for the areas in use. Although there are other ways for spreadsheet programs to minimize their allocations, the virtual memory solution to sparse arrays is particularly convenient, because you can still address the array as a range of contiguous addresses.

Freeing Memory

When a process ends, the system automatically releases any memory that was still in use. To free memory sooner, call VirtualFree.

```

BOOL VirtualFree
(
    LPVOID lpvAddress,           // address of memory to free
    DWORD  dwSize,              // size of memory block
    DWORD  fdwFreeType           // decommit or release
);

```

VirtualFree can decommit a set of committed pages (leaving their addresses still reserved) or it can release a range of reserved addresses, or it can do both at once. Decommitting can release small blocks, and the blocks may include a mix of both reserved and committed pages.

When you are releasing reserved addresses, you must free the entire range of addresses as originally allocated, and all the pages in the range must be in the same state—either all reserved or all committed. lpvAddress must contain the base address previously returned by VirtualAlloc, and the value of dwSize is ignored because the whole range is freed at once. dwSize matters only when decommitting sections of a range.

The fdwFreeType argument specifies how the committed memory is freed and can be either (but not both) of the following flags:

MEM_DECOMMIT Decommits the specified region of committed pages. Since attempting to decommit an uncommitted page does not result in a failure, a range of committed or uncommitted pages can be decommitted without error.

MEM_RELEASE Releases the specified region of reserved pages. If MEM_RELEASE is specified, the dwSize argument must be zero; otherwise, the operation will fail.

Before decommitting a page, be sure no part of it is still in use. The *List* demo, discussed later in the chapter, fits four list items on each 4KB page. Deleting one item does not make it safe to delete a page, because the other three items might still be in use.

Programs that use the virtual memory commands usually need some kind of garbage-collection mechanism to decommit pages when they become empty. The mechanism could be a low-priority thread that occasionally cycles through an allocated area looking for entirely empty pages.

Protecting Memory

After reserving address space, you call VirtualAlloc again to commit individual pages and VirtualFree to decommit or release them. When committing pages, VirtualAlloc also changes the protection state from no-access to read-only or read/write. To change the protection state for a page already committed, call VirtualProtect.

```

BOOL VirtualProtect
(
    LPVOID lpvAddress,           // address of memory to protect
    DWORD  dwSize,              // size of area to protect
    DWORD  fdwNewProtect,       // new protection flags
    PDWORD pfdwOldProtect       // variable to receive old flags
);

```

lpvAddress and dwSize indicate the range of addresses to protect. The two flag parameters each contain one of the familiar protection flags: PAGE_NOACCESS, PAGE_READONLY, or PAGE_READWRITE. Flags apply to whole pages. Any page even partially included in the given range will be changed. The pfdwOldProtect parameter returns the previous state of the first page in the range.

VirtualProtect works only with pages already committed. If any page in the range is not committed, the function fails. The pages in the range do not, however, need to have identical protection flags.

The primary advantage of protection is in guarding against your own program bugs. To see an example, refer to the revised AddItem and DeleteItem procedures used in the *List* demo described later in this chapter.

Querying Memory Information

Sometimes you need to get information about a block of memory. Before writing to a page, for example,

you might want to find out whether the page has been committed. VirtualQuery fills a structure variable with information about a given block of memory:

```
DWORD VirtualQuery(
    LPVOID lpvAddress,          // address of area to be described
    PMEMORY_BASIC_INFORMATION pmbiBuffer,
                                // address of description buffer
    DWORD dwLength );          // size of description buffer

typedef struct _MEMORY_BASIC_INFORMATION /* mbi */
{
    PVOID BaseAddress;          // base address of page group
    PVOID AllocationBase;       // address of larger allocation unit
    DWORD AllocationProtect;    // allocation's initial access
                                // protection
    DWORD RegionSize;           // size, in bytes, of page group
    DWORD State;                // committed, reserved, free
    DWORD Protect;              // group's access protection
    DWORD Type;                 // type of pages (always private)
} MEMORY_BASIC_INFORMATION;
typedef MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;
```

The lpvAddress parameter of VirtualQuery points to an arbitrary address. Any given location in memory may be part of two different allocation units. It may be part of a large block of reserved pages, and it may also be part of a smaller region of pages subsequently committed, decommitted, or protected together. A region consists of all contiguous pages with the same attributes.

In the BaseAddress field, VirtualQuery returns the address of the first page in the smaller region that contains lpvAddress. The AllocationBase field returns the address of the larger allocation that first reserved lpvAddress. AllocationBase matches the value returned previously by VirtualAlloc. Whatever protection flags the original VirtualAlloc applied to the range are returned in AllocationProtect (MEM_NOACCESS, MEM_READONLY, or MEM_READWRITE).

The other fields describe the smaller subgroup of like pages, giving its size, current status, and protection flag. The last field always returns MEM_PRIVATE, indicating that other processes cannot share this memory. The existence of this field suggests that Microsoft may later consider adding other mechanisms for processes to share memory.

Although they are not part of the virtual memory command set, two other commands also retrieve information about memory. GlobalMemoryStatus returns the total size and remaining space for physical memory, the page file, and the current address space. GetSystemInfo returns, among other things, the system's physical page size and the lowest and highest virtual addresses accessible to processes and DLLs. (Generally these values are 4KB, 0x00010000, and 0x7FFEFFFF.)

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Locking and Unlocking Pages

Two other virtual memory commands lock and unlock pages. A locked page cannot be swapped to disk while your program executes. When your program is not currently executing, however, all of its pages, including locked pages, may be swapped to disk. In effect, locking a page guarantees that it will become a permanent part of the program's working page set. In a busy system, the working set manager may reduce the number of pages a process may lock. The maximum for any process is approximately 30 to 40 pages. The exact value varies slightly with the size of system memory and the application's working set.

Locking memory is discouraged because it constrains the VMM and makes organizing physical memory more difficult. For the most part, only device drivers and other system-level components lock any of their pages. A program that must respond very rapidly to system signals might lock some pages to ensure that unexpected disk reads don't delay the response.

```
BOOL VirtualLock( LPVOID lpvAddress,    // start of area to lock
                  DWORD  dwSize );      // size of area to lock
```

```
BOOL VirtualUnlock( LPVOID lpvAddress, // start of area to unlock
                   DWORD  dwSize );     // size of area to unlock
```

There is no lock count on virtual memory. VirtualLock commands do not always require a matching VirtualUnlock. You can, for example, lock three contiguous pages with three different commands and then unlock them all with a single command. All the pages must already be locked, but the range does not need to correspond exactly with the range given in any previous lock command.

Before being locked, memory must be committed. When a process ends, the system automatically unlocks any remaining locked pages. VirtualFree releases pages even if they are locked.

NOTE:

Be aware that GlobalLock and VirtualLock do very different things. GlobalLock simply translates handles into pointers. It locks an allocated object into a virtual address but has no effect at all on physical memory. VirtualLock, on the other hand, is more severe. It locks pages, not objects, and the locked pages are forced into physical memory whenever the program runs.

Heap Functions

A *heap* is a block of memory from which a program allocates smaller pieces as needed. A 16-bit Windows program draws memory from both a global heap and a local heap. The local heap is faster, but is limited to 64KB.

For Windows 2000 (as well as Windows 95/98/NT), a flat address space abolishes the difference between *global* and *local* and between *near* and *far*, making the entire address space a single, undifferentiated heap.

Even with such a large, contiguous memory space, working from a smaller heap sometimes still makes sense. Reserving and committing virtual memory has obvious advantages for large dynamic or sparse structures. But what about algorithms that call for many small allocations? The heap memory commands allow you to create one or more private heaps in your address space and suballocate smaller blocks from them.

The heap commands conveniently group allocations together in a small section of the address space. Clustering allocations serves several purposes:

- It can separate and protect related allocations. A program that makes many small allocations that are all the same size can pack memory most efficiently by making them contiguous. A heap allows that.
- If all of your linked-list nodes come from one heap and all of your binary-tree nodes come from another heap, a mistake in one algorithm is less likely to interfere with the other.
- Memory objects used in conjunction with each other should be grouped together to minimize page swapping. If several addresses happen to reside on the same memory page, a single disk operation retrieves all of them.

Creating a Heap

The memory you get from a heap is just like the memory you get any other way. In fact, you can write your own heap implementation using the virtual memory commands; that's exactly what the Windows subsystem does. Heap commands make internal calls to the virtual memory API. To create a heap, you give a starting size and an upper limit:

```
HANDLE HeapCreate(  
    DWORD dwOptions,           // heap allocation flag  
    DWORD dwInitialSize,       // initial heap size  
    DWORD dwMaximumSize );    // maximum heap size
```

Behind the scenes, the Win32 subsystem responds by reserving a block of the maximum size and committing pages to support the initial size. Subsequent allocations make the heap grow from the bottom to the top. If any allocation calls for new pages, the heap commands automatically commit them. Once committed, they remain committed until the program destroys the heap or the program ends.

The system does not manage the inside of a private heap. It does not compact the heap or move objects within it. Therefore, a heap may become fragmented if you allocate and free many small objects. If allocations fill the heap to its maximum size, subsequent allocations fail. If `dwMaximumSize` is 0, however, the heap size is limited only by available memory.

The `dwOptions` parameter allows a combination of three flags to be set:

HEAP_GENERATE_EXCEPTIONS Instructs the system to raise an exception—such as an out-of-memory condition—to indicate failure. Without this flag, `HeapAlloc` indicates failure by returning `NULL`.

HEAP_NO_SERIALIZE Specifies that mutual exclusion will not be used while the `HeapAlloc` function is accessing the heap (see notes following).

The **HEAP_NO_SERIALIZE** flag should not be specified when accessing the process heap since 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 Ensures that allocated memory is initialized as zero. (By default, memory is not cleared on allocation.) If the function succeeds, it allocates at least as much memory as requested, and may allocate slightly more to reach a convenient boundary.

By default, without any flags (the argument is passed as 0), the heap prevents threads that share memory handles from interfering with each other. A serialized heap disallows simultaneous operations on a single handle. One thread blocks until another finishes. Serialization slows performance slightly. A program's heap does not need to be serialized if the program has only one thread, if only one of its threads uses the heap, or if the program itself protects the heap, perhaps by creating a mutex or a critical section object (see Chapter 6, "Creating and Synchronizing Multiple Threads," for more information about threads, mutexes, and critical section objects).

Allocating from a Heap

HeapAlloc, HeapReAlloc, and HeapFree—like the Win16 commands their names recall—allocate, reallocate, and free blocks of memory from a heap. All of them take as one parameter a handle returned from HeapCreate.

```
LPSTR HeapAlloc(  
    HANDLE hHeap,           // handle of a private heap  
    DWORD  dwFlags,         // control flags  
    DWORD  dwBytes );       // number of bytes to allocate
```

HeapAlloc returns a pointer to a block of the requested size. It may include any combination of the three control flags discussed for the HeapCreate function.

NOTE:

Specifying any of these flags will override the corresponding flags specified in the dwOptions parameter when the heap was created using HeapCreate.

To discover the exact size of any block, call HeapSize. Besides the bytes in the block itself, each allocation consumes a few extra bytes for an internal supporting structure. The exact size varies but is near 16 bytes. You need to know this only because it may prevent you from squeezing as many allocations out of one heap as you expect. If you create a 2MB heap and attempt two 1MB allocations, the second one is likely to fail.

To change the size of a memory block after it has been allocated, call HeapReAlloc.

```
LPSTR HeapReAlloc(  
    HANDLE hHeap,           // handle of a private heap  
    DWORD  dwFlags,         // flags to influence reallocation  
    LPSTR  lpMem,           // address of memory block to reallocate  
    DWORD  dwBytes );       // new size for the memory block
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Besides the three flags discussed previously which HeapAlloc uses to zero memory, to deny serialization, and to generate exceptions, the dwFlags parameter of HeapReAlloc accepts one other flag: HEAP_REALLOC_IN_PLACE_ONLY. (To the best of our knowledge, five words are a record for Microsoft's manifest constants.) This flag prevents HeapReAlloc from moving the memory block to a more spacious area. If other nearby allocations prevent the block from expanding in place, this flag makes HeapReAlloc fail rather than relocate. (HEAP_REALLOC_IN_PLACE_ONLY would normally be used with one of the other flags.)

NOTE:

Specifying any of these flags will override the corresponding flags specified in the dwOptions parameter when the heap was created using HeapCreate.

Destroying a Heap

When you have no more use for an allocated block, release it with HeapFree. When you have no more use for the heap itself, release it with HeapDestroy.

```
BOOL HeapFree(  
    HANDLE hHeap,           // handle of a private heap  
    DWORD dwFlags,          // may be HEAP_NO_SERIALIZE  
    LPSTR lpMem );         // address of a memory block to free
```

NOTE:

The HeapFree function accepts HEAP_NO_SERIALIZE as the single option flag.

```
BOOL HeapDestroy( HANDLE hHeap );
```

Freeing a memory block does not decommit the page it occupied, but it does make the space available for subsequent allocations in the same heap. HeapDestroy decommits and releases all the pages in the heap whether or not the heap still contains allocated blocks. After HeapDestroy, the hHeap handle is invalid (undefined).

Global and Local Memory Commands

In making the transition from a 16-bit to 32-bit system, many of the earlier system and memory API commands have been dropped. Most of the obsolete memory commands, such as `AllocSelector`, refer to low-level features specific to Win16 or to Intel CPUs. For backward compatibility, many of the more familiar memory commands are retained:

GlobalAlloc	LocalAlloc
GlobalDiscard	LocalDiscard
GlobalFlags	LocalFlags
GlobalFree	LocalFree
GlobalHandle	LocalHandle
GlobalLock	LocalLock
GlobalMemoryStatus	No local equivalent
GlobalReAlloc	LocalReAlloc
GlobalSize	LocalSize
GlobalUnlock	LocalUnlock

The Win32 environment forces a few semantic changes. Most important, both sets of commands, global and local, now work with the same heap. On loading, every process receives a default heap, and the old API commands work from that.

It is now legal (though confusing) to mix global and local commands in a single transaction. For example, you could, just to be perverse, allocate an object with `GlobalAlloc` and release it with `LocalFree`. Also, the 32-bit pointer that `LocalLock` now returns is indistinguishable from the 32-bit pointer that `GlobalLock` returns.

The default heap expands as needed, limited only by physical memory. Even the humble `LocalAlloc` can allocate megabytes. The default heap automatically serializes operations to prevent different threads from corrupting the heap by using the same handle at the same time.

Pages allocated by `GlobalAlloc` or `LocalAlloc` are committed and marked for read/write access. Unlike `HeapFree`, `GlobalFree` checks for empty pages and releases them back to the system. The allocation commands still recognize the flags that make memory fixed or movable, but with Windows 2000's paging and virtual addressing, even "fixed" memory moves. The only practical use for `GMEM_FIXED` is to make `GlobalAlloc` return a pointer instead of a handle.

A few of the other flags are now ignored, including `LOWER`, `NOCOMPACT`, `NODISCARD`, `NOT_BANKED`, and `NOTIFY`. More significant than the loss of these minor flags is the loss of `GMEM_DDESHARE`. Like other Win32 object handles, handles to allocated memory refer to the object table of a specific process. A handle passed from one program to another becomes invalid in the new address space. `DuplicateHandle` makes it possible for processes to share some handles, but it fails on memory handles. The `GMEM_DDESHARE` flag still exists (is still defined) because Microsoft apparently plans for it to signal some optimization appropriate for DDE conversations, but the old method of sharing memory handles is simply not supported. Sharing a block of memory now requires a memory-mapped file.

The Win32 global and local memory commands differ from the heap commands in creating movable and discardable objects. Objects created by `HeapAlloc` do not change their virtual address (though their physical address may change). The local and global functions, at the expense of more memory management overhead, do move objects to minimize fragmentation. With a memory manager as flexible as the VMM, discardable memory is less important than it used to be.

The primary advantage of the old API functions is the obvious one: They are portable. To write source code that ports easily from 16 bits to 32 bits, limit yourself to the Win16 memory commands. In Win32, the global and local sets are interchangeable, but you should pick the set that would make the most sense in a Win16 program. For small, fast allocations, use the local functions. For large allocations, use the global functions. Under Windows 2000 both sets perform alike, but since their advantage is portability, you should use them in a portable fashion.

The heap and virtual memory command sets are faster and more efficient than the older commands. You can allocate from the default heap with less overhead by doing this:

```
HeapAlloc( GetProcessHeap(), 0, dwSize );
```

`GetProcessHeap` returns a handle to the default heap.

WARNING:

Do not pass the handle returned by `GetProcessHeap` to `HeapDestroy`.

System Limits

Because 32-bit Windows does not use the descriptor tables that limited 16-bit Windows to a system-wide total of 8192 handles, the new system supports many more allocations. Nevertheless, some limits remain:

- `VirtualAlloc` never reserves an area smaller than 64KB, so allocating 32,767 blocks fills up the 2GB user address space.
- `VirtualAlloc` cannot create more than 32,767 (32K) handles in one process. (`HeapAlloc` has no such limit; Microsoft says it has created over a million handles on a single heap in tests.)
- `GlobalAlloc` and `LocalAlloc` combined cannot create more than 65,535 (64K) handles. The limit applies only to movable objects, however.

NOTE:

Memory objects allocated using the `GMEM_FIXED` or `LMEM_FIXED` flags do not return handles. Instead, they return direct pointers to the memory allocated.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

Validation Commands

Another set of commands tests the validity of pointers. Each receives a virtual address and returns TRUE if the process does not have a particular access privilege. Call these to make your programs more robust by testing values before using them. The validation commands are listed in Table 10.1.

TABLE 10.1: Validation Commands

Function	Argument	Validation Test
IsBadCodePtr	Pointer to a function	Tests for read access to the beginning of the function
IsBadReadPtr	Pointer to a memory block	Tests for read access to a range of addresses
IsBadStringPtr	Pointer to a string	Tests for read access to all bytes up to the end of the string
IsBadWritePtr	Pointer to a memory block	Tests for write access to a range of addresses

WARNING:

Be aware that other threads and even other processes (such as debuggers) could conceivably change the contents or protection of a memory page between the time an IsBad function confirms validity and the time you try to use the address yourself.

C Runtime Equivalents

Under Windows, the malloc family of C runtime functions calls system heap routines internally. The C routines work perfectly well in Windows 2000, although they do not perform the same suballocations that they perform in the 16-bit versions.

The runtime memory buffer commands such as memset now have competition from four new Win32 routines: CopyMemory, FillMemory, MoveMemory, and ZeroMemory. However, these commands link dynamically to the system DLLs only on a MIPS machine. Obviously, they provide no special benefits for Windows 2000 applications at all. Instead, on an x86 machine, all four commands map back to the standard

C runtime functions: memcpy, memmove, and memset.

The List Demo: Reserving and Committing Memory

The *List* demo, shown in Figure 10.4, creates a list by reserving virtual memory and committing it as the user enters new items. A list box filling the window's client area displays the entire list. The List menu allows you to add and delete items.



FIGURE 10.4 List program accepting a new item entry

To create a dynamic list of identically sized nodes in 16-bit Windows, you would probably create a structure of linked nodes. Each new entry would require an allocation, using GlobalAlloc (or malloc with MSC 7.0). An array would be easier to program but would be much more wasteful, since it would allocate more memory than it ever used. Under Windows 2000, however, the virtual memory commands make a large dynamically allocated array quite practical, and that's how *List* implements its data structure.

Writing a virtual memory program calls for a decision about how to deal with uncommitted pages. The List demo program uses exception handling to commit pages on demand as page faults occur. A second approach would be to have procedures to query the state of each page before writing and to keep all pages write-protected except when actually writing new data to specific pages.

NOTE:

The *List* demo is included on the CD that accompanies this book, in the Chapter 10 folder.

Setting the List's Size and Structure

The header file defines two constants governing the size and structure of the list. Each array element has room for 1024 characters. Because the system's page size (4KB) is a multiple of the element size (1KB), no list item will cross a page boundary; therefore, retrieving an item never requires reading more than a single page from the disk. The *List* demo sets the array's maximum size to a mere 500, figuring that filling even that small number will challenge the patience of most readers. Setting the limit to 20,000 would still tie up less than 1 percent of the 2GB user address space.

Among the program's global variables are two arrays: iListLookup and bInUse.

```
int iListLookup[MAX_ITEMS]; // matches index to array position
BOOL bInUse[MAX_ITEMS];    // marks array positions in use
```

The lookup array links lines in the list box to elements in the dynamic array. If iListLookup[4] is 7, then string 5 in the list box is stored in element 7 of the array. (The list box and the array both begin numbering at 0.)

The second array contains a Boolean value for each element in the dynamic array. Whenever the program adds or deletes a string, it changes the corresponding element of bInUse to TRUE for an occupied position and FALSE for an empty position. To add a new string, the program searches bInUse for an empty array element. To delete the currently selected string, the program locates the array element by referring to iListLookup.

Initializing Data Structures

When the program starts, the CreateList function is called to reserve memory and initialize the supporting data structures. VirtualAlloc reserves a 1MB range of addresses. Like all reserved pages, these must be marked PAGE_NOACCESS until they are committed.

```
BOOL CreateList()
{
    int i;
    // reserve one meg of memory address space
    pBase = VirtualAlloc( NULL, // starting address (anywhere)
```

```

MAX_ITEMS * ITEM_SIZE,    // one megabyte
MEM_RESERVE,              // reserve; don't commit
PAGE_NOACCESS );          // can't be touched

if( pBase == NULL )
{
    ShowErrorMsg( __LINE__ );
    return( FALSE );
}

// initialize the status variables
for( i = 0; i < MAX_ITEMS; i++ )
{
    bInUse[i] = FALSE;                // show no entries in use
    iListLookup[i] = 0;
}
bListEmpty = TRUE;                  // update global flags
bListFull = FALSE;
return( TRUE );
}

```

Adding an Item

When you choose Add Item from the *List* demo's menu, the AddItem procedure performs the following steps:

1. Locate the first empty slot in the array (iIndex).
2. Ask the user to enter the new string in a dialog box.
3. Copy the new string into the memory allocated during CreateList.
4. Update the list box and several global variables.

The first unused position in the array may happen to occupy an uncommitted memory page. In that case, the `lstcpy` command that tries to put a string there generates an exception. The exception handler around `lstcpy` looks for the `EXCEPTION_ACCESS_VIOLATION` signal and responds by calling `VirtualAlloc` to commit a single page from the previously reserved range.

```

void AddItem ( void )
{
    char szText[ITEM_SIZE];          // text for one item
    int  iLen;                       // string length
    int  iIndex;                     // position in array
    int  iPos;                       // position in list box
    int  iCount;                     // count of entries in list box
    BOOL bDone;                      // TRUE when free array entry found

    // determine the location of the first free entry
    bDone = FALSE;
    iIndex = 0;
    while( ( ! bDone ) && ( iIndex < MAX_ITEMS ) )
    {
        if( ! bInUse[iIndex] )        // is entry in use?
            bDone = TRUE;              // found an empty entry
        else
            iIndex++;                  // advance to next entry
    }

    // ask user for new text string
    iLen = GetItemText(szText);
    if( ! iLen ) return;
}

```

The `try` block copies the new text to an empty place in the item array. If that memory page is uncommitted, `lstcpy` raises an exception. The exception filter commits the page and the command continues.

```

try

```



```

{
    // put text in the item
    lstrcpy( &(amp; pBase[iIndex * ITEM_SIZE] ), szText );
}
except( CommitMemFilter( GetExceptionCode(), iIndex ) )
{
    // the filter does all the work
}
    // mark this entry as in use
bInUse[iIndex] = TRUE;
bListEmpty = FALSE;

```

Next, the program adds the new text to the list box. The string is inserted at iPos and iListLookup[iPos] is updated to indicate where in the item array the new entry was stored (iIndex).

```

iCount = ListBox_GetCount( hwndList );
iPos = ListBox_InsertString( hwndList, iCount, szText );
iCount++;
ListBox_SetCurSel( hwndList, iPos );
iListLookup[iPos] = iIndex;
if (iCount == MAX_ITEMS)           // did we fill the last place?
{
    bListFull = TRUE;
}
return;
}

```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

The CommitMemFilter function provides an exception-handling filter for the AddItem function. If a page fault occurs, CommitMemFilter attempts to commit the page and, on success, returns to lstrcopy and proceeds. If CommitMemFilter fails, the search for an appropriate exception handler continues.

```
LONG CommitMemFilter(
    DWORD dwExceptCode,    // code identifying the exception
    int iIndex )           // array element where fault occurred
{
    LPVOID lpvResult;
    // If the exception was not a page fault, then refuse
    // to handle it. Make the system keep looking for
    // an exception handler.
    if( dwExceptCode != EXCEPTION_ACCESS_VIOLATION )
    {
        return( EXCEPTION_CONTINUE_SEARCH );
    }
    // Try to commit the missing page.
    lpvResult = VirtualAlloc(
        &( pBase[iIndex * ITEM_SIZE] ), // bottom of area to commit
        ITEM_SIZE,                       // size of area to commit
        MEM_COMMIT,                      // new status flag
        PAGE_READWRITE );               // protection status
    if( ! lpvResult )                   // did allocation fail?
    {
        // if we can't commit the page,
        // then we can't handle the exception
        return( EXCEPTION_CONTINUE_SEARCH );
    }
    // The missing page is now in place. Tell the
    // system to go back and try again.
    return( EXCEPTION_CONTINUE_EXECUTION );
}
```

Deleting an Item

When DeleteItem removes an element from the virtual array, it also checks to see whether any other entries remain on the same memory page. If all four entries are empty, it decommits the page, releasing 4KB to the system. The virtual addresses that pointed to the page remain reserved. Whether or not it frees a page, DeleteItem then removes the string from the list box and updates the global status variables.

Two other procedures help with these tasks. GetBasePageEntry receives an array index and returns the index of the first element on the same page frame. In other words, it rounds down to the nearest multiple of four. AdjustLookupTable removes the entry it held for a newly deleted list box string and slides all of the subsequent elements up to fill in the gap.

```
void DeleteItem ( void )
{
    int  iCurSel; // position of current selection in list box
    int  iPlace;  // position of current selection in item array
    int  iStart;  // first position on same memory page as
                  // selection
    int  i;       // loop variable
    BOOL bFree;   // TRUE if all 4 entries on one page are unused
    BOOL bTest;   // for testing return results

    // retrieve memory offset of currently selected entry
    iCurSel = ListBox_GetCurSel( hwndList );
    iPlace = iListLookup[iCurSel];
    // zero out the deleted entry
    FillMemory( &(pBase[iPlace * ITEM_SIZE]), ITEM_SIZE, 0 );
    // mark this entry as free
    bInUse[iPlace] = FALSE;
```

Next, we determine which entry number is first on the current page. If all four entries on the page are empty, we'll decommit the page to release memory.

```
    iStart = GetPageBaseEntry( iPlace );
    bFree = TRUE;
    for( i = 0; i < 4; i++ )                // check four entries
    {
        if( bInUse[i + iStart] )            // in use?
            bFree = FALSE;                  // page is not free
    }
```

If a whole memory page is now unused, free it.

```
    if( bFree )                             // is page free?
    {                                         // YES; release it
        bTest = VirtualFree( &( pBase[iStart * ITEM_SIZE] ),
                             ITEM_SIZE, MEM_DECOMMIT );
        if( ! bTest )
        {
            ShowErrorMsg( __LINE__ );
            ExitProcess( (UINT)GetLastError() );
        }
    }
```

Last, update the list box display and the lookup table array, then check to see if any entries remain in the list.

```
    ListBox_DeleteString( hwndList, iCurSel );
    AdjustLookupTable( iCurSel );
    bListEmpty = TRUE;
    i = 0;
    while( ( i < MAX_ITEMS ) && ( bListEmpty ) )
```

```

    {
        // if the item is in use, then the list is not empty
        bListEmpty = !bInUse[i++];
    }
    // reposition the selection marker in the list box
    if( ! bListEmpty )
    {
        if( iCurSel )           // did we delete the first item?
        {                       // no; select item above deletion
            ListBox_SetCurSel( hwndList, iCurSel-1 );
        }
        else                     // deleted item was at top
        {                       // select new top entry
            ListBox_SetCurSel( hwndList, iCurSel );
        }
    }
    return;
}

```

The DeleteList function is called when the program ends to delete all the entries in the list and free the memory the entries occupied.

```

void DeleteList()
{
    // decommit the memory and then release the address space
    // Note: we must supply a size for the MEM_DECOMMIT operation
    if( ! VirtualFree( (void*)pBase, MAX_ITEMS*ITEM_SIZE,
                      MEM_DECOMMIT ) )
        ShowErrorMsg( __LINE__ );
    // now release the memory from the base address
    // – no size required
    if( ! VirtualFree( (void*)pBase, 0, MEM_RELEASE ) )
        ShowErrorMsg( __LINE__ );
    return;
}

```

Given an index into the list, GetPageBaseEntry determines which entry is first on the same page by finding the first integer divisible by four and less than or equal to iPlace.

```

int GetPageBaseEntry ( int iPlace )
{
    while( iPlace % 4 )
    {
        iPlace--;
    }
    return( iPlace );
}

```

When a list box entry is deleted, the array that matches the list box entries and the memory offsets must be updated. The iStart parameter gives the position in the list box from which a string was just deleted.

```

void AdjustLookupTable ( int iStart )
{
    int i;
    // This loop starts at the position where an entry
    // was just deleted and works down the list, scooting
    // lower items up one space to fill in the gap.

    for( i = iStart; i < MAX_ITEMS - 1; i++ )
    {
        iListLookup[i] = iListLookup[i + 1];
    }
}

```

```

    }
    iListLookup[MAX_ITEMS - 1] = 0;
    return;
}

```

NOTE:

The remaining procedures in the *List* demo run the dialog box where the user enters text, display the About box, and show error messages when something goes wrong. They appear in the complete listing on the CD.

Write-Protecting the List Pages

Instead of waiting for page faults and scurrying for last-minute commitments, an alternative version of the *List* demo could invest a little overhead and manage its array in a more deliberate fashion. Here are versions of *AddItem* and *DeleteItem* that call *VirtualProtect* before and after each modification, to ensure that every page is committed in advance and write-protected afterward.

```

void AddItem( void )
{
    MEMORY_BASIC_INFORMATION MemInfo;    // info about a memory block
    char szText[ITEM_SIZE];              // text for one item
    DWORD dwOldState;                    // memory status flags
    int iLen;                            // string length
    int iIndex;                          // position in array
    int iPos;                            // position in list box
    int iCount;                          // count of entries in list box
    BOOL bDone;                          // TRUE when free array entry found
    BOOL bTest;                          // for testing return values

    // determine the location of the first free entry
    bDone = FALSE;
    iIndex = 0;
    while( ( ! bDone ) && ( iIndex < MAX_ITEMS ) )
    {
        if( bInUse[iIndex] == 0 )        // is entry in use?
            bDone = TRUE;                // found an empty entry
        else
            iIndex++;                    // advance to next entry
    }
    // retrieve the text
    iLen = GetItemText(szText);
    if( ! iLen ) return;

```

Retrieve information about this entry's memory page. If it is committed, remove the access protection to allow reading and writing. If it is not committed, allocate it.

```

    // fill out a MEMORY_BASIC_INFORMATION structure
    VirtualQuery( &(amp; pBase[ITEM_SIZE * iIndex] ),
                  &MemInfo, sizeof(MemInfo) );
    if( MemInfo.State == MEM_COMMIT )
    {

```

The memory has already been committed. Change the access mode to permit reading and writing.

```

        bTest = VirtualProtect( &( pBase[ITEM_SIZE * iIndex] ),
                                // bottom of area to protect
                                ITEM_SIZE,                          // size of area to protect
                                PAGE_READWRITE,                    // new protection status
                                &pdwOldState );                    // old protection status
        if( ! bTest )
            ShowErrorMsg( __LINE__ );    // protection failed

```

```

}
else                                // this page is not yet committed
{

```

Since memory has not been committed yet, VirtualAlloc is called to commit a block at the bottom of the previous memory block.

```

LPVOID lpvResult;

lpvResult = VirtualAlloc( &(amp; pBase[iIndex * ITEM_SIZE] ),
                          // bottom of area to commit
                          ITEM_SIZE,                          // size of area to commit
                          MEM_COMMIT,                          // new status flag
                          PAGE_READWRITE );                    // protection status
if( ! lpvResult )
{
    ShowErrorMsg( __LINE__ );
    return;                                                    // allocation failed
}

// put text in the item
lstrcpy( &(amp;pBase[iIndex * ITEM_SIZE]), szText );
// restore the protection state of this page to read-only
bTest = VirtualProtect( &(amp;pBase[iIndex * ITEM_SIZE] ),
                        // bottom of area to protect
                        ITEM_SIZE,                          // size of area to protect
                        PAGE_READONLY,                        // new protection status
                        &dwOldState );                        // previous protection status
if( ! bTest )
{
    ShowErrorMsg( __LINE__ );
}

// mark this entry as in use
bInUse[iIndex] = 1;
bListEmpty = FALSE;

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.


```

        ITEM_SIZE,                                // size of area to protect
        PAGE_READWRITE,                          // new protection status
        &pdwOldState );                          // previous protection status
if( ! bTest )
{
    ShowErrorMsg( __LINE__ );
    return;
}
FillMemory( &(amp; pBase[iPlace * ITEM_SIZE] ), ITEM_SIZE, 0 );
    // mark this entry as free
bInUse[iPlace] = 0;

```

Determine which entry number is first on the current page. If all four entries on the page are empty, we'll decommit the page to release memory.

```

iStart = GetPageBaseEntry( iPlace );
bFree = TRUE;
for( i = 0; i < 4; i++ )                        // check four entries
{
    if( bInUse[i + iStart] )                    // in use?
        bFree = FALSE;                        // page is not free
}

```

If a whole memory page is now unused, free it. If not, restore its read-only protection.

```

if (bFree)                                        // is page free?
{
    // YES; release it
    bTest = VirtualFree( &(amp; pBase[iStart * ITEM_SIZE] ),
        ITEM_SIZE, MEM_DECOMMIT );
}
else
{
    // NO; protect it
    bTest = VirtualProtect( &(amp; pBase[ITEM_SIZE * iPlace] ),
        ITEM_SIZE, PAGE_READONLY, &pdwOldState );
}
if( ! bTest )
    ShowErrorMsg( __LINE__ );

```

Last, the list box display is updated along with the lookup table array, then a check is made to see if any entries remain in the list before repositioning the selection marker.

```

ListBox_DeleteString( hwndList, iCurSel );
AdjustLookupTable( iCurSel );
bListEmpty = TRUE;
i = 0;
while( ( i < MAX_ITEMS ) && ( bListEmpty ) )
{
    // if the item is in use then the list is not empty
    bListEmpty = !bInUse[i++];
}
    // reposition the selection marker in the list box
if( ! bListEmpty )
{
    if( iCurSel )                // did we delete the first item?
    {
        // no; select item above deletion
        ListBox_SetCurSel( hwndList, iCurSel - 1 );
    }
    else
    {
        // deleted item was at top
        // select new top entry
        ListBox_SetCurSel( hwndList, iCurSel );
    }
}

```

```

    }
    return;
}

```

File-Mapping Objects

For two processes to share a block of memory, they must create a file-mapping object. Such objects have two purposes: They facilitate file I/O and they create a memory buffer that processes can share. To explain sharing memory, we need to begin with files.

Mapped Files

After opening a disk file for use as a memory-mapped file, a program may optionally create an associated file-mapping object in order to treat the file as a block of memory. The system reserves a range of addresses from the process's space and maps them to the file instead of to physical memory. The process reads and writes to these addresses as it would to any other memory address, using functions like `lstrcpy` and `FillMemory`. You can even typecast the base address to make an array pointer and retrieve file records by indexing the array. The VMM responds to page faults in a file-mapping object by swapping pages from the disk file rather than the system's paging file.

This ingenious file I/O technique also allows processes to share memory. If two programs can open handles to the same file, what difference does it make if the file happens to be in memory? If the cooperating programs want to share memory but don't need to create a disk file, they can link their shared file-mapping object to the system's paging file. Then the memory in the mapped file is paged exactly the same way all other memory is paged.

Setting up a new mapped file requires three commands:

CreateFile Opens a disk file.

CreateFileMapping Returns a handle to a file-mapping object.

MapViewOfFile Links a region in the file to a range of virtual addresses and returns a pointer.

`CreateFile` should be familiar simply because this function is used for opening existing named pipes (see Chapter 6) as well as for opening conventional files.

`CreateFileMapping` creates a new system object, adds an entry to the process's object table, and returns a handle. The new object creates the potential for parts of the file to be held in memory, but you cannot actually read the file until you also create a view into it.

A *view* is a small section of a larger object—a window into a section of the file. `MapViewOfFile` creates a view by associating positions in the file with positions in the address space. Operations on that range of addresses become operations on the file.

You can create a view big enough to contain the entire file. However, in theory, a file can be much larger than your entire address space, so an alternative is necessary. The alternative is to create a smaller view and move it when you need to reach new parts of the file. After creating a file-mapping object, you can map, unmap, and remap your view of it over and over. You can even map several simultaneous views of a single file. The relationship between a file and its view is shown in Figure 10.5.

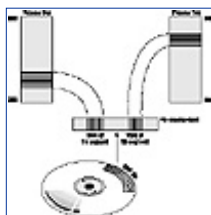


FIGURE 10.5 Two programs with different views of the same file-mapping object

Creating a File-Mapping Object

`CreateFileMapping` requires a file handle to associate with the new object. Normally you receive the handle from `CreateFile`. To share memory without creating a separate file, you may instead pass the file handle as `(HANDLE)0xFFFFFFFF`. The system then maps from the system paging file.

```

HANDLE CreateFileMapping(
    HANDLE hFile,                // handle of file to map
    LPSECURITY_ATTRIBUTES lpSa, // optional security attributes
    DWORD fdwProtect,           // protection for mapping object
    DWORD dwMaxSizeHigh,        // high-order 32 bits of object size
    DWORD dwMaxSizeLow,         // low-order 32 bits of object size
    LPTSTR lpzMapName );        // name of file-mapping object

```

The `fdwProtect` parameter sets the protection flag for all the memory pages the mapped file uses. It may be `PAGE_READONLY`, `PAGE_READWRITE`, or `PAGE_WRITECOPY`. The first two of these—`PAGE_READONLY` and `PAGE_READWRITE`—have been introduced previously but `PAGE_WRITECOPY` has not.

PAGE_WRITECOPY Provides copy on write access to the committed region of pages. However, the files specified by the `hFile` parameter must have been created with `GENERIC_READ` and `GENERIC_WRITE` access. See **Uses of Copy-On-Write Protection** following.

In addition to the protection flags, applications can also specify section attributes by combining (ORing) one or more of the section attributes with the preceeding page protection flags.

SEC_COMMIT (default) All pages of a section are allocated physical storage in memory or in the paging file on disk.

SEC_IMAGE Identifies the file specified for a section's file mapping as an executable image file. No other attributes are valid with `SEC_IMAGE`—the mapping information and file protection are taken from the image file.

SEC_NOCACHE All pages of a section are set as non-cacheable. `SEC_NOCACHE` requires either the `SEC_RESERVE` or `SEC_COMMIT` flag to also be set.

The `SEC_NOCACHE` attribute is intended for architectures requiring various locking structures to be in memory locations which are never fetched into the processor's memory. Using the cache for these structures on 80x86 and MIPS machines only slows down the performance as the hardware keeps the caches coherent. However, some device drivers require noncached data so that programs can write directly to the physical memory.

SEC_RESERVE All pages of a section are reserved without allocating physical storage.

The range of pages reserved are inaccessible to all other allocation operations until they are released although reserved pages can be committed in subsequent calls to the `VirtualAlloc` function. The `SEC_RESERVE` attribute is valid only if the `hFile` parameter is `(HANDLE)0xFFFFFFFF`; i.e., a file-mapping object backed by the operating system paging file.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The next two parameters of `CreateFileMapping`, both `DWORDs`, together tell the size of the file-mapping object. If the file-mapping object is smaller than the file it maps, not all of the file is accessible through it. The last part of the file is excluded from sharing. The system interprets `dwMaxSizeHigh` and `dwMaxSizeLow` as the high and low halves of a single value. The size is an 8-byte quantity to allow for the possibility that disk files may exceed the value of `ULONG_MAX` (4GB). Programs that work only with files of sub-astronomical sizes always set `dwMaxSizeHigh` to 0. Setting both parameters to 0 instructs the system to set the maximum size from the file's current size. If `hFile` is passed as `(HANDLE)0xFFFFFFFF`, however, the size may not be 0; that is, you must set an explicit size in order to work from the system swap file.

The final parameter, `lpzMapName`, gives the new object a name. `lpzMapName` is not the name of a disk file; if there is a disk file, its name is passed to `CreateFile`. `lpzMapName` assigns a name to the new file-mapping object for the convenience of other processes. Processes use the name to share the object, just as they use pipe names to share pipes. Other processes open their handles to the same object by passing the name string to `OpenFileMapping`. The name of a file-mapping object may not exceed `MAX_PATH` characters (currently 260) and may not contain backslashes. (`MAX_PATH` is defined in `WinDef.H`.)

NOTE:

The rule against backslashes in mapped filenames contrasts with the requirements for naming pipes. Pipe names always begin with the sequence `\\.\pipe\` but may contain additional backslashes indicating other subnodes. The rules for naming are different because pipes are implemented as a file system, and mapped files belong to the heap management system.

`CreateFileMapping` returns a valid handle if it succeeds and `NULL` if it fails. If the name string in `lpzMapName` designates a mapping object that already exists, and if the requested protection attributes match, `CreateFileMapping` returns a valid handle to the existing object. That may not be what you want.

A program trying to create a new mapping object may be surprised if another program happens to have used the same name for its own object. To detect whether your new handle belongs to an old object, call `GetLastError` even after `CreateFileMapping` *succeeds* and test for `ERROR_ALREADY_EXISTS`.

Mapping a View

`MapViewOfFile` connects a section of memory with a section of the file. It makes part of the file visible in

memory. A mapped file is “mapped” because of what this function does: It associates every byte of the memory range with a corresponding byte in the file.

```
LPVOID MapViewOfFile(  
    HANDLE hMapObject,          // mapped file to view  
    DWORD fdwAccess,           // access mode  
    DWORD dwOffsetHigh,        // high-order 32 bits of file offset  
    DWORD dwOffsetLow,         // low-order 32 bits of file offset  
    DWORD dwViewSize );        // size of view in bytes
```

The hMapObject handle must be created with CreateFileMapping or OpenFileMapping. The second parameter requests access privileges for the pages within the view. The privileges requested here must not conflict with those already set in the original CreateFileMapping command. For example, a file-mapping object created with the PAGE_READONLY flag will not support a view with FILE_MAP_WRITE access. Many processes may open views to a single file-mapping object. The following are the view access flags:

FILE_MAP_WRITE Grants write access. Requires PAGE_READWRITE.

FILE_MAP_READ Grants read-only access. Requires PAGE_READONLY or PAGE_READWRITE.

FILE_MAP_ALL_ACCESS Synonym for FILE_MAP_WRITE. Requires PAGE_READWRITE.

FILE_MAP_COPY Grants copy-on-write access. Requires PAGE_WRITECOPY.

When you finish with one view and want to inspect another region of the file, unmap the first view and call MapViewOfFile again.

```
BOOL UnmapViewOfFile( LPVOID lpvBaseAddress );
```

lpvBaseAddress is the same value received earlier from MapViewOfFile. UnmapViewOfFile writes any modified pages in the view back to the disk and releases the virtual address space reserved for the mapping. (The FlushViewOfFile command also forces modifications to be saved.)

File-Mapped Object Sharing

In order for two processes to share a file-mapping object, both must acquire a handle. Child processes may inherit file-mapping handles from their parents. If the second process is not a child, and if the name string is not coded into both programs, one process must pass the file-mapping object to the other process through a pipe, a mailslot, a DDE conversation, or by some other arrangement. OpenFileMapping converts a name string into a handle for an existing object.

```
HANDLE OpenFileMapping(  
    DWORD dwAccess,           // access mode  
    BOOL bInheritHandle,      // TRUE for children to inherit handle  
    LPCTSTR lpszName );       // points to name of file-mapping object
```

After receiving its handle, the second process also calls MapViewOfFile to see what the file contains.

Preserving Coherence

If several processes open views on a shared file-mapping object, any changes one makes will be visible to the others. All their view pointers will point to different places in the same coherent object. The file-mapping object coordinates modifications from all of its open views. The file may become incoherent, however, if the views derive from two different concurrent file-mapping objects linked to a single file. If the viewers write their changes to different file-mapping objects, they create conflicting versions of the file.

In Figure 10.6 you can see how two file-mapping objects may contain different versions of the disk file and fall out of sync. Any modifications the first two processes make will be deposited in the first file-mapping object; any modifications made by the third process will be put in the second file-mapping object. If all processes unmap their views and write their changes to the disk, only one set of changes is saved, because one set writes over the other on the disk. When file views become incoherent, he who saves last saves best.

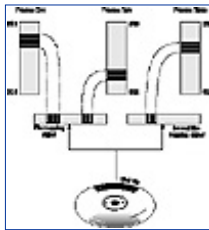


FIGURE 10.6 How deriving views from different file-mapping objects produces incoherence

To enforce coherence, Microsoft recommends that the first program to open the file should specify exclusive access (set 0 in the share-mode parameter). Then no other program can open the file to create a second mapping object.

Three other situations can also produce incoherence:

- Normal I/O performed on a disk file that other processes are viewing as a mapped object causes incoherence. The changes will be lost when the file-mapping object saves its version of the file.
- Processes on different machines may share mapped files, but since they cannot share the same block of physical memory across a network, their views remain coherent only as long as neither process writes to its view.
- If two processes modify a copy-on-write mapped file, they are not writing to the same pages. The modified pages are no longer shared, and the views are not coherent.

Uses of Copy-On-Write Protection

Copy-on-write protection means that when a process tries to write on a page, the system copies the page and writes only to the copy. The original page cannot be modified. In other words, the process has read-only access to the original and gets automatic copies of any parts it decides to change.

Copy-on-write protection can be useful for preserving a buffer's original state until all the changes are final. A half-edited spreadsheet, for example, with all its formulas linking cells, might be useless in its unfinished state. Simply saving changed areas back to the original file could cause problems because the changes in one edited place might invalidate the data in an unedited place. Copy-on-write protection leaves the original buffer and its file intact while editing proceeds. Of course, the problem doesn't arise with unmapped files. In that case, you would need to keep an extra copy of the entire spreadsheet, not just the modified pieces.

As another example, debuggers use copy-on-write protection when altering code to add a breakpoint. If another instance of the process is running outside the debugger, both instances still share the entire source code image except the altered page.

The Browser Demo: Mapping Memory

Each instance of the *Browser* demo opens a handle to the same file-mapping object and maps a view of it. The object represents a small buffer of only 6KB and is backed by the system's paging file.

A multiline edit box in the program window shows the contents of the shared buffer. A set of six radio buttons permits the user to inspect different sections of the shared buffer. When the user selects a new section, the program copies part of the shared buffer into the edit box, where it can be examined and modified. In Figure 10.7, two instances of *Browser* have written information to different parts of the same buffer.



FIGURE 10.7 Two instances of the Mapped Memory File Browser program sharing views of the same mapped file

Pushbuttons issue commands to write from the screen back to the buffer, read from the buffer to refresh the screen, clear text from the edit window, and quit the program. Three more radio buttons set the page protection for the current page to read, read/write, or copy-on-write. If you click the Write button while

viewing a read-only page, the program traps the resulting exception.

NOTE:

The *Browser* demo is included on the CD that accompanies this book, in the Chapter 10 folder.

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Creating and Using the Mapped File

Two constants in the header file direct the program in creating and using the mapped file. The program's window has room for six radio buttons across its width, so we want to divide the buffer into six sections. Each section should roughly fill the buffer, so that as the user switches from section to section, new contents come into view. SECTION_SIZE tells how big one section is (1KB), and NUM_SECTIONS tells how many sections there are (six).

The resource script includes a string to name the file-mapping object. Because all instances use the same name string, all receive handles to the same object. Also in the resources is a dialog template describing the program's main window, which is a dialog box.

Initializing the Mapped File

Browser keeps information about its file-mapping object in two global variables: *hMapFile* is a handle to the object, and *lpView* points to a mapped view of the object.

Because the program's main window is a dialog box, *WinMain* omits the usual message loop. Instead, it simply calls an initialization procedure and invokes a modal dialog box. When the dialog box quits, *WinMain* releases its objects and ends.

InitApp creates the file-mapping object and maps a view of it. Because the instances of *Browser* want only to share memory, not create a disk file, the *CreateFileMapping* command passes (HANDLE)0xFFFFFFFF as the file handle. The buffer will be mapped from the system's paging file. If we wanted to create a new disk file for the *Browser* instances to share, we would need to call *CreateFile* first:

```
// create the file to be mapped
hFile = CreateFile( szFileName, // file name string
    GENERIC_READ | GENERIC_WRITE, // access rights
    0, // don't share files being mapped
    (LPSECURITY_ATTRIBUTES)NULL, // default attributes
    CREATE_ALWAYS, // if it doesn't exist, create it
    FILE_ATTRIBUTE_NORMAL, // no special attributes
```

```
NULL ); // no special attributes
```

The file-mapping object allows read/write access, but individual views may request different protection flags. `PAGE_READWRITE` is compatible with all the possible view access flags.

After creating the file-mapping object, `InitApp` next maps a view of it. Initially, the view allows the user both read and write privileges by specifying `FILE_MAP_ALL_ACCESS`. The next two zeros make the beginning of the mapped view begin with the first byte of the file, and the final zero parameter causes the view to extend to the end of the file.

`CreateFileMapping` sets the file's size to `NUM_SECTIONS * SECTION_SIZE`, which happens to be 6KB. The size and starting point of the view never change in this program. The starting point can't change because it must be a multiple of 64KB. For a 6KB file, the view must begin at byte 0. To show different sections of the 6KB shared buffer, *Browser* uses `lpView` to treat the buffer as an array of characters.

All instances of *Browser* initialize in the same way. If the program mapped its buffer to an independent disk file, the first instance would need to initialize differently. Only the first instance would call `CreateFile`, and subsequent instances could be made to call `OpenFileMapping` rather than repeating the original `CreateFileMapping` command.

```
// get a handle for an existing file-mapping object
hMapFile = OpenFileMapping( FILE_MAP_ALL_ACCESS,
                           // access privileges
                           FALSE,           // inheritable?
                           szMapFile );    // name of object

if( ! hMapFile )
{
    ShowErrorMsg( __LINE__ ); // initialization failed
    return( FALSE );
}
```

Because Windows 2000 always passes `NULL` to `WinMain` for the `hPrevInstance` parameter, *Browser* requires another mechanism to check for other instances. `FindWindow` works for overlapping windows, but standard dialog windows don't have a documented class name. (The class for standard dialog windows seems to be `#32770`, as you can verify with Microsoft's Spy utility, but undocumented features may change in future releases.) Successive instances of *Browser* could also identify their precedence if the first instance created a named pipe or added a string to the global atom table. Each instance could determine whether it has a predecessor by checking for the existence of the signal object.

Running the Dialog Box

The next procedures receive and respond to messages for the dialog box. The *Browser* dialog box initializes its controls when it receives `WM_INITDIALOG`. Among other things, `Browser_DoInitDialog` sets a limit on the number of characters the user may enter in the edit box. The limit prevents the user from entering more text than the current section of the mapped file can hold. `Browser_DoCommand` calls other procedures to manipulate the mapped file view in response to input from the user.

```
void Browser_DoCommand( HWND hDlg,
                       UINT uCmd ) // control ID of button
{
    static int iCurSection = 0;
```

The static variable `iCurSection` records the user's most recent selection from the Section radio buttons.

```
switch( uCmd )
{
    case IDD_WRITE:
        DoWrite( hDlg, iCurSection );
        break;

    case IDD_READ:
        DoRead( hDlg, iCurSection );
```

```

        break;

    case IDD_CLEAR:
        DoClear( hDlg );
        break;

    case IDD_SECTION1:
    case IDD_SECTION2:
    case IDD_SECTION3:
    case IDD_SECTION4:
    case IDD_SECTION5:
    case IDD_SECTION6:
        iCurSection = uCmd - IDD_SECTION1;
                                                // save current choice
        DoClear( hDlg );                      // clear edit box
        DoRead( hDlg, iCurSection );         // show new file section
        break;

    case IDD_READ_ACCESS:
    case IDD_READWRITE_ACCESS:
    case IDD_WRITECOPY_ACCESS:
        ChangeAccess( uCmd );
        break;
    case IDCANCEL:
    case IDD_QUIT:
        EndDialog( hDlg, TRUE );
        break;
    }
    return;
}

```

Modifying the Shared Object

When the user clicks the Write, Read, or Clear button, the program calls DoWrite, DoRead, or DoClear. In response to commands from the Access radio buttons, the program calls ChangeAccess.

DoWrite and DoRead send WM_GETTEXT and WM_SETTEXT messages to the edit control. DoWrite copies the contents of the edit buffer to a section of the mapped file, and DoRead copies a section of the mapped file to the edit control. In both procedures, this expression points to the beginning of the current section of the mapped view:

```
&( lpView[ iSection * SECTION_SIZE ] )
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

lpView is type LPSTR, so it points to an array of characters. iSection tells which radio button is currently checked. The first button is 0, so the result of iSection*SECTION_SIZE is the number of the first byte in the given file section. lpView[] evaluates to a character, and &(lpView[]) gives the address of that character.

```
void DoWrite( HWND hDlg, int iSection )
{
    // get the edit control handle
    HWND hwndEdit = GetDlgItem( hDlg, IDD_EDIT );

    if( hwndEdit )
    {
        try
        {
            // copy text from edit buffer to mapped file
            Edit_GetText( hwndEdit,           // edit control
                        &( lpView[iSection*SECTION_SIZE] ),
                        // pointer into view
                        SECTION_SIZE );      // bytes in section
        }
    }
}
```

The exception handler in DoWrite is designed to catch the error when the user write-protects the mapped view and then tries to use the Write button. We could disable the button, but allowing the error to occur seemed more instructive and a better demonstration of what write protection does.

```
except( EXCEPTION_ACCESS_VIOLATION )
{
    // assume exception was due to read-only access
    MessageBox( hDlg, "Memory is read only.\n\r"
                "No data was written.",
                "Browser Message",
                MB_OK | MB_ICONEXCLAMATION );
}
}
```

```

    return;
}

```

The DoRead function reads text from the mapped file, copying the retrieved text to the edit control buffer.

```

void DoRead( HWND hDlg, int iSection )
{
    HWND hwndEdit;

    // get the edit control handle
    hwndEdit = GetDlgItem( hDlg, IDD_EDIT );
    if( hwndEdit )
    {
        // retrieve the text from memory and display it
        Edit_SetText( hwndEdit,
                      &( lpView[iSection*SECTION_SIZE] ) );
        // pointer in view
    }
    return;
}

```

On the other hand, the DoClear procedure does nothing to the mapped file, but it does clear the text in the edit buffer.

```

void DoClear( HWND hwnd )
{
    HWND hwndEdit;
    char *szText = "\0";
    // empty the edit buffer by filling it with a null string
    hwndEdit = GetDlgItem( hwnd, IDD_EDIT );
    if( hwndEdit )
    {
        Edit_SetText( hwndEdit, szText );
    }
    return;
}

```

Last, ChangeAccess unmaps the current view of the file-mapping object in order to remap it with a different access privilege. Every instance of the program may choose different access flags because each has its own independent view of the object. An instance that uses FILE_MAP_COPY remains synchronized with the other instances' views until it writes to the buffer. At that moment, the system intervenes and creates a copy of the protected page for the program to modify. From then on, the program never sees the original unmodified page again, even though the other instances do.

```

void ChangeAccess( int iNewAccess ) // requested access rights
{
    DWORD fdwAccess;           // new access flag
    UnmapViewOfFile( lpView ); // close the previous mapping
    switch( iNewAccess )        // choose new access flag
    {
        case IDD_READWRITE_ACCESS:
            fdwAccess = FILE_MAP_ALL_ACCESS;
            break;
        case IDD_READ_ACCESS:
            fdwAccess = FILE_MAP_READ;
            break;
        default: // IDD_WRITECOPY_ACCESS
            fdwAccess = FILE_MAP_COPY;
            break;
    }
    // remap the view using the requested access and view number
}

```

```

lpview = MapViewOfFile( Hmapfile, // handle to mapping object
                        fdwAccess, // access privileges
                        0, 0, // starting offset in file
                        // (low order 32-bits, high order 32-bits)
                        0 ); // view area size (all of file)
if( lpView == NULL ) // error?
{
    ShowErrorMsg( __LINE__ ); // yes; tell user
}
return;
}

```

Since *Browser* shows the buffer in 1KB sections, four sections fill one page frame. If one instance writes to the first section with copy-on-write protection, it loses synchronization in the first four sections (one page frame). Its view of sections 5 and 6, however, will continue to reflect changes made by other instances. Also, changes the first program makes to its copy-on-write buffer will never be visible to other instances. The copied pages are private.

Summary

In this chapter, you've seen some of the power of the Windows 2000 VMM. The virtual memory API passes on to you some of these powers, especially the ability to reserve a range of addresses without immediately committing physical memory to support them. The ability to guarantee that a range of contiguous addresses will be available as an object grows, or as an array slowly fills in, makes working with sparse memory structures easy.

Given the VMM's effectiveness in isolating each process from memory used by any other process, sharing blocks of memory becomes problematic. Programs have moved apart into private address spaces. Gone are the neighborly days of passing memory handles from program to program. In their place are the high-tech miracles of memory-mapped files, and with them come all the modern concerns for protection, coherence, and point of view. The *Browser* demo discussed in this chapter demonstrates how to address some of these concerns.

In the next chapter, we'll discuss another concern of modern applications: security.

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 11

Security and Cryptography

- Comparing the security support of Windows NT/2000 with that of Windows 95/98
- NT security and the security API
- Data encryption and the Crypto API
- Other considerations for securing data

In this chapter we will set out by discussing the level of security functionality provided to programmers by Windows NT/2000, beginning with an in-depth look at the Security API, which is one of the features of Windows NT/2000.

Understanding how the security model works and how it can be incorporated into your own software is becoming increasingly apropos, as ever more data is placed online. By understanding how NT implements security for itself and makes this same level of security available for writing server applications, you will also have a better understanding of the types of issues that need to be considered when designing and writing good client applications that need to access secure data.

Next, we will look at an area of security that is identically available both to Windows NT/2000 and to Windows 98: data encryption and authentication, as provided by the Crypto API. Data authentication and confidentiality, online financial transactions, and protection from rogue software are reasons enough to enlist the help of cryptography, and the Crypto API provides a standard way to access encryption, hashing, signing, and authenticating algorithms, in a modular fashion, from a variety of different security vendors.

Finally, we'll wrap up the chapter by looking at some further points of consideration, which relate generally to the implementing of security.

Comparing Windows NT/2000 with Windows 95/98 Security Support

While Windows NT/2000 has always provided a relatively high level of security support, beginning with Windows NT 3.1, Windows 95 offered very little. With the introduction of Windows 98, a number of

security-related features were added and/or improved upon from Windows 95. These changes, in a nutshell, are as follows:

- Support for Secure Channels
- Support for Smart Cards
- Built-in Crypto API Support
- Built-in Authenticode Support
- Built-in Microsoft Wallet Support

All of these, of course, had their debuts under Windows NT.

Secure Channels support includes Point to Point Tunneling Protocol (PPTP), which enables users to connect securely to a remote network—if necessary, even over an intervening insecure network. This is accomplished by means of encrypted encapsulated packets, and enables one protocol to be nested inside another. Thus a user can connect to the Internet via TCP/IP, for example, then establish a secure IPX connection to his or her office network. Support for Secure Socket Layer (SSL) has been upgraded to SSL 3.0, which increases the level of encryption that can be applied to Internet and intranet data exchanges.

Smart card support consists of a two-layer driver model designed to encompass smart card reader hardware, together with the APIs used to authenticate, write to, and retrieve data from smart cards. These cards will have at least three important uses: as user authentication in place of (or in addition to) logon sequences, for transacting financial business over the Internet and elsewhere, and as portable data repositories, for storing bits of information such as one's drug allergies or dental history.

The last three items first appeared as part of Microsoft Internet Explorer (IE), and are still being included with the various platform versions of IE.

TABLE 11.1: How Windows NT/2000 Security Differs from Windows 95/98 Security

Technology	Windows 95	Windows 98	Windows NT/2000
Authenticode	ADD-ON (IE4/5)	YES	YES
PPTP client	ADD-ON	YES	YES
PPTP server	NO	YES	YES
Smart cards	ADD-ON	YES	YES
Crypto API	ADD-ON (IE4/5)	YES	YES
Microsoft Wallet	ADD-ON (IE4/5)	YES	YES
Group-level security	PARTIAL	PARTIAL	YES
File-level security	NO	NO	YES
Object rights and privileges	NO	NO	YES

As you can see in Table 11.1, NT/2000 and Windows 98 share many of the same security technologies, particularly the Internet-related security technologies. The chief way they differ is in Windows NT/2000's absolute, "from the ground up," internal security features that incorporate myriad low-level security checks at every point where objects can be accessed within the operating system. A good example of this is in file access. In Windows 95, and now Windows 98, security is on only a folder and drive level—you cannot set individual access rights at the file level. In part this is because there are many different means of getting at a file in Windows 95/98: various DOS interrupts, BIOS functions, Windows APIs, various language file access APIs (such as Pascal and C++), and so on. There are many resulting security loopholes that would require extensive patching to fix, and would likely break many existing applications in the process. (Keep in mind that much of this code was "adapted" from good ol' 16-bit code.) Then too, there is the overhead that such checks would add to file access. Such compromises were understandably deemed undesirable for an operating system that is marketed more to home users than to business users, who have a greater need for security. However, as the Internet keeps enlarging to subsume more of our daily affairs, such security will become increasingly necessary even for home users, who have been flocking enthusiastically to the Internet.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

What this difference means to programmers is primarily what it meant with Windows 95/98: to create truly secure applications requires the use of Windows NT/2000, although applications that make use of encryption can still provide an effective level of access control in certain instances, if implemented carefully. However, it is essential to keep in mind when contemplating incorporating security features into your software that the client-server or multitier approaches have even greater benefits when looked at from the vantage point of implementing security. How this works is covered in detail in the next section.

In short, these differences between Windows 95/98 and NT/2000 center primarily around the thorough way that Windows NT/2000 internally checks access rights, and in the security hooks exposed at the programming level that allow programmers to build the same level of robust security into their own applications. This security technology is exposed by what is referred to as the Security API—an API which, while considered part of the Win32 API (or of late, the “Win Base API”), is fully implemented only in Windows NT/2000.

In this section, we will look at how NT/2000’s security functions work, with respect to permitting and restricting access to secured objects. As you will see, performing what would appear to be a simple and straightforward operation can often require the use of multiple, rather complex functions and data structures.

Security and Encryption: Windows NT/2000 versus Windows 95/98

In case you’ve been under a rock somewhere the last few years, beware of the fact that very few of the security API functions are implemented in Windows 95/98. This should come as no surprise, because one of the main advantages Windows NT/2000 has over Windows 95/98 is its extensive built-in security. Your application may *call* any one of these functions while running on a Windows 95/98 workstation, but the program must be prepared to check the error-return code, to note that the function failed, and to take appropriate action.

Remember, though, that the only correct way to use the NT security API functions is in the context of a server application. While it is possible to design a server application to run on a Windows 95/98 workstation, it would not be possible to do this where the server application needs to implement NT/2000-level security using the Win32 security API. And such would seem to be the case for most, if not all, server applications. Thus, the better and simpler approach is to *require* the application to be run on NT/2000, and to simply issue a complaint message and exit if someone tries to run the app on a Windows 95/98 workstation.

When demanding platform compliance, however, simply performing a check against the OS version and

exiting if the version number doesn't match is not the appropriate approach. For example, some older versions of Windows applications—when executed on the latest version of Windows NT or Windows 2000—have exhibited confusion, complaining that they require a *newer* version of the operating system. In the same vein, we've seen the OS installation complain that the existing version of Internet Explorer (installed as part of Visual Studio) is incorrect and should be uninstalled before proceeding (and remember, Microsoft has testified in court that IE cannot be uninstalled).

Thus, rather than simply checking the version number and exiting if a mismatch is found, another approach is required.

One solution could be to check the version number to ensure that it is *greater than or equal* to a specific version, rather than merely *equal* to it.

A better approach, however, would be to look at those categories of functionality required by the application that may not be supported by OSs that are otherwise fully capable of running the application. Then, at runtime, have the application perform checks for functionality and, if the supporting functionality is missing, either exit (with an explanation and suggested corrective actions) or modify its own behavior/functionality accordingly.

On the other hand, you should be pleased (or relieved) to know that Windows 98 does provide full support for encryption and decryption (via the Crypto API) and the Crypto.exe demo program discussed later in this chapter will function normally on both Windows NT/2000 and Windows 98 platforms. This assurance aside, however, you are still advised to verify that your applications actually do perform as expected on all platforms.

NT/2000 Security

There are a large number of API functions currently pertaining to security in Windows NT, not including the Cryptography API (Crypto API).

NOTE:

Any security functions supported by Windows NT/2000 but not supported by Windows 95/98 will find function stubs in the Win32 API, thus allowing NT/2000 applications to load on Windows 95/98 without errors. As the programmer, however, you are responsible for deciding what functionality an application should supply in the absence of security support or even whether the application should be permitted to execute without full security functionality.

But what sorts of applications really need to be concerned with NT/2000 security? Simply put, anything intended to run on a *server* is a strong candidate for needing to implement security. Additionally, at the client level, any applications needing to access their own secured objects (shared memory, files, semaphores, the registry, and so on) and most applications that will run as a *service* will need to make use of at least some NT security functions.

Planning for Security

When thinking about NT/2000's security, one important point you should keep in mind is the distinction between automatically protected objects in NT/2000 and those objects you create yourself and must protect by implementing your own security checks. An example of automatic security would be the security checking that takes place within the kernel whenever access attempts occur on files or directories that are located on an NTFS partition. On the other hand, "manual" or "do it yourself" security would be called for if you needed to prevent (for example) unauthorized access to only certain types of functionality within an application, or to certain parts of a database file.

Preparing for security involves four main steps:

1. At the planning stage, create a list of all data objects and/or types of functionality that should not be open to all users on an indiscriminate basis.
2. Catalog the specific rights or privileges that can be assigned or restricted to individuals or groups using the system/application.
3. Add security checks at every point in your software where the protected objects or functionality can be accessed, or build security checks directly into object classes themselves.
4. If any objects can be assessed in any way outside of the application—such as by standard file access,

for example—access also needs to be restricted here. Commonly this would be done by specifying Administrator-only access for the object and then have the application run using Administrator privileges, thus granting access to the application.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Users and Objects

The first thing to get clear about NT/2000 security is that it is based on a per-user security model. Whenever a user successfully logs on to a trusted server, that user is identified internally by an access token. Everything that user then does, from starting applications to accessing printers and drives to opening and saving files, is allowed or not allowed based upon what rights and privileges are granted that user.

In addition to determining who has access to what, NT/2000's security API also provides system-level and application-level event logging features, so you can later determine who *did* have access to what, and when.

If you look for a moment at Figure 11.1, you see a simplified representation of how NT/2000's security system keeps track of *users* versus *objects*. As each user logs in and is validated by an NT system, that user is given a unique access token that sticks with them, so to speak, as long as they are logged in, wherever they go within the NT/2000 network. Each system object, on the other hand, is represented by means of a security descriptor (SD), which holds a number of pieces of security-related information, including who the owner and/or primary group are. The SD can also hold zero, one, or more entries in a system access control list (SACL) and/or a discretionary (owner) access control list (DACL). We'll discuss each of these pieces in more detail later.



FIGURE 11.1 Whenever a user attempts to access an object, NT/2000's security compares that user's access token with the permissions in the security descriptor for that object.

System and User Objects

An NT/2000 system object, in the context used in this chapter, does not mean a C++ object, an OLE object, or a COM object, but rather is any object created and managed by one of the three main NT/2000 subsystems:

kernel, user, or GDI. For various reasons, NT/2000's built-in security is implemented only on kernel objects. Thus you are not required (via the API) to worry about security when creating bitmaps, cursors, pens, brushes, icons, metafiles, or window handles, for example. But you do need to provide at least a security descriptor parameter (or a NULL value) when creating or managing any kernel object, and this includes any file, folder, storage device, pipe, mailslot, serial port, registry key, console device, or printer queue, among other things.

In addition to system objects, there are also user- or application-defined objects. These are any objects (loosely understood) that your application defines, creates, and recognizes. They can include particular functionality in your application, NT objects from the USER and GDI subsystems (if you need to protect a bitmap, for instance), subdivided areas of files, or anything else your application must interact with and control access to—whether it be a piece of data, a menu command, a particular view of a dataset, or whatever else you might come up with.

Sometimes, a further distinction is made between NT/2000 objects created by NT/2000 and NT/2000 objects created by the end user via their use of the system or various applications. This difference becomes important in understanding the difference between SACLs and DACLs, as we'll point out later.

Built-In Security versus Application Security

Whereas NT/2000 internally and automatically handles the storage and checking of security settings on its own kernel system objects, any other objects you define and create are not protected automatically, so if you want/need security restrictions enforced on your objects, then you will need to implement this security yourself.

Let me repeat this: Anything not covered by NT/2000's built-in/automatic security to which you wish to add access restrictions (security) requires that you specify the desired permissions and then check those permissions every time an attempt is made to access your object. NT/2000's security API does give you some help in this regard, as its set of functions are the identical functions used by NT/2000's own internal security. Therefore, the process you go through to secure your own objects has much in common with what NT/2000 does to provide security for its objects.

Of course, if you're creating some object that, in turn, makes use of one or more system objects, then NT/2000 may automatically provide *some* level of security for these system objects. If you store your object in a file, for example, that file will be subject to NT/2000's built-in file security just like any other file (if it's located on an NT/2000FS partition, that is).

WARNING:

File access security depends on the use of an NTFS volume and is not implemented on FAT/FAT32 file systems. See Chapter 3, "The NTFS File System," for further information.

This overlap of NT/2000's system security with additional security that you may implement is not only welcome, but is also absolutely essential to the proper implementation of security in your application. For example, NT/2000 itself knows how to restrict access to files, subdirectories, and drives. But it doesn't know anything, for example, about the fact that part of your database file contains data that should be available only to certain users, while other parts may be appropriate for general access.

In this particular case, your database file would actually need to be protected in two ways. First, you would have to use NTFS's file security to permit direct file access only by system (or database) administrators. Second, you would have to write your database server application such that it takes all the incoming data requests from various users and provides the desired additional level of security based on permissions you have previously defined at the user or group levels.

This server application, by the way, would itself be able to access the database file directly, because it would run using the Administrator (or other appropriate) access rights.

NOTE:

When implementing security on any object that could get accessed outside of your application, always restrict any and all *direct* access (file, communications, and so on) to this object by also using NT/2000's built-in security attributes, wherever applicable. Typically, this means setting *Administrator-only* access on all "outside" approaches to your object, then having your server application internally make further decisions about granting or denying access on a case-by-case basis.

As you may surmise then, it is generally foolhardy to attempt to implement security from within your application, when either the application or its data files will end up being stored on a FAT (or other non-NTFS) partition. Of course you could partially get around this by using encryption on your data files, but still there are likely to be security loopholes if users are able to directly access your application's data in any way.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Security Is for Servers

When planning your security implementation, *always* design using the client-server model! In other words, do not use the security API just anywhere in your Win32 applications, but only in software that will be run on a server. Split the project into a client application, which individual users will use, and a server application, which runs on a server and handles all incoming requests for secured objects.

Why insist on this extra work? Well, there are several reasons:

- First of all, the software running on the client side might end up being run under Windows 98, or another operating system that doesn't support security—perhaps a Web browser, for example.
- Second, this clean break between a server app (which handles all security and therefore “clears” all your secured-object requests) and the client app(s) (which makes use of the objects) centralizes the security issues to one machine rather than several or many, leaving that many fewer potential security loopholes.
- Third, this separation of function provides you with much greater flexibility down the road, in terms of adding new types of client applications or extending the size of your LAN, without the hassle of needing to change your server implementation (which would thereby reopen a can of security worms, so to speak).

NOTE:

Important! *Always* use the *client-server* model when designing systems needing NT/2000's security services. The security checking should always (and only!) be performed by an application functioning on a server, or in a server capacity. A client application running on the user's system should use any of the standard network communications means to request access to the secured objects from the server application. Aside from other good design reasons, this technique allows you to combine secure access with the ability to access the objects via workstations that do not internally support NT/2000 security (Windows 98, for instance, or perhaps a Java applet running in somebody's Web browser on a Unix or a Mac machine).

So how is the client application supposed to communicate with the server application? There is more than one right answer to this; however, the method you choose must be a secure one. Typically, any of the following means of communicating between client and server applications can be used:

- Named pipes

- Remote procedure calls (RPC)
- Distributed COM (or DCOM)
- Secure sockets (and supporting Web server)

These are not the only means you could use, of course. In most cases, any one or any combination of the above are excellent choices.

Structures and Terminology

You cannot do much programming in the Win32 API without running into the many kernel object manipulation functions, which always take a pointer to a SECURITY_ATTRIBUTES structure. Take CreateFile, for instance, which is defined as follows:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,           // pointer to name of the file
    DWORD   dwDesiredAccess,      // access (read-write) mode
    DWORD   dwShareMode,         // share mode
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // (see below)
    DWORD   dwCreationDisposition, // create mode
    DWORD   dwFlagsAndAttributes,  // open, delete, close attributes
    HANDLE  hTemplateFile );      // optional source for attributes
```

What? All this, just to create a file? That was my first reaction as I came across this in the Software Development Kit (SDK) documentation. There is a strong tendency, I am sure, for programmers to take one look at this and decide to go back to using their favorite “normal” means of opening files: fopen, OpenFile, or Delphi’s Reset, for example.

As it turns out, CreateFile has far more uses than its name implies, as it’s the primary means in Win32 of opening all sorts of NT/2000 objects in addition to files: pipes, mailslots, communications sessions, disk devices, you name it. In addition, CreateFile is not only useful for creating these objects, but can also be used for reopening, closing, and even deleting files when finished, given the right combinations of flags. CreateFile is also capable of doing both synchronous and asynchronous (overlapped) I/O operations. Obviously, performing any one of these operations on a secured object causes NT/2000’s built-in security checking to go to work, comparing access rights contained in your lpSecurityAttributes structure with any permissions or restrictions that have been placed on the object, and causing your function to succeed or fail, as appropriate.

Typically, you would most often use CreateFile by simply filling in a NULL value for the lpSecurityAttributes. This is fine for routine object access, because when you specify the NULL value, the default set of SECURITY_ATTRIBUTES for the current process is used, which in most cases is exactly appropriate. By the way, this is also exactly what happens when you open files using fopen, OpenFile, or any other conceivable means of file access under NT/2000. As you might guess, all these functions behave similarly because in NT/2000 they all trace right back to the same code (and security checking) used by CreateFile itself.

Obviously, if we want to employ anything more sophisticated than the default security behavior by using a NULL value for our SECURITY_ATTRIBUTES structure, we are going to have to supply an actual SECURITY_ATTRIBUTES structure. To do this, roll up your sleeves and prepare to do some rather strenuous mental exertion.

Security Attributes

The SECURITY_ATTRIBUTES structure looks benign and simplistic at first glance:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD   nLength;           // use sizeof(SEcurity_ATTRIBUTES)
    LPVOID  lpSecurityDescriptor; // “complex” structure
    BOOL    bInheritHandle;    // whether child process inherits
} SECURITY_ATTRIBUTES;
```

Setting the nLength field is simple (and important!). Setting the bInheritHandle flag is straightforward enough—after all, it’s a Boolean value. However, that middle field, which is actually a pointer to a SECURITY_DESCRIPTOR, is where the rubber really hits the road (in a security sense). The security descriptor (SD) is one of four *major* data structures used by NT/2000’s security API. (The

SECURITY_ATTRIBUTES structure doesn't count as "major," since its purpose is only to be a wrapper for supplying the SD to NT/2000.) If you've noted the earlier reference to (and diagram of) access tokens and SDs in this chapter, you will recall that, whereas access tokens represent *users* of objects in NT/2000, the SD represents an object or a group of objects to NT/2000's security system. It makes sense, then, that the various Win32 API functions dealing with securable objects each takes a SECURITY_DESCRIPTOR, by encapsulation in the SECURITY_ATTRIBUTES structure.

In the following sections, we'll look at each of the four main types of security structures, including the SECURITY_DESCRIPTOR, and then discuss how each is created, initialized, and manipulated using the functions supplied by NT/2000's security API.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Four Major NT/2000 Security Structures

Although these are not the only security-related information structures, the following four structures are certainly the “heavy hitters” used for keeping track of NT/2000 security-related information:

- Access tokens
- Security Descriptors (SDs)
- Security Identifiers (SIDs)
- Access Control Lists (ACLs)—also known as system access control lists (SACLs) and discretionary (owner) access control lists (DACLS)

One thing you should initially note about these structures is that, although they are structures, you’re required to consider them as *opaque*, meaning initializing them and setting their values must be accomplished by means of the supplied appropriate API functions. Exactly which functions should be used for manipulating each structure is one thing we’ll look at next. (In case you’re wondering, I have no idea why access tokens don’t get commonly abbreviated as their peers do. Maybe we should just start our own trend and begin calling them ATs. I won’t though, at least in this chapter.)

NOTE:

As the SDK points out, you should never try to manipulate these security structures directly. There is a whole assemblage of API functions available for you to use to initialize these structures and politely “modify their guts.”

Let’s look at each of these four structures now in greater detail.

Access Tokens

Each user logged onto an NT/2000 system is assigned an access token. This token is checked against SDs of any objects that the user is trying to access. Note that an access token contains user privileges, while an SD contains access rights. If you’re wondering what the difference is between rights and privileges, please see the sidebar entitled “Rights versus Privileges in NT/2000” for an explanation.

Rights versus Privileges in NT/2000

What's the difference between rights and privileges? In NT/2000's view of things, a *right* is the ability given to an individual user or a group of users to access some given object in a particular way. Examples of rights include giving read-only (or read-and-write) access to a file or folder or to a shared CD-ROM drive, or giving access to print to some printer on the network. Because rights are attached to objects (but in relation to users), they are stored in the security descriptor (SD) for an object.

Privileges, on the other hand, are the predefined abilities to perform particular operations on the system. Examples of system privileges include being able to shut down or start system services, install or remove system device drivers, or perform backups. Unlike rights, privileges are stored in the user's access token, primarily because most often a privilege will override otherwise assigned rights. It turns out that it's much more practical to have this information attached simply to the user than it would be to update the settings of each object the privilege would affect. Take a typical example: performing system backups. Here, the user must be able to access most or all of the system files in order to perform the backup. It is simply not feasible to try to maintain an exceptions list attached to each file that gets created, in order to keep track of who now has backup privileges.

Security Descriptors (SDs)

Inside the SDs are the collected pieces of information that together exactly specify the owner, permitted users and groups, and permissions granted (or denied) to particular users or groups of users of that object.

Subelements of the SDs include the owner and group SIDs, as well as two ACLs. The ACLs can each contain zero, one, or more access control entries (ACEs). One ACL is actually an SACL and the other is a DACL, and these hold any system-assigned restrictions/permissions and owner-assigned restrictions/permissions, respectively.

Again, NT/2000's security API provides an entire array of functions for querying and manipulating the various security structures. Note that since these are structures, rather than object-oriented programming (OOP)-type objects, the way you interact with them is by using appropriate API functions that expect as a parameter a pointer to a particular type of security structure.

Self-Relative versus Absolute SDs Security Descriptors can be represented in either self-relative or absolute format. If you look at Figure 11.2 you can see an illustration of how these two formats differ.



FIGURE 11.2 Security descriptors come in two flavors: *absolute*, which is useful for updating the SD's fields, and *self-relative*, which is better suited for transferring the SD to a file, a different process, or across the network.

The reason for two formats is that there are benefits to storing the data in each way. On the one hand, these SDs contain fields (the ACLs) that contain a variable number of entries (ACEs), and each field could be either NULL or empty, rather than contain information. Therefore, it makes the job of updating (adding and removing) entries and settings in an SD much simpler if the SD itself is stored in memory as a structure of pointers (to structures with other pointers, usually). Otherwise, each time anything changed, a larger chunk of memory would need to be allocated, and the entire structure would need to be copied to the new area along with the new changes. This "update-friendly" format that makes use of pointers is called (somewhat confusingly) the *absolute* format.

On the other hand, because pointers point to RAM, and RAM is generally accessible only by the computer owning it, it would not work very well to send an SD across the network (or save it to disk) since you'd only be sending (or writing) a set of pointers, rather than the various structures themselves. Therefore, the SD can be stored in a second format where the entire structure, including all its current fields, is stored in one

contiguous block of memory. This more “transfer-friendly” format is called the *self-relative* format. As you might surmise, this is the format in which SDs are returned to you when you request them from the system. You can look at (read) the SD in this format, but when you need to change (write to) the SD in any way, you must first convert the SD into absolute format.

Fortunately, there are two API functions you can use to do this conversion: `MakeAbsoluteSD` and `MakeSelfRelativeSD`. Unfortunately, you will need to use them each and every time you get or put an SD or need to make changes to it.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Security Identifiers (SIDs)

A SID is a structured identifier of variable length that uniquely refers to a user or a group of users. Internally, a SID contains (among other things) a 48-bit authority value, which is a unique value comprised of a revision level; an identifier-authority value; and a subauthority value. Once a SID is created, an NT/2000 system will not permit it to ever be reused, even if the SID is deleted. The Security Descriptor of each object contains two SIDs: an owner SID and a primary group SID. The owner SID identifies the current owner of the object, and the group SID identifies that group of users that have specific access rights or restrictions for the given object.

Note that SIDs are used in several other areas of NT/2000 security. Every access token, for example, contains a SID that identifies the groups to which that user belongs. Also, each ACE in an ACL has its own SID that specifies the individual or group being granted or denied a set of permissions or rights.

Access Control Lists (ACLs)

ACLs are lists of rights or restrictions placed on users or groups for a given object (or group of objects). Each SD can contain both a SACL and a DACL. These are identical in form, except that the SACL keeps track of permissions set on an object at the system level, and the DACL keeps track of permissions set by the owner of the object. The SACL obviously can be changed only by users who possess system-level access, whereas the DACL can be changed by whoever is currently specified as the object's owner. I say "currently" because one of the rights that can be set on an object is the ability to change who the current owner is.

Both types of ACLs can contain zero, one, or more ACEs. Each ACE contains three parts: users or groups this ACE pertains to, which rights are affected, and whether the rights are being granted or revoked.

An ACL can be in one of three states. It can be empty, it can be NULL, or it can contain one or more ACEs. In an ACL's initial state, it is empty, which is in effect the same as saying that no access rights have been given for this object, so no users will have access to it. An ACL that is set explicitly to NULL, on the other hand, means *no* security has been assigned to the object and therefore *all* access is granted.

Once you add a single ACE to a previously NULL ACL, the object loses its "general access" (or lack of security), and is available only to those users or groups specified by an ACE.

NOTE:

Do not confuse a NULL DACL with an empty DACL! They are actually opposites in their effect. An empty DACL (the initial state in a new SD) means no *access* has been granted. A NULL DACL means no *restrictions* exist for the object, so it is accessible to all users. A DACL only becomes NULL by being set that way explicitly.

So, you should now understand (at least in theory) how an object's access is specified for any condition: complete access, no access, access granted to some, and access restricted to some.

Client Impersonation

The procedure whereby a server application checks that a connected client actually has access rights to a requested object is called *client impersonation*. What this means is that the client application, when making an object request of the server application, passes along its access token. The server application can then use that access token to log on as the client or impersonate the client, attempting to access the desired object in the same manner that the client is asking for. It is true that the server application could instead look up the security information for the client, then get the security information for the asked-for object, and see whether the client has the desired permissions that way, but impersonation is the simpler and more foolproof method of doing this.

Here are the steps a server application might typically make to use client impersonation in implementing server-side security:

1. ClientApp establishes a connection with the ServerApp.
2. ClientApp asks the ServerApp for access to a particular object.
3. ServerApp uses `ImpersonateLoggedOnUser` to obtain an impersonated access token, which it then uses to attempt access in the manner ClientApp is asking for.
4. ServerApp reverts to its own access level, by making a call to `RevertToSelf`, so that it can be available to handle other clients' requests.
5. If ServerApp's access attempt succeeded, ServerApp passes back to the client either an actual handle to the requested object, or else some other prearranged proxy token by which the ClientApp can refer to the requested object in future requests to the ServerApp.
6. Each time the ClientApp wants access (again) to a secured object, ServerApp should call `ImpersonateLoggedOnUser` again before accessing the object, to ensure that ClientApp access rights haven't changed during the interim. When the request is handled, ServerApp again calls `RevertToSelf`.
7. When ClientApp disconnects from ServerApp, the ServerApp can delete the impersonated access token.

Note that for private objects, the server and client must establish some consistent means of referring to desired objects. This could be done using either a handle or token-type that the server assigns and keeps track of, or it may perhaps be a string or whatever other protocol the server and client applications implement to refer to server-app managed objects. Also note that for private objects whose security is being handled by the server application (as opposed to some other server application running on the same or another machine), it is still important to use `ImpersonateLoggedOnUser`. In the course of handling the client's request, the server may need to open or access objects for which NT/2000 does have some level of built-in security assigned. By using impersonation through the entire request-handling section, the server application ensures that it won't inadvertently do something on the client's behalf that is explicitly restricted from that user. Also note that individual server threads (or processes) can each have a different impersonation access token assigned, which is of course necessary for permitting simultaneous services to multiple clients.

Checking Access Rights

Whenever an attempt is being made to access an NT/2000-secured object, NT/2000 internally calls a function called `AccessCheck`, which compares the user's access token with the access rights contained in the requested object's SD. This either succeeds or fails, and in the case of failure, the function that made the access request returns an `ERROR_ACCESS_DENIED` error (traditionally called an "error #5," but always use the constant).

In an identical way, server applications need to call `AccessCheck` whenever and wherever access to a server-app-protected object is being attempted. If you leave even one place in your code where the server can access an object on behalf of a client without calling `AccessCheck` (and heeding the results!), your server application's security can be compromised, so be careful! It greatly simplifies your implementation if you can make all access to a server-app-protected object occur in the same single place in your code.

There is a similar function called `PrivilegeCheck` that is useful for checking for required privileges. Also, before

calling the checking functions, you may need to create a rights-mask containing the appropriate bits set for the requested access rights.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Adding New Rights

In addition to the built-in or predefined system rights that appear in the User Manager (or User Manager for Domains), you can create new rights of various types, should your application require it. There are three types of rights, and it is important to understand the differences between them.

Before explaining the differences, let me point out that rights are stored, as you might expect, in a bit field, so individual rights are checked using bit masks and constants.

Standard rights Rights where the assigned bit means the same thing, or approximately the same thing, across any objects that use them.

Specific rights Rights that could use the same bit values in different objects, but that mean different things for different objects. This is the type of right you would use to define some completely new right, which has little in common with already existing (or predefined) rights.

Generic rights Rights with very broad meanings that can be interpreted somewhat differently by different applications. Generic rights can be mapped to specific and standard rights, where there is some similarity of meaning between the name of the general right and the functionality being mapped to it.

NT/2000's Security API

Rather than starting out taking a subset of the security functions and trying to discuss them abstracted from all the others, I've put together Table 11.2, which contains all known (current) security functions, broken down into various sections or groups, according to their common usage. Between each group I'll point out what I think are important issues to be aware of in that function group. Note that in at least a few cases, functions may appear in more than one category, as is appropriate.

Also, I somewhat regret not including each function's prototype (parameters) in the table, but this seemed to me to add more clutter than pertinent information here. My primary purpose is to get you up to speed quickly with an overview of all the security functions, and to more easily convey to you the scope that these functions cover, and how they interrelate. The online help included with your compiler provides a comprehensive, alphabetized list of all these functions, with corresponding prototypes just a click away. (It does not, however, include categorized groupings of the security functions, which is why you may find the following table to be

of help.)

I do think the security functions are one of the more complex areas of NT/2000 programming, particularly due to the non-object-oriented nature of the implementation. Just creating one of the needed structures is usually a multistep process. For performance reasons, I can understand why it was implemented this way, but you may find that writing an object wrapper for these functions is far preferable to having to deal with allocating, deallocating, initializing, recursing, changing, converting back and forth, and updating these structures.

Security is an important area, and it should not be implemented less often simply because it's less trouble to do so. The best strategy I can suggest for dealing with these security functions is to work through them at least once and learn how they work, then either cut and paste from existing, working code as needed, or (better yet) wrap up the relative complexity in a tidy object wrapper.

TABLE 11.2: Access Token Functions

Access Token Name	Function
AdjustTokenGroups	Adjusts the groups in an access token
AdjustTokenPrivileges	Adjusts privileges for an access token
CheckTokenMembership	Determines whether a specified SID is enabled in an access token
CreateRestrictedToken	Creates a new access token which is a restricted version of an existing access token
DuplicateToken	Creates a new access token identical to one supplied
DuplicateTokenEx	Creates a new access token duplicating an existing token
GetTokenInformation	Returns user, group, privileges, and other information about a token
IsTokenRestricted	Reports whether a token contains a list of restricting SIDs
OpenProcessToken	Retrieves the access token for a given process
OpenThreadToken	Retrieves the access token for a given thread
SetThreadToken	Assigns an impersonation token to a given thread
SetTokenInformation	Modifies user, group, privileges, or other information within a given access token

You retrieve a thread's or a process's access token via `OpenProcessToken` or `OpenThreadToken` (respectively). Note that `AdjustTokenGroups` and `AdjustTokenPrivileges` are the obvious "power functions" in this group. One good use for the `DuplicateToken` functions is when you might need to back out (cancel) changes that were just made to an access token. Simply create a copy before making the changes, and then if need be, you can use the copy as your backup to restore the token's previous settings.

Access Token Name	Impersonation Function
CreateProcessAsUser	Identical to <code>CreateProcess</code> but creates the process for a given access token
DdeImpersonateClient	Allows a DDE server to impersonate a DDE Client application, to make system accesses using the client's security
ImpersonateLoggedOnUser	Lets a calling thread impersonate a given user (takes an access token)
ImpersonateNamedPipeClient	Allows the server end of a named pipe to impersonate the pipe client
ImpersonateSelf	Returns an access token impersonating that of the calling process (often used for changing access at a thread level)
LogonUser	Allows a server app to acquire an access token for a given user, to use in impersonated access

RevertToSelf	Terminates the impersonation of a client application
RpcImpersonateClient	Used by a server thread that is processing client remote procedure calls to impersonate the active client
SetThreadToken	Assigns an impersonation token to a thread or used to cause a thread to stop using an impersonation token

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

LogonUser isn't included in the online alphabetical list of security functions, but I think it belongs here. You can frequently use it in your server applications to ensure that any requests you're fulfilling on the requesting user's behalf match the security settings assigned to that user. You can log in as a particular user, receive an access token for that user, then use it with calls to `ImpersonateLoggedOnUser` to make your thread have the corresponding access rights for that user. New processes can be created in a similar way for another user by using `CreateProcessAsUser`. Stop impersonation using `RevertToSelf`.

If you've created a process under a different user access token (with `CreateProcessAsUser`) and you need to change only the access rights for a particular thread, you may want to use `ImpersonateSelf` instead.

Access Token Name

SD Function

`MakeAbsoluteSD`

Creates an SD in absolute format, given one in self-relative format

`MakeSelfRelativeSD`

Creates an SD in self-relative format, given one in absolute format

`InitializeSecurityDescriptor`

Initializes a new SD for use. Defaults to no rights granted of any kind

`IsValidSecurityDescriptor`

Validates an SD by checking the revision level of each part of the SD

`GetSecurityDescriptorControl`

Retrieves an SD's control and revision information

`GetSecurityDescriptorLength`

Returns the size, in bytes, for a given SD

`GetSecurityDescriptorDacl`

Returns a pointer to the DACL for a given SD

`GetSecurityDescriptorGroup`

Returns a pointer to the primary group SID for a given SD

`GetSecurityDescriptorOwner`

Returns a pointer to the owner SID for a given SD

`GetSecurityDescriptorSacl`

Returns a pointer to the SAACL for a given SD

`SetSecurityDescriptorDacl`

Updates a given SD's DACL with new settings

SetSecurityDescriptorGroup	Updates a given SD's group SID with new settings
SetSecurityDescriptorOwner	Updates a given SD's owner SID with new settings
SetSecurityDescriptorSacl	Updates a given SD's SACL with new settings

SDs are the other “big” structure in NT/2000's security (besides access tokens), as they contain all the security permissions for a given object (or group of objects), represented internally as a collection of substructures: SIDs, ACLs, and ACEs.

After allocating memory for an SD, you'll need to call `InitializeSecurityDescriptor` before you can use the SD. Note the get and set functions for the four main parts of each SD: the owner SID, the group SID, the SACL, and the DACL.

Access Token Name	SID Function
<code>AllocateAndInitializeSid</code>	Allocates and initializes a SID with up to eight subauthorities (groups or users)
<code>AllocateLocallyUniqueId</code>	Allocates a locally unique identifier
<code>InitializeSid</code>	Initializes a SID structure with a given number of subauthorities
<code>CopySid</code>	Returns a copy of a SID to a buffer
<code>EqualPrefixSid</code>	Boolean test of two SIDs' prefix values for equality
<code>EqualSid</code>	Boolean test of two SIDs' prefix values for equality
<code>FreeSid</code>	Boolean test of two SIDs' exact equality
<code>GetSidIdentifierAuthority</code>	Returns a pointer to a SID's top-level authority (in <code>SID_IDENTIFIER_AUTHORITY</code>)
<code>GetSidLengthRequired</code>	Returns the length needed, in bytes, to store a SID with a given number of subauthorities
<code>GetSidSubAuthority</code>	Returns a pointer to the <i>n</i> th subauthority for a given SID
<code>GetSidSubAuthorityCount</code>	Returns the number of subauthorities in a given SID
<code>GetLengthSid</code>	Returns the length, in bytes, of a given SID
<code>IsValidSid</code>	Validates a given SID by verifying that the revision number is within a known range and that the number of subauthorities is below the maximum
<code>LookupAccountSid</code>	Returns the account name and first domain found for a given SID

SIDs uniquely identify users and groups that have or do not have access to some object or permission. Note that with SIDs, you can allocate and initialize using a single function call. As with SDs, a newly allocated SID needs to be initialized before you can use it. Also note the `CopySid` function, which can be used for “undo” operations as mentioned above. `EqualSid` is handy for comparing two SIDs.

Access Token Name	ACL, DACL, or SACL Function
<code>InitializeAcl</code>	Creates a new ACL structure
<code>GetAclInformation</code>	Retrieves information (such as revision number, size, and ACE count) for a given ACL
<code>IsValidAcl</code>	Validates a given ACL by checking version number and verifying that the number of ACEs matches the <code>AceCount</code>
<code>SetAclInformation</code>	Sets information about an ACL

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

ACLs contain all the “ands, buts, ors, and nors” that qualify access to an object for particular users and/or groups. Each entry in the list is an ACE. Don’t forget to initialize your ACL after allocating and prior to using.

Access Token Name	ACE Function
AddAce	Adds one or more ACEs to a given ACL, at a specified index position.
AddAccessAllowedAce	Adds an Access Allowed ACE to an ACL, thereby granting access for a particular SID.
AddAccessDeniedAce	Adds an Access Denied ACE to an ACL, thereby denying access for a particular SID.
AddAuditAccessAce	Adds a system-audit ACE to a SACL.
AddAuditAccessAceEx	Adds a system-audit ACE to the end of a SACL. Identical to the AddAuditAccessAce function, except for allowing flags to be specified that control whether the new ACE can be inherited by child objects.
AddAuditAccessObjectAce	Adds a system-audit ACE to the end of a SACL. The new ACE can audit access to an object, or to a property set or property on an object, and it may also be used to add an ACE that only a specified type of child object can inherit.
DeleteAce	Deletes the <i>n</i> th ACE from a given ACL.
FindFirstFreeAce	Returns a pointer to the first free position in an ACL.
GetAce	Returns a pointer to the <i>n</i> th ACE for a given ACL.

These functions operate on ACLs (both DACLs and SACLs), permitting you to add permissions or place restrictions as needed. Note that when calling AddAce or DeleteAce you must supply an index parameter, as the ordering of ACEs in an ACL matters. (For more details on ACE-ordering, please see the discussion about this in my comments on the fileuser.cpp example program, which is discussed later in this chapter.)

Note that the ordering of the ACEs in the list determines the order in which permissions/restrictions are

checked, and ACLs are simply lists of structures (and not some nice OOP collection). Therefore, when adding a new ACE, you must first traverse the list, using `FindFirstFreeAce`, or else insert the ACE into the desired place in the list using `AddAce`.

Access Token Name	Function for Checking Access
<code>AccessCheck</code>	Used by a server application to check whether a client has access to an object.
<code>AccessCheckAndAuditAlarm</code>	Performs <code>AccessCheck</code> and generates corresponding audit messages.
<code>AccessCheckByType</code>	Determines whether a security descriptor grants a specified set of access rights to the client identified by an access token. Checks the client's access to a hierarchy of objects, such as an object, its property sets, and properties. Grants or denies access to the hierarchy as a whole. Typically used by server apps to check access to a private object.
<code>AccessCheckByTypeAndAuditAlarm</code>	Determines whether a security descriptor grants a specified set of access rights to the client being impersonated by the calling thread. Checks the client's access to a hierarchy of objects, such as an object, its property sets, and properties. Grants or denies access to the hierarchy as a whole. If the security descriptor has a SACL with ACEs that apply to the client, the function generates any necessary audit messages in the security event log; alarms are not supported in the current version of Windows NT.
<code>AccessCheckByTypeResultList</code>	Determines whether a security descriptor grants a specified set of access rights to the client identified by an access token. Checks the client's access to a hierarchy of objects, such as an object, its property sets, and properties. Reports the access rights granted or denied to each object type in the hierarchy. Typically used by server applications to check access to a private object.
<code>AccessCheckByTypeResultListAndAuditAlarm</code>	Determines whether a security descriptor grants a specified set of access rights to the client being impersonated by the calling thread. Checks access to a hierarchy of objects, such as an object, its property sets, and properties. Reports the access rights granted or denied to each object type in the hierarchy. If the security descriptor has a SACL with ACEs that apply to the client, the function generates any necessary audit messages in the security event log; alarms are not supported in the current version of Windows NT.
<code>AdjustTokenPrivileges</code>	Enables or disables privileges in the specified access token—requires <code>TOKEN_ADJUST_PRIVILEGES</code> access.
<code>AreAllAccessesGranted</code>	Compares granted with desired rights to see whether ALL specified access rights have been granted.
<code>AreAllAccessesGranted</code>	Checks whether a set of requested access rights have been granted. Access rights are represented as bit flags in a 32-bit access mask.
<code>AreAnyAccessesGranted</code>	Compares granted with desired rights to see whether ANY specified access rights have been granted.
<code>GetTokenInformation</code>	Retrieves a specified type of information about an access token—calling process must have appropriate access rights to obtain the information.
<code>LookupPrivilegeValue</code>	Retrieves the locally unique identifier (LUID) used on a specified system to locally represent the specified privilege name.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

AccessCheck and PrivilegeCheck are the two main functions used for checking whether a given user token has the requested rights or privileges, respectively. Use MapGenericMask to convert generic access rights to rights that apply more specifically to the given object. Note that the AuditAlarm functions may not actually generate auditing information on earlier versions of NT/2000. Also, keep in mind the overhead (timewise) that attaches to each object access while auditing is enabled.

Access Token Name

Privileges Function

MapGenericMask

Maps generic access rights of a given mask to specific and standard access rights

PrivilegeCheck

Tests whether a given access token has the specified privilege(s)

LookupPrivilegeDisplayName

Retrieves the displayable name associated with a particular privilege

LookupPrivilegeValue

Returns the LUID associated with a given privilege on a given system

LookupPrivilegeName

Returns the privilege name associated with the given Locally Unique Identifier (LUID)

ObjectPrivilegeAuditAlarm

Generates audit messages when the given user (access token) attempts to perform privileged operations on a particular object

PrivilegedServiceAuditAlarm

Generates audit messages when the given user (access token) attempts to perform privileged operations

(Additional functions)

See the LSA functions in next section

The two most frequently used functions in this group are MapGenericMask and PrivilegeCheck. Note that prior to using LookupPrivilegeName, you would need to call LookupPrivilegeValue, to obtain the LUID for the privilege.

WARNING:

The LUID will (by definition) be completely different from one machine to the next, even though the name of the privilege may be identical to a privilege on another system, so be sure to specify the correct server name when

Access Token Name	Local Service Authority (LSA) Functions (Special)
InitLsaString	Creates an LSA_UNICODE_STRING for the given privilege name
LsaOpenPolicy	Opens (or creates) a given policy on a target machine
LsaLookupNames	Returns the account name (and SID) for a given access token
LsaRemoveAccountRights	Revokes privileges for the specified user(s)
LsaAddAccountRights	Grants privileges to the specified user(s)
LsaClose	Closes a given policy
LsaEnumerateAccountRights	Enumerates rights or privileges that are currently granted to a given account
LsaEnumerateAccountsWithUserRight	Enumerates all accounts on a given system that have been granted a given privilege

These and other LSA functions appear to be part of a larger API category that will be called the Win32 Licensing API (still in beta as of this writing). I include them here because they are the only means of programmatically administering certain aspects of user privileges. In particular, note that LsaAddAccountRights can be used to create a new policy (by name) on a given machine. Currently, these functions support Unicode only. They are supplied via NT/2000's LSAPI32.DLL.

Access Token Name	Get/Set Security Information Function
GetFileSecurity	Obtains the SD for a particular file or directory
GetKernelObjectSecurity	Obtains the SD for a specified kernel object
GetPrivateObjectSecurity	Obtains the SD for a specified private object
GetUserObjectSecurity	Obtains the SD for a specified user object
SetFileSecurity	Updates a file or directory's security using the supplied SD
SetKernelObjectSecurity	Updates a kernel object's security using the supplied SD
SetPrivateObjectSecurity	Updates a private object's security using the supplied SD
SetUserObjectSecurity	Updates a user object's security using the supplied SD
CreatePrivateObjectSecurity	Allocates and initializes a self-relative SD to be used with a new private object
DestroyPrivateObjectSecurity	Deletes the given private object's SD

These functions are the essential tools for getting and setting (changing) the SDs for secured objects. Note the functions for private object security. These are what you'll need whenever you wish to create security restrictions for accessing some object (or class of objects) that your application defines. Don't confuse private objects with user objects, which are simply NT/2000 secured objects created by a user (such as a file the user has created).

Access Token Name	Window Station Function
GetProcessWindowStation	Returns a handle to the window station associated with the calling process
SetProcessWindowStation	Assigns a given window station to the calling process

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

These functions enable a process to access the objects of a particular *window station*. These objects include a desktop, a clipboard, and a list of global atoms.

Access Token Name

Auditing Function

ObjectOpenAuditAlarm	Generates audit messages to log when a user attempts to gain access to an existing object or create a new one
ObjectCloseAuditAlarm	Generates audit messages when the handle of an object is closed
ObjectDeleteAuditAlarm	Generates audit messages when the handle of an object is deleted
ObjectPrivilegeAuditAlarm	Generates audit messages when the given user (access token) attempts to perform privileged operations on a particular object
PrivilegedServiceAuditAlarm	Generates audit messages when the given user (access token) attempts to perform privileged operations
AccessCheckAndAuditAlarm	Performs AccessCheck and generates corresponding audit messages

These functions cause audit messages to be generated whenever the specified type of access is attempted. If you use audit alarms, you should keep in mind that some overhead is added to each object-access operation while logging is enabled.

TIP:

If your application needs to frequently access an audited object, you should experiment with enabling and disabling auditing to decide how much impact auditing will have on performance.

Checking and Updating SDs: The *FileUser* Program

TIP:

The first demo in this chapter shows how to retrieve a file's current SD and then update it by adding either an AccessAllowed ACE or an AccessDenied ACE for a specific user. To keep this example simple, we're using a command-line, Win32 console application that expects (as command line parameters) a filename, a username, and a "+" (access allowed) or "-" (access denied). The complete source code for this program is found on the accompanying CD. The *FileUser* demo application is expected to run from a client (workstation) connected to a server through a trust relationship.

After checking command-line parameters, *FileUser* begins by allocating memory for two Security Descriptors. One will be used to retrieve the file's current SD while the other will be used to build the new SD, containing the contents of the current SD plus a new ACE, which will be added to the DACL.

```
pFileSD    = malloc( FILE_SD_SIZE );
pNewFileSD = malloc( FILE_SD_SIZE );
```

Note that there are different constants defined for different types of SDs. In this case, since we will be dealing with an SD for a file, the type will need to be `FILE_SD_SIZE`.

The next step is to set a flag which will be used shortly to determine which type of ACE to add (an AccessAllowed or AccessDenied type), based on whether the user enters a "+" (to allow) or a "-" (to deny).

```
bAllowAccess = (argv[3][0] == '+');
```

Retrieving the SID

After setting the action flag, *FileUser* continues by retrieving the SID for the account name specified on the command line (`argv[2]`). Note that we're asking for a `SID_NAME_USE` SID type, as `SIDType` was defined as:

```
SID_NAME_USE    SIDType;

// get the SID of the user (or group)
Check( LookupAccountName( (LPSTR) NULL, argv[2], UserSID,
                          dwSIDLength, szDomainName,
                          &dwDomainLength, &SIDType ),
      "LookupAccountName" );
```

The `LookupAccountName` function is used to perform a search for the username on the current server or domain, and if successful, will return:

- The domain name (in `szDomainName`) where the user was found
- The length of the domain name returned (in `dwDomainLength`)
- The user's SID (in `UserSID`)

`UserSID` is defined as a 2K buffer, which, except in very rare cases, should be sufficient.

TIP:

The `Check` function provides rudimentary error trapping. If the called function returns `TRUE` (success), `Check` simply returns doing nothing. If, however, an error occurs, `Check` tests the last error result and reports the failed function together with the error explanation. Note that the `Check` function is provided for debugging purposes but is not appropriate for a release product where a more informative error or event report should be used.

Getting the File's SD

Next, the current Security Descriptor for the file is required and is retrieved by calling `GetFileSecurity`. If successful, this function will copy the file's SD to `pFileSD`, together with additional security information in `SecInfo`, which is of type `SECURITY_INFORMATION`.

```
// get the file's current SD.
Check( GetFileSecurity( argv[1], SecInfo, pFileSD, dwFileSDLength,
```

```
        &pwdLengthNeeded ),  
    "GetFileSecurity" );
```

While pNewFileSD has been allocated in memory, it has not yet been initialized, a task which is handled by calling InitializeSecurityDescriptor. The SECURITY_DESCRIPTOR_REVISION is defined by the Win32 API, and should always be used with new SDs to ensure that the SD will contain the current version number, so that validation functions can work correctly.

```
    // initialize a new SD (larger to hold the new user's SID)  
    Check( InitializeSecurityDescriptor( pNewFileSD,  
                                         SECURITY_DESCRIPTOR_REVISION ),  
          "InitializeSecurityDescriptor");
```

While the new file SD is valid, it is also empty and the existing information needs to be copied from the current file SD and the new user ACE added before setting this new pNewFileSD back as the file's SD.

WARNING:

If this new SD were assigned "as is" to the file, no one would have access to the file, since an empty SD means no access rights have been granted. (Remember, for SDs, empty = no access, while NULL = open access.)

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Working with the DACL

The GetSecurityDescriptorDacl is used to retrieve the file's current DACL (assuming that it has one) so that these can be duplicated in pNewFileSD. If there is a DACL in pFileSD, GetSecurityDescriptorDacl will load pFileDACL with the current DACL and set the bContainsDACL flag to TRUE. Or, if there isn't a DACL, bContainsDACL will be set to FALSE. At the same time, the flag bDefaultedDACL will be set to FALSE if the current DACL was set explicitly by a user, or TRUE if the DACL came from a default DACL source.

```
// get the file's DACL (from the SD)
Check( GetSecurityDescriptorDacl( pFileSD, &pbContainsDACL,
                                   &ppFileDACL, &pbDefaultedDACL ),
       "GetSecurityDescriptorDacl");
```

Once the DACL has been retrieved, the next step is to determine the record's size so that a new (and larger) DACL can be created to include the new ACE. The GetAclInformation function returns this data in the ACLSizeInfo structure.

```
// if file currently has a DACL, get the ACL size info.
if( bContainsDACL )
    Check( GetAclInformation( pFileDACL, &pACLSizeInfo,
                              sizeof(ACL_SIZE_INFORMATION),
                              AclSizeInformation ),
           "GetAclInformation" );
```

Notice that the size of the structure (which is of type ACL_SIZE_INFORMATION) is passed as an argument while the AclSizeInformation enumerated type is specified to instruct GetAclInformation that the information required is the ACL size.

Once this is returned, the correct size for the new DACL can be calculated as the size of the current DACL plus the space needed for one new ACE. The structure record AclInfo.AclBytesInUse reports the current DACL's size and we need to add sufficient space for either an ACCESS_ALLOWED_ACE or an ACCESS_DENIED_ACE (depending on the bAllowAccess flag) plus enough space for the new user's SID.

Also, since NT/2000 adds a terminating code at the end of a self-relative DACL, sizeof(DWORD) bytes are subtracted from the total. (This is the size of the terminator and it does not need to be counted twice.)

```
// calculate size we'll need for current ACL, plus new ACE.
dwNewACLSize = ACLSizeInfo.AclBytesInUse +
    ( bAllowAccess ? sizeof( ACCESS_ALLOWED_ACE )
      : sizeof( ACCESS_DENIED_ACE ) ) +
    GetLengthSid( UserSID ) - sizeof(DWORD);
```

Once the size has been calculated, the new DACL can be allocated:

```
// allocate and initialize new ACL
pNewFileDACL = (PACL) malloc( dwNewACLSize );
```

Now pNewFileDACL is a pointer to the right amount of memory for a DACL, but it's not really a DACL until it's initialized. Again, don't forget to initialize these structures after allocating!

```
Check( InitializeAcl( pNewFileDACL, dwNewACLSize, ACL_REVISION2 ),
    "InitializeAcl" );
```

For educational purposes, the next call is to IsValidAcl, in order to make sure this is a valid DACL. At this point, it should certainly be valid, as there's nothing to it except the revision information.

```
// Verify we've set up our DACL correctly.
Check( IsValidAcl( pNewFileDACL ), "IsValidAcl" );
```

Adding ACEs

At this point a decision is required, because if the new ACE is an ACCESS_DENIED_ACE, this ACE needs to be added to the new DACL before copying over the other ACEs. Alternately, if this is an ACCESS_ALLOWED_ACE, this should be added at the end of the list of other ACEs.

The difference comes because of how NT/2000 checks the ACLs. Any entries denying access should be placed *before* entries allowing access. Otherwise, it is possible for NT/2000 to encounter an entry granting access—perhaps access granted to a group—and to act on this entry without checking further and without reaching the exclusion entry.

So, when adding an ACCESS_DENIED_ACE, you must add it now, before copying the other ACEs to the new DACL.

```
// if DENYING access rights, place the new ACE first
// in the new DACL.
if( ! bAllowAccess )
    Check( AddAccessDeniedAce( pNewFileDACL, ACL_REVISION2,
        GENERIC_READ, &UserSID ),
        "AddAccessDeniedAce" );
```

Note that GENERIC_READ is specified as the right being allowed or denied. You could experiment by changing this to GENERIC_WRITE (or adding another command-line switch to set the file access) to allow or deny write access to the file.

Next, the process loops through the current DACL's list of ACEs, copying each in turn to the new DACL. Of course, if ACLSizeInfo.AceCount is 0, this step can be skipped since there aren't any ACEs to copy. On each loop, GetAce is called to fill the pTempACE pointer with the indexed ACE and then AddAce is called to add the ACE to pNewFileDACL.

```
// copy each existing ACE (if any) over to the new DACL
if( bContainsDACL ) // if current file SD contains a DACL...
{
    // and if the DACL has any ACEs
    if( ACLSizeInfo.AceCount > 0 )
    {
        // for each ACE, copy it to the new file DACL
```

```

for( iAceNum=0; iAceNum<ACLSizeInfo.AceCount; iAceNum++ )
{
    // read ACE from current ACL, write to new ACL
    Check( GetAce( pFileDACL, iAceNum, &ppTempACE ),
           "GetAce" );
    Check( AddAce( pNewFileDACL, ACL_REVISION, MAXDWORD,
                  pTempACE,
                  ((PACE_HEADER) pTempACE)->AceSize ),
           "AddAce" );
}
}
}

```

Once any existing DACLs have been copied, this is the point to add an ACCESS_ALLOWED_ACE as:

```

// if GRANTING access rights, add new ACE at end of the list
if( bAllowAccess )
    Check( AddAccessAllowedAce( pNewFileDACL, ACL_REVISION2,
                              GENERIC_READ, &ppUserSID ),
           "AddAccessAllowedAce" );

```

Now that all of the needed ACEs have been added to the new DACL, the list needs to be set to the new file descriptor (pNewFileSD) through a call to SetSecurityDescriptorDacl as:

```

// set the new DACL to the new SD
Check( SetSecurityDescriptorDacl( pNewFileSD, TRUE,
                                pNewFileDACL, FALSE ),
       "SetSecurityDescriptorAcl" );

```

Here, the first Boolean value (TRUE) confirms that there is a DACL to use while the second Boolean value (FALSE) reports that the DACL wasn't arrived at by a default mechanism.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Setting the New SD

Now that all of the information is ready, it's time to write the new SD for the file using a call to SetFileSecurity. The DACL_SECURITY_INFORMATION flag means that only the DACL should be updated while the other security information—the group and owner SIDs and the SACL—remains unchanged.

```
// set our new SD to the object.  
Check( SetFileSecurity( argv[1], DACL_SECURITY_INFORMATION,  
                        pNewFileSD ),  
        "SetFileSecurity" );
```

NOTE:

If the objective had been to update the SACL or the owner or group SIDs, these could also be changed. However, setting the SACL would require running the program with system-level access, while setting the group SIDs would require choosing whether to replace the current (existing) user(s) or to add the new user to these. To change the owner SID, rights to change the file's ownership would also be required. As it is, in order to execute the FileUser program successfully, you must have owner access to set the DACL.

In this example, all that remains is to clean up by deallocating the pNewFileDACL, the pNewFileSD, and the pFileSD, which were allocated earlier.

```
// finally, cleanup, prepare to leave.  
free( pNewFileDACL );  
free( pNewFileSD );  
free( pFileSD );  
  
return(0);  
}
```

The FileUser demo should offer some understanding of how security rights are set or modified for an object. Modifying this example program to allow no access rights or all access rights, or to change ownership of the file, should be a relatively simple process. Likewise, with a change to the DACL type, this code can be used

to modify the security on other types of objects as well.

Now that you've seen how data objects—or functionality—can be secured on an NT/2000 server application using the security function built into NT/2000, the next section will look at another, complementary approach to securing information using encryption.

Encryption Technology Concepts

Sending information over a network involves transmitting a series of packets. While traversing the net, these packets can be intercepted by a variety of means at a variety of points along the path, and there is no absolutely secure method of ensuring that data cannot be intercepted. However, if interception prevention cannot be assured, encryption can be used to ensure that interception does not mean that the security of the data is compromised.

While cryptography does not necessarily require a computer—in the ancient days of the Roman Empire, Caesar directed his generals using encrypted messages—computer encryption/decryption is certainly faster and more convenient than any other method. This is especially so when the data is already present in a computer format, as it is so often today.

NOTE:

This is not to imply that data must be in any specific format prior to encryption or following decryption, only that cryptography can conveniently be applied to any computer file, irrespective of the file's format.

Secure Communications and the Government

As an interesting conversation topic, one could make the observation that anytime there is competition, the securing and protecting of information becomes an essential strategy. Historically, codes and ciphers were put to use early both in commerce and warfare, with governments of course being key users (pun intended!) of the information securing technology of the time. It is only to be expected, then, that governments worldwide would continue to have an active interest in the use (and abuse) of data-securing techniques. Combine that with the fact that we're now living in the "information age," particularly with the increasing "Internetization" of all sectors of information, and we find an unprecedented interest in—and growing need for—data-encryption technology by both governments and citizens alike.

At the government level, we find in the United States, for example, that current federal laws categorize most effective encryption technology as *munitions*, making it a felony to export at least the more sophisticated types of data encryption to other countries. France, on the other hand, takes the opposite approach of outlawing the *import* of any encryption technology into France. This is only one good reason why NT/2000's Crypto API supports optional use of encryption in a modular framework, permitting third-party encryption-provider companies to supply customers with installable encryption algorithms. (The French version of Windows NT/2000, by the way, has no support for built-in encryption.) Even more controversial now in the U.S. is the government's expressed desire to be "caretaker" of encryption technology (with the Clipper III project and its successors).

It remains to be seen how the tension between governments of various countries and their citizens, in the use of data encryption, plays itself out. Certainly there are important conflicting needs (the maintaining of law and order versus rights to privacy, for example) that will require some sort of ongoing one-legged balancing act.

Do You Need Encryption?

Encrypting data imposes some costs—in development time, in affecting the rate of data throughput, and, of course, in an increased complexity for users—and the decision to use or not use encryption has to be weighed by a number of factors, including:

- How many people need to have access to this information?
- How quickly will the information become outdated (and therefore less sensitive)?
- What do you consider an acceptable level of risk (of information compromise)?
- If the data needs to be exchanged across a network, how secure is the network itself?
- Consider the possible repercussions of your secured data falling into the wrong hands—in other words, what's the worst that could happen?
- How valuable (in terms of time, money, or other values) is this information to you?

- How valuable (in the same or other terms) could this information be to someone else?
- What costs are involved, both in implementing and in using the desired encryption technology?

Eventually, as computers become faster and as software development tools progress, encryption objects can be expected to make the costs of encryption almost trivial. At the same time, by integrating the encryption objects into systems, the complexity required by decryption will also vanish.

The remaining costs—principally data throughput—should also be minor since the primary overhead lies simply in the encryption/decryption process, and the coded data is normally not materially different in size from the original data file.

Of course, as computers become faster and more powerful, the tools for breaking codes also become more powerful. The end result is a simple race, as developers work to create more powerful algorithms for encryption at the same time that the code breakers (including a few sophisticated crooks) are seeking more powerful methods of circumventing protective measures.

Encryption Methods

In recent years, many advances have been made in cryptographic technology, with various approaches being used, varying levels of strength against “cracking” being achieved, and tradeoffs being incurred, which inevitably come with taking a variety of approaches.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Public-Key Encryption

One type of encryption that is quite popular because of its strength and other features is *public-key* encryption.

Public-key encryption involves each party having two keys, one that is kept private to everyone but the owner, and another that can be made very public—it can be placed on Web pages, sent in an e-mail message, given out on floppy disks, and so on. Messages (or data) are then encrypted using the private key of the sender, plus the public key of the intended recipient. On the other end, the data can be successfully decrypted only using the opposite pair of the sender's public key, combined with the recipient's private key. This type of encryption achieves its high level of security at the expense of speed—it takes several orders of magnitude longer to encrypt and decrypt using public-key technology than it does using certain other forms of encryption algorithms.

Digital Signatures and Authentication

On the other hand, the public-key approach does provide a second, built-in feature that is arguably just as important as encryption itself. Public keys provide the ability to “sign” encrypted data in such a way that the recipient can be certain that the decrypted message he is reading actually came from the individual whose public key was used in decrypting. Because two keys are always used, it would be necessary for someone to steal one or the other private key before they could “forge” a public-key encrypted message that decrypts successfully using the recipient's private key. Moreover, it is possible also to use the same encryption techniques to get the signature or “verified sender” benefit without needing to encrypt the information. This becomes useful in situations where neither party cares whether the information conveyed is made public, their chief concern being simply whether the message originated from whom it claims to have originated, and whether the message was altered (accidentally or intentionally) while in transit.

Symmetric-Key Encryption

Another popular form of encryption, called *symmetric-key* encryption, requires using the same key for both encrypting and decrypting. Obviously, this works only if the key used remains undisclosed to anyone but the sender and recipient. This technique, while not as secure as public-key encryption, is significantly faster in operation—as much as a thousand or more times faster, in fact, than using public-key technology. A disadvantage of the symmetric-key approach is that it requires that the key also be conveyed somehow from

sender to recipient, and this also means that the key itself is subject to interception.

Combining Public- and Symmetric-Key Encryption

Because on the one hand public-key algorithms provide extreme security (plus signatures) at the expense of speed, and symmetric algorithms provide speed at the expense of somewhat weaker security, one approach now commonly used includes making use of a combination of both techniques. A symmetric encryption algorithm is used to encrypt the main body of plaintext (or data), then the symmetric key that was used gets encrypted using the much stronger, but slower, public-key encryption. This public-key encrypted symmetric key can then be safely included along with the body of encrypted text (cyphertext), and allows the symmetric key to be retrieved by the recipient(s) whose public keys were used to encrypt the sender's symmetric key. This combines the benefits of faster encryption for the bulk of the information with stronger encryption to safeguard the enclosed symmetric key, and also provides the benefit of a digital signature.

Stream versus Block Cyphers and Padding

Certain encryption algorithms are designed to operate on streams of data—they take the data one character at a time, and output encrypted data at the same rate. Block-encryption algorithms, on the other hand, are designed to encrypt chunks or blocks of data all together. Generally, block encryption is viewed as being somewhat more secure. Note that the final block of data will rarely fill the entire block, so typically padding bytes are added to the final block.

Hashing

Many encryption algorithms generate a residual (hash) value that is continually combined with the current data, producing a new hash that is again combined with the current hash. A hash value of a certain number of bits is generated by applying some encryption-like algorithm to the data, such that the resulting hash value will be different if even one bit of the source data is lost or tampered with. Obviously, the longer the hash value, the lower the odds are of a hash value equaling another hash value produced from a tampered-with file. Hashing can be used either by itself or along with encryption, in either case providing a value that can be used for digital signing and/or for error/tamper-proofing a chunk of data.

NOTE:

Virus checkers frequently use a form of hashing on executable files to determine whether the file has been tampered with. Java code, ActiveX controls, and other software downloaded from the Internet will be subject to similar verification techniques.

Crypto API and Cryptographic Service Providers (CSPs)

In recent years, many companies, individuals, and government institutions have devised their own algorithms based on variations of the public-key and symmetric-key approaches. At least for now, the stronger candidates among these algorithms are quite secure indeed. However, one remaining problem in using encryption generally is a lack of standards. Not only are the various algorithms that are used quite different, but even using the same published algorithm, software implementations by different people can result in incompatible encrypted output.

Initiatives such as Microsoft's new Crypto API are exciting in their potential, as certain of the standards problems are clearly being addressed by the modular design, which allows users the ability to install any third-party cryptographic algorithms as simply as installing a new printer driver. This frees programmers to concentrate on building applications that can make use of whichever algorithms are chosen by the users, subject to certain restrictions such as applicability of the algorithm to encryption of streaming versus blocks of data, or key-only types of encryption algorithms.

A *Cryptographic Service Provider* (CSP) is any agency that offers an algorithm or a set of algorithms corresponding to the Crypto API interface such that application software is able to make use of the algorithms by selecting them at runtime.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Crypto API Functions

The Microsoft RSA Base Provider consists of a software implementation of the PROV_RSA_FULL provider type. The RSA public-key cipher is used for both key exchange and digital signatures, with a key length of 512 bits. The RC2 and RC4 encryption algorithms are implemented with a key length of 40 bits. The MD2, MD5, and SHA hashing algorithms are also provided. Table 11.3 outlines the Cryptographic Service Provider (CSP) functions.

TABLE 11.3: CSP Functions

Function	Description
CryptAcquireContext	Acquires a handle to the current user's key container within a particular CSP
CryptGetProvParam	Retrieves attributes of a CSP
CryptReleaseContext	Releases the handle acquired by CryptAcquireContext
CryptSetProvider	Specifies the user default CSP for a particular CSP type
CryptSetProvParam	Specifies attributes of a CSP

TIP:

There are a vast number of other cryptography functions included in the API. While the five listed here are simply a sample of the basic operations, a number of additional functions are discussed in the following.

You can consider CryptAcquireContext and CryptReleaseContext as two bookend functions that need to go at the start and end, respectively, of the code in your applications that makes use of any of the Crypto functions. Note that CryptAcquireContext can be called with a NULL value for the CSP, which results in the default CSP for the current user being used.

Key Functions

The key functions used for cryptography are:

CryptDestroyKey Destroys a given key

CryptExportKey Converts a key from the CSP into a *key blob* (an exportable version of the key used both for encryption and decryption) that can be safely shared or saved to disk

CryptGenRandom Generates some random data (used for salt bits)

CryptGetKeyParam Retrieves the parameters for a given key

CryptGetUserKey Retrieves a handle to an exchange key or a signature key

CryptImportKey Converts a key blob into a key usable by a CSP

CryptSetKeyParam Sets the parameters for a key

CryptGenRandom is very useful for coming up with keys to be used with symmetric-key algorithms. Remember, however, that your generated keys must be saved somewhere (preferably somewhere secure), as this is not done automatically. The proper way to save a key is to create an exportable version of it using CryptExportKey, then use CryptImportKey when you need to read it back from disk.

Encryption Functions

The encryption/decryption functions are:

CryptEncrypt Encrypts a block or stream of data using the specified encryption key

CryptDecrypt Decrypts a block or stream of data using the specified encryption key

These two functions are of course the Crypto star players, performing encryption or decryption using the specified algorithm on a data block or stream.

Hashing Functions

Hashing functions are used for a variety of purposes, including providing *signatures* to validate encrypted objects.

CryptCreateHash Creates an empty hash object

CryptDestroyHash Destroys a hash object

CryptGetHashParam Retrieves a parameter from a hash object

CryptHashData Hashes a block of data and includes the result in a specified hash object

CryptHashSessionKey Hashes a session key and includes the result in a specified hash object

CryptSetHashParam Sets a parameter of a given hash object

CryptSignHash Signs the specified hash object

CryptVerifySignature Verifies the digital signature from a signed hash object

Hashing can be used for verifying that received data is from an authentic source, and that it hasn't been tampered with. The odds of two different chunks of data randomly happening to have the same hash value is so small as to be *almost* impossible. CryptHashSessionKey and CryptHashData are both very useful for creating hashes of keys or data, respectively.

CryptSignHash and CryptVerifySignature can be used on either end to add a signature to the hash or to verify that a signature “squares” with a hash value.

Adding Encryption Support to Applications

The *Crypto* sample application demonstrates how the Crypto API functions can be used to provide encryption support for an application. However, before using the *Crypto* application (or the Crypto API functions) you will need to ensure that your system is configured appropriately.

Getting Started

Older systems, such as Windows 95, may require downloading and installing the Crypto API library (WinCrypt.DLL) from Microsoft's Web site. Also, older non-Microsoft compilers may require generating import libraries for the WinCrypt.DLL.

For Older Systems/Compilers

If your system or your compiler do not support the Crypto API functions, the Crypto API SDK is available by free download from the Internet at Microsoft's developer pages or from the MSDN (Microsoft Developer Network) CD-ROM.

For Microsoft Compilers, the Crypto API is already supported. That is, the Crypto API SDK is included. Likewise, Borland C++ 5.0 and later and C++ Builder already include the Crypto API header and library.

For older Borland C/C++ compilers, the import libraries can be created using ImpLib.exe (run as implib dllname).

For Borland Pascal or Delphi (prior to version 3.0), an import unit is required. This is simply a Pascal unit with function prototypes specified (in Pascal) along with an implementation section that specifies which DLL each function is exported from. For examples of import libraries, refer to the Delphi Windows import units.

Installing the Crypto API Runtime Support

For Windows 95 workstations that do not have the Crypto API runtime installed, you may:

- Install Microsoft's Internet Explorer (version 3.0 or later), which uses, and therefore installs, the Crypto API.
- Run the Setup applet supplied with the Crypto API SDK.

Support for the Crypto API functions is included in Windows 98 and later. All systems, however, will require a default CSP and default keys to be generated before the Crypto API functions can be used.

TIP:

For demonstration purposes, the *Crypto* program includes an option to generate the default CSP and default keys. Alternately, you can use the `InitUser.C` sample applet included with the Crypto API SDK.

Once your system is configured appropriately, the *Crypto* program will demonstrate encrypting and decrypting files. How these processes are carried out is discussed in the following sections.

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Required Project Settings

If you are going to include the Crypto API functions in your application, you will need to begin by making one provision in your project settings to define `_WIN32_WINNT` as `0x0400` (for Windows NT) or `0x0500` (for Windows 2000). Using Visual C++, select Project Settings C/C++. From the “Settings For:” list, select “All Configurations”; then, in the “Preprocessor definitions:” field, type `_WIN32_WINNT=0x0500` as shown in Figure 11.3. Notice that there are no spaces in the preprocessor entry and also that the entries are separated by commas.

TIP:

By specifying `_WIN32_WINNT=0x0500` rather than `_WIN32_WINNT=0x0400`, you’re setting yourself up for future changes where NT (version 4.0) will be superseded entirely by a new version. Since the existing `#ifdef` statements specify equal to or greater than 4.0, no present conflict exists.



FIGURE 11.3 Setting Preprocessor definitions

Click the OK button.

Next, in your project’s `StdAfx.h` header file, include the line:

```
#include <wincrypt.h>
```

And that’s it; your project is ready to support the Crypto API functions.

TIP:

If you look in the `WinCrypt.H` header, you will find `_WIN32_WINNT` included as: `#if(_WIN32_WINNT >= 0x0400)`. Without the preprocessor entry (preferred) or an appropriate `#define` statement, the compiler will not include the header entries. While this appears to refer to Windows NT/2000, this preprocessor definition is also required for

Creating a Default CSP

The *Crypto* demo when it is first executed is shown in Figure 11.4. As you can see, only the Initialize and Exit buttons are enabled. Since we don't know if your system has been initialized for the Crypto API functions yet, the reasonable option, for demonstration purposes, is to require initialization before going any further.



FIGURE 11.4 Initializing cryptographic support

When you click on the Initialize button the results should look something like Figure 11.5 (assuming your system has not been initialized with a default CSP and keys).

If your system has already been initialized with a default CSP and keys, the status messages reported will be somewhat briefer, reporting only the creation of the key container and retrieval of the user key. In either case, however, you are now ready to encrypt or decrypt files.

NOTE:

For dedicated Command Prompt enthusiasts, the *Encrypt.CPP* and *Decrypt.CPP* programs (included on the CD) demonstrate RC2 block-type encryption. Each is called with two command-line parameters: the names of the source and output files.

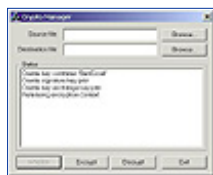


FIGURE 11.5 Initialization completed

Encrypting a File

As you can see in Figure 11.5, edit boxes are provided to enter the source and destination filenames or, optionally, you can use the Browse buttons for each to call the File Open dialog for selection.

The first few steps are the same regardless of whether you've chosen to encrypt a file or to decrypt one, and the operation starts with two constants defined as:

```
const IN_BUFFER_SIZE  = 10240;      // 4 * 40 * 64 * (8 bits)
    // note: needs to be multiple of block size
const OUT_BUFFER_SIZE = IN_BUFFER_SIZE + 64;
    // extra padding
```

Two block-encryption algorithms are supplied by the Microsoft base CSP: RC2 and RC4 encryption. Since the size of the buffer should, ideally, be a multiple of the encryption block size, the value selected here is a simple multiple of the 64-bit and 40-bit keys. Also, since (depending on the encryption algorithm used) the encrypted result can be slightly larger than the raw data, the output buffer is allocated some additional space.

The bulk of the encryption or decryption is carried out in the *Process* procedure and begins by getting a handle to the default *PROV_RSA_FULL* crypto provider:

```
if( ! CryptAcquireContext( &amphProvider, NULL, NULL,
                          PROV_RSA_FULL, 0 ) )
{
    ReportStatus( "Error %x during CryptAcquireContext!" );
    return;
```

```
}
```

Since we've specified NULL rather than an explicit CSP, the current user's default CSP is called. While we can assume that this is the Microsoft Crypto API CSP, this could be any CSP installed on the system and, if there are multiple CSPs installed, we might want to be explicit here.

The ReportStatus function is used to record any errors that occur as entries are made in the list box shown in the dialog. Of course, if an error occurs, there's no point in continuing, and a simple return terminates the operation.

Our next step is to read the name of the default CSP—using the PP_NAME key—and report this as an entry in the Status list box. Of course, this is an optional step since the name is purely for information.

```
cbData = 1000;
if( ! CryptGetProvParam( hProvider, PP_NAME, pbData,
                        &pcbData, 0 ) )
...

```

In like fashion, we'll also query and report the name of the default key container:

```
cbData = 1000;
if( ! CryptGetProvParam( hProvider, PP_CONTAINER, pbData,
                        &pcbData, 0 ) )
...

```

These previous two steps were simply for information and aren't really required during encryption or decryption.

But now we're ready to get down to the nuts and bolts of the operation, and we begin by opening our source file (the file to be encrypted) and by creating the destination file (where the encrypted data will be written).

```
pFileIn = new CFile( (LPCTSTR) m_csSource, CFile::modeCreate |
                    CFile::modeNoTruncate | CFile::modeRead );
pFileOut = new CFile( (LPCTSTR) m_csDestination,
                    CFile::modeCreate | CFile::modeWrite );

```

Generating a Random Key

After opening the two files, assuming that we're encrypting a file, our next step is to generate a random key—in "simple blob" format—by calling the CryptGetUserKey and CryptGenKey functions:

```
if( m_bEncrypt )
{
    // Generate a random key (in "simple blob" format)
    // to use for encryption.
    CryptGetUserKey( hProvider, AT_KEYEXCHANGE, &hExchangeKey );
    CryptGenKey( hProvider, CALG_RC2, CRYPT_EXPORTABLE, &hKey );

```

The hExchangeKey and hKey values are used to generate a key blob, which is an exportable version of the key used both for encryption and decryption. Generating the key blob involves two calls to the CryptExportKey function.

The first call is made with the fifth parameter specified as NULL, causing CryptExportKey to return the key size (in dwByteCount) while the second call, passing pbBuffer as the fifth parameter, returns the key blob proper (in pbBuffer).

```
dwByteCount=0;
CryptExportKey( hKey, hExchangeKey, SIMPLEBLOB, 0, NULL,
               &pdwByteCount );
CryptExportKey( hKey, hExchangeKey, SIMPLEBLOB, 0, pbBuffer,
               &pdwByteCount );

```

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Now that we have the size and the key blob, these two pieces of information are written to the output file (the encrypted destination) with the size of the key blob first, then followed by the key blob itself.

```
pFileOut->Write( &dwByteCount, sizeof(dwByteCount) );  
pFileOut->Write( pbBuffer, dwByteCount );
```

At this point, the output file contains two items: the size of the key and the key. Now we're ready to encrypt the data itself and write the encrypted version to the output file:

```
do  
{  
    dwByteCount = pFileIn->Read( pbBuffer, IN_BUFFER_SIZE );  
    finished = ( dwByteCount < IN_BUFFER_SIZE );  
    CryptEncrypt( hKey, 0, finished, 0, pbBuffer,  
                &dwByteCount, OUT_BUFFER_SIZE );  
    pFileOut->Write( pbBuffer, dwByteCount );  
}  
while( ! finished );
```

Here, blocks of material are read from the source file, passed to the CryptEncrypt function and then written to the output file, with this process repeating until the end of the file is reached. Notice, however, that the third parameter passed to CryptEncrypt is a Boolean variable that tells CryptEncrypt when the last block of data to encrypt is being passed.

While most of the blocks of encrypted data will be the same size as the raw data read from the source file, the final block of data encrypted may be smaller. However, since we're using a block encryption mode, CryptEncrypt will pad the final block out to the full block size.

TIP:

Refer to the online documentation for options and alternative encryption formats.

Once we've finished encrypting the data, there are a few cleanup tasks to handle:


```

        CryptDestroyKey( hKey );
        CryptDestroyKey( hExchangeKey );
    }
    pFileIn->Close();
    pFileOut->Close();
}

```

After destroying the two keys and closing the file, there's still one operation remaining:

```
if( ! CryptReleaseContext( hProvider, 0 ) )
...

```

After releasing the CSP, the encryption process is concluded. At this point, using the *Crypto* demo, the Status report should look something like Figure 11.6.

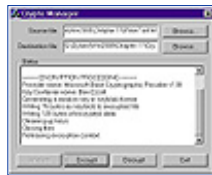


FIGURE 11.6 After encrypting a data file

As you can see from the report, the key (the key blob) is 76 bytes (608 bits) in length, which, presumably, should be a fairly secure key. The fact that the key is almost as large as the data is simply coincidence, because we've used a fairly small file to encrypt.

Decrypting a File

Decrypting a file begins in the same fashion as the encryption process, by calling `CryptAcquireContext` to get a handle to the default `PROV_RSA_FULL` crypto provider and then opening the source and output files.

Unlike the encryption process, however, the decryption process begins by reading the size of the key from the encrypted source file and then reading the key blob:

```
pFileIn->Read( &dwByteCount, sizeof(dwByteCount) );
pFileIn->Read( pbBuffer, dwByteCount );
```

After retrieving the key as an exported key blob, `CryptImportKey` is called to import the crypto key (as the name implies), which was just read from the file header:

```
CryptImportKey( hProvider, pbBuffer, dwByteCount,
                0, 0, &amphKey );
```

The crypto key is imported to make it internal to the CSP just as the export function was used to create the key blob so that it could be included in the output file.

On the surface, this might appear to be a weakness. That is, since the key is included in the encrypted file, it might initially appear as if the security of the encryption has been compromised right here. That is, anyone could simply decrypt the file using the stored key.

The key itself, however, is not a stand-alone factor, but is used in combination with key information provided by the key container. Of course, if someone had access to the system (and was able to log on as the user who encrypted the file) then decrypting the file would be trivial.

On the other hand, for encrypting a file to be transmitted to someone else, there are other options provided by the Crypto API, including the `CryptEncryptMessage` and `CryptExportPublicKeyInfo` functions. Refer to the online documentation for details.

After importing the key, decrypting the bulk of the file is simple and relies on a do loop:

```
do
{
    dwByteCount = pFileIn->Read( pbBuffer, IN_BUFFER_SIZE );
    finished = ( dwByteCount < IN_BUFFER_SIZE );
}
```

```

CryptDecrypt( hKey, 0, finished, 0, pBuffer,
              &pdwByteCount );
pFileOut->Write( pBuffer, dwByteCount );
} while( ! finished );

```

Just as the encryption process required a Boolean key to state when the end of the file had been reached, the CryptDecrypt function also requires this data. However, once the loop is finished, all that remains is to clean up:

```

CryptDestroyKey( hKey );
}
pFileIn->Close();
pFileOut->Close();
if( ! CryptReleaseContext( hProvider, 0 ) )
...

```

This concludes the decryption process. Using the *Crypto* demo program, the Status list reports the operations as shown in Figure 11.7.

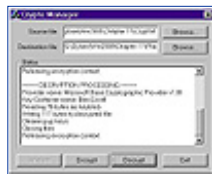


FIGURE 11.7 After decrypting a data file

Okay, you've seen how to set up the cryptographic services, how to encrypt a source file, and how to decrypt it. But what about security problems?

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Inadvertent Compromising of Security

One thing that is always going to be true about security and encryption is that its effectiveness will, in large part, be determined by the care with which it is used. Some of the most elaborate data-protecting schemes can be more easily compromised than simpler ones just because people get lazy (take it for granted) or are insufficiently aware of the weaker points which inescapably exist in every design. Therefore, I'd like to give you some examples of potential pitfalls to be aware of when designing a secure information system.

One of the simpler encryption techniques is based on *exclusive-OR-ing* (XORing) the plaintext data with a key value. If you do this on a file that has lots of blank space or repeated strings of the same character, it's very possible to determine the encryption key, since by simply re-XORing the repeated areas with their own value, the plaintext values cancel out, leaving the key. You might think that with such a weakness, XORing against a key value wouldn't continue to be used. But one advantage of XORing is its extreme speed, as XOR is one of the most fundamental computer operations performed. There are several ways, however, to strengthen an XOR algorithm:

- Use a longer—ideally, an *extremely* long—key value
- Combine additional information, such as the offset position of the character in the file or the value of the preceding character(s), with the plaintext character before encrypting
- Use XOR as only one level of a multilevel encryption scheme

This brings us to two important points. The first is that, surprisingly, even an operation as simple as XOR can be used to create the strongest type of encryption known, provided the key used is completely random and is equal (or longer) in length than the data stream to be encrypted. How you come up with completely random numbers is another whole matter—entire books can and have been written on that subject.

The second point is that combining two or more techniques can result in a much stronger resulting algorithm. Taking this point up to a different level, consider how a system that assigns memorable but unpredictable passwords (or keys) has a significant edge over a system that assigns very difficult to remember passwords. In the second case, it is human nature that a certain number of people will inevitably write down their password on a slip of paper and perhaps even tape it to their monitor!

As a different example on the same theme: consider the advantage one encrypted email has over another, where the one is written using couched phrases, or only implicitly conveys a message, whereas the second

states its content in plain language. The first one stands a good chance of remaining confidential even if the encryption is broken, whereas the second message is cracked at the same point its encryption is breached. So, multiple levels—particularly of different types—can combine synergistically to achieve a more secure result.

Most efforts to break encryption eventually come down to “brute force” approaches—simply trying every possible key value until one “fits.” Encryption’s effectiveness usually depends on the huge range of possible key values to make the chore of guessing take so long as to be impractical. However, many strong encryption algorithms are much stronger in theory than they turn out to be when implemented in software, because the programmer slips up and introduces less randomness in a key value by using pseudo-random (more predictable) computer-generated sources for key values, or parts of key values. Such mistakes result in less work required to crack the resulting encryption, because the possible range of key values has been reduced.

Another, somewhat more sophisticated, version of “brute force” attack uses educated guesses to attempt to narrow down the key value. A password-guesser that starts out by using a person’s first and last names, spouse’s name, pet’s name, and other common sources for passwords is an example of this. In the case of encrypted email, eliminating predictable headers (such as “Dear Sue,”) and footers (“Thanks, Joe”) and compressing blank space (and perhaps also removing the carriage returns and line feeds!) goes a long way toward tightening up the plaintext and making it harder for such attacks to gain an easy foothold.

Remember too that any encryption scheme is only as strong as its weakest link. It does no good to have an extremely fancy encryption scheme if the decryption keys, for example, are not safeguarded. When saving a key value along with encrypted data, how do you encrypt the key itself? It doesn’t hurt to use an even stronger encryption for the key value, and to store the key at a random location in the file, or even vary the length of the key used.

Finally, keep in mind that, as the saying goes, there is more than one way to skin a cat, and if someone cannot get in through the front door, they may resort to the back door, side door, or window, or rip themselves a hole in your system. Consider the case of a sophisticated encryption algorithm that is snugly packed away in a nice DLL, ActiveX control, or some other dynamic (or even static) code library. If a function in this library takes passwords or key values, or stores them by calling another function, couldn’t someone simply replace the code library with their own, or (less likely in Windows NT/2000) even intercept or “hook” a function that gets called? The moral is: Be very, very careful at each stage of the design and implementation process, to ensure that you haven’t inadvertently left a gaping hole.

Summary

If all this talk about security and encryption sounds too paranoid for you, then perhaps your system doesn’t really have a need for security. But I’ll bet even then that you’d agree on this point: Security is one area where, if you need it at all, it only makes sense to do it right.

Next, we’ll tackle a less paranoid and perhaps more entertaining subject as we look at the Graphics Device Interface (GDI) and how to determine what capabilities and options are supported by any specific system.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART III

Windows 2000 Graphics and Multimedia

- CHAPTER 12: The Windows Graphics Device Interface
- CHAPTER 13: DirectX and OpenGL Graphics—An Overview
- CHAPTER 14: Multimedia Operations
- CHAPTER 15: Multimedia MMX for the Pentium III

CHAPTER 12

The Windows Graphics Device Interface

- Device-context access
- Information-context accessn Device capability information
- Windows mapping modes
- The viewport versus the window

This chapter provides an introduction to the *Windows Graphics Device Interface (GDI)* beginning with the *device context (DC)*, which is the heart of the GDI. Then it covers the device-context information that can be retrieved. Finally, you'll learn about the Windows mapping modes, as well as the viewport versus window coordinates and sizes.

NOTE:

While it is fairly obvious that graphics play an important—or even essential—role in developing windows applications, due to limitations in space, we will not be giving comprehensive coverage of graphics in this book. Instead, in this chapter and the next, we will introduce some of the more important basics of graphics before proceeding to some of the more esoteric and advanced topics in subsequent chapters. However, to supplement the gap between the basics and the advanced topics, you will find under the heading *Graphics Utilities* on the CD extensive supplementary material as well as demo programs with source code for a variety of graphics

The Device Context

Unlike in DOS, where applications simply own the entire display, printer, or other device, under Windows, the output device is a shared resource. The device context is a method of permitting peaceful coexistence between applications sharing the resource.

Device-Context Handles

Before a Windows application can draw anything to the screen, the application must begin by gaining access to the GDI, which means obtaining a handle to the device context. Asking Windows for a handle to a device context (hDC or hdc) is the equivalent of asking permission to share the output resource. And, in like fashion, including the handle in subsequent calls to GDI output functions not only tells Windows which output device is being addressed, but also assures the GDI that permission has been granted to access the shared device.

The hdc is not only a handle to the device context, but also to a virtual device that has been allocated to the application. The virtual device provides a private space for the application's own display operations. Furthermore, the device-context handle is not only a pointer to a virtual space, but is also a pointer to a data structure detailing device (graphic) settings.

NOTE:

After the contents of the virtual display are mapped by the GDI to the physical display, the virtual display can be discarded, at least until further screen operations are needed.

For example, when a call is made to the TextOut function, font and color information are not included in the call because attributes have already been set for the device context (default settings are provided if no other selections are made). Thus, TextOut requires only string data and coordinates as parameters for each output operation. In like fashion, other device contexts used by other applications have their own attribute settings, which are independent of the current device context.

Device-Context Handles in MFC

Using MFC-based programming, you usually handle screen updates and drawing operations only in response to calls to the OnPaint function (even when you generate these calls indirectly using an Invalidate instruction). You usually expect the MFC shell to pass you a handle to the device context. Or, more accurately, it passes a pointer to a CDC class instance, which you subsequently pass on to your specialized paint or drawing routines.

However, don't become too complacent about expecting this level of service and thereby lapse into not attempting any graphics operations except when you have been passed a pointer to the device context class. Remember that whether you use a handle to the device context or an instance of the CDC class, the device context is available at any point. You are not required to "wait" for permission to draw! You can, instead, demand permission.

Why Query a Device Context?

The attribute settings for a device context can include color palettes, mapping modes, alignment settings, and so on. These settings can be queried or changed by calling other API functions, called with the appropriate hdc parameter to identify the proper device context.

By querying a device context, you can get various information, such as the following:

- Information about supported palettes (colors)
- Font-sizing information (what fonts are available for display and in which sizes)
- Available printer fonts
- Printer resolutions (for optimizing bitmapped images)
- The presence or absence of special hardware capabilities

Accessing the Device Context

To access the device context, you use either the BeginPaint or GetDC functions to retrieve the hdc. It's important

to emphasize that no application should attempt to hold onto a device context beyond the immediate operation. Under Windows, device-context handles are a limited resource and must be shared among executing applications. Use the `EndPaint` or `ReleaseDC` functions when you're finished with the device-context handle.

NOTE:

In like fashion, if you're using an instance of the `CDC` class rather than the `BeginPaint` or `GetDC` function to access a device handle, you should also conclude by closing the instance when finished. However, when you're using a class instance, the device context is released automatically by the class destructor when the class goes out of scope. By all means, use the convenience inherent in the class, but keep in mind what is happening behind the scenes.

A device context should always be released or deleted as soon as the immediate output operation is completed. Of course, once the device-context handle is released, the device context itself is no longer valid.

The PAINTSTRUCT Structure

To obtain a handle to the device context, the `BeginPaint` function is called with two parameters, and when finished, the `EndPaint` function accepts the same pair.

```
hdc = BeginPaint( hwnd, &ampps );  
...  
EndPaint( hwnd, &ampps );
```

Here, two calling parameters are included:

- The `hwnd` argument identifying the application accessing the device context
- A pointer to the `ps` structure, a structured variable of type `PAINTSTRUCT`, which contains information specifying which portion of the screen should be repainted (`rcPaint`), whether the background (existing image) should be erased (`fErase`), and so on

The `PAINTSTRUCT` structure is defined in `WinUser.H`:

```
typedef struct tagPAINTSTRUCT  
{ HDC      hdc;  
  BOOL     fErase;  
  RECT     rcPaint;  
  BOOL     fRestore;  
  BOOL     fIncUpdate;  
  BYTE     rgbReserved[32];  
} PAINTSTRUCT, *PPAINTSTRUCT, *NPPAINTSTRUCT, *LPPAINTSTRUCT;
```

The `BeginPaint/EndPaint` functions are commonly used in response to `WM_PAINT` messages when only a portion of a screen should be updated (which is faster than repainting an entire screen).

In similar fashion, the `GetDC` and `ReleaseDC` functions are used when operations are required over the entire client window area.

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Other Device-Context Accesses

There are several other methods of accessing the device context:

GetWindowDC Unlike **GetDC**, provides access to the entire window, including the window frame, caption, and menu bar. Access is restricted, however, to the current application's window.

CreateDC Obtains a device context that allows operations outside an application's client window area, such as for a screen-capture utility.

CreateIC Obtains a handle to an information context as opposed to a device context. An information context, unlike a device context, does not provide a handle for output operations. Instead, the handle returned can be used to obtain information about an output device without actually outputting anything. The use of **CreateIC** is described in the next section.

Acquiring an Information (Device) Context

In most cases, when access to a device such as the video display is required, the **GetDC** function is invoked to return a handle (**hDC**) to the device context. In the *DC* (for *Device Capacity*) demo described in this chapter, however, two device-context handles are declared: **hDC** and **hDCInfo**. Since either handle could be used for either purpose, this is slightly redundant. On the other hand, using two handles makes it clear that two quite different device contexts are being used: a conventional *hardware* device context and an *information* device context.

The difference between these two contexts is simple. Unlike the conventional device context, which is used for output, the information context, provided by **CreateIC**, allows you to access information about the context but does not provide any output capabilities. The information context requires less overhead and is sufficient for retrieving information.

The first requirement in using the **CreateIC** function is to determine the device for which you are requesting information. For the video display, the video device is accessed using the default specification "DISPLAY".

```
hdc = CreateIC( "DISPLAY", NULL, NULL, NULL );
```

TIP:

For Windows 2000 Win32-based applications, you can call `CreateIC` or `CreateDC` by identifying the driver using the null-terminated string "DISPLAY" and entering the remaining arguments as `NULL`. You may also specify a device context for a printer using the string "WINSPool" (WinNT/2000 only).

Under Windows 3.x, the `WIN.INI` file, normally found in the Windows directory, contains printer device information that can be extracted with the `GetProfileString` function and used to create an information device context. Under Windows NT/2000 (and 95/98), most of the information previously stored in the `WIN.INI` file has been moved to the registry, a less-readily readable storage format. Therefore, the printer device context is no longer available in this fashion and must be accessed via a different route.

The string information needed to describe the printer can be returned by calling the `EnumPrinterDrivers` and `EnumPrinters` procedures. With this information, the `CreateIC` function and subsequent processes are carried out in the same fashion as demonstrated for the display device in the *DC* demo.

The example includes a simple routine to query the system and return a list of printer devices. Figure 12.1 shows an example of the Printer Selection dialog box displayed by the *DC* demo.



FIGURE 12.1 The Printer Selection dialog box displayed in the *DC* demo

NOTE:

Examples of the `EnumPrinterDrivers` and `EnumPrinters` procedures can be found in the `EnumPrt.C` program (part of the *Printer.C* demo) distributed with the Visual C++ toolkit.

The Printer Selection dialog box is called in a fashion slightly different than that used for most dialog boxes. In the `WndProc` procedure, under the `msg/WM_COMMAND` response, you'll find the following:

```
case IDM_PRINTER:
    CheckMenuItem( hMenu, IDM_DISPLAY, MF_UNCHECKED );
    CheckMenuItem( hMenu, IDM_PRINTER, MF_CHECKED );
```

The first two provisions are simple. They show which type of device is being selected by making sure that the Display menu option is unchecked and that the Printer menu option is checked.

The third provision creates the dialog box and passes a reference to the `PrinterProc` procedure, which contains the handler and the provisions to query the available printers. If the `DialogBox` procedure returns `TRUE` (meaning that a selection has been made), then the program invalidates the client window so that it will be redrawn with new information.

```
if( DialogBox( hInst, "PRINTER", hwnd, PrinterProc ) )
    InvalidateRect( hwnd, NULL, TRUE );
break;
```

The important part at this point is the `PrinterProc` procedure.

```
BOOL APIENTRY PrinterProc( HWND hDlg,   UINT msg,
                          UINT wParam, LONG lParam )
{
    DWORD cbPrinters = 4096L, cbNeeded, cReturned;
    char  szPrinter[40] = "\0";
    int   i, j;

    switch( msg )
    {
```

The first step in handling the query is to initialize the dialog box, beginning by allocating a buffer to hold the information we're about to request.

```

case WM_INITDIALOG:
    if( ! ( gpPrinters = (PPRINTER_INFO_1)
        LocalAlloc( LMEM_FIXED | LMEM_ZEROINIT,
                    cbPrinters ) ) )
    {
        ErrorMsg( "gpPrinters local alloc failed." );
        return( FALSE );
    }

```

After allocating the buffer, and assuming success, we proceed by issuing a query requesting that the system printers be enumerated.

NOTE:

The DC demo program passes PRINTER_ENUM_LOCAL as the first argument for the EnumPrinters function, restricting the selection to local devices and excluding network printers. If you would prefer to query a remote printer, refer to the EnumPrinters function documentation for options and flags.

```

    if( ! EnumPrinters( PRINTER_ENUM_LOCAL, NULL, 1,
        (LPBYTE) gpPrinters, cbPrinters,
        &pcbNeeded, &pcReturned ) )
    {

```

Note that we have also anticipated the failure of this query, giving full consideration to the possibility that the buffer provided will not be large enough.

```

        if( GetLastError() == ERROR_INSUFFICIENT_BUFFER )
        {

```

However, if the buffer isn't big enough, the response is simple: Free the allocated buffer and then allocate a new buffer using the size requirement reported by the original query.

```

            LocalFree( (LOCALHANDLE) gpPrinters );
            gpPrinters = (PPRINTER_INFO_1)
            LocalAlloc( LMEM_FIXED | LMEM_ZEROINIT, cbNeeded );
            cbPrinters = cbNeeded;

```

And, after reallocating the buffer using the new information, we query the printer list a second time.

```

            if( ! EnumPrinters( PRINTER_ENUM_LOCAL, NULL, 1,
                (LPBYTE) gpPrinters, cbPrinters,
                &pcbNeeded, &pcReturned ) )
            {
                ErrorMsg( "Can't enumerate printers" );
                return( FALSE );
            }
        }
    else
    {
        ErrorMsg( "Can't enumerate printers" );
        return( FALSE );
    }
}

```

The queries conclude with report provisions to cover complete failure—just in case. Assuming that one of our queries has been successful, we check the cReturned count. If cReturned is zero, we simply report that there are no printers installed, and then return.

```

    if( ! cReturned )
    {

```

```
        ErrorMessage( "No printers installed" );  
        EndDialog( hDlg, FALSE );  
        return( FALSE );  
    }  
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

Otherwise, since at least one printer has been reported, we copy the reported printers to the dialog box's list box and then set the current selection to the first item (with the zero index).

```
for( i=0; i<(INT)cReturned; i++ )
    SendDlgItemMessage( hDlg, IDM_PRINTLIST, LB_ADDSTRING, 0,
        (LPARAM) gpPrinters[i].pDescription );
SendDlgItemMessage( hDlg, IDM_PRINTLIST, LB_SETCURSEL,
    0, 0 );
return( TRUE );
```

The final provision occurs when the OK button is clicked. (No provisions have been included to handle a double-click on a selection.)

```
case WM_COMMAND:
    switch( LOWORD( wParam ) )
    {
        case IDOK:
            // LB_GETCURSEL message retrieves the index of
            // currently selected item, if any, in a single-
            // selection list box
            i = SendDlgItemMessage( hDlg, IDM_PRINTLIST,
                LB_GETCURSEL, 0, 0 );
            // LB_GETTEXT message retrieves a string from a
            // list box
            SendDlgItemMessage( hDlg, IDM_PRINTLIST, LB_GETTEXT,
                (LPARAM) i,
                (LPARAM)(LPCTSTR) szPrinter );
            // Parse the string into device, driver and port
            // names
            j = 0;
            i = -1;
            while( szPrinter[++i] != ',' )
```

```

        gszDeviceName[i] = szPrinter[i];
        gszDeviceName[i] = '\0';
        while( szPrinter[++i] != ',' )
            gszDriverName[j++] = szPrinter[i];
        gszDriverName[j] = '\0';
        j = 0;
        while( szPrinter[++i] != '\0' )
            gszPort[j++] = szPrinter[i];
        gszPort[j] = '\0';
        EndDialog( hDlg, TRUE );
        return( TRUE );

```

Clicking the OK button simply copies the selection information to the local buffers for further use before closing the dialog box. If the user does not make a selection, and provisions are included for a default selection, the dialog box will refuse to close.

Alternatively, if the Cancel button is clicked, the dialog box simply closes without supplying a printer specification.

```

        case IDCANCEL:
            EndDialog( hDlg, FALSE );
            return( FALSE );
    }
    break;
}
return 0;
}

```

Once you have retrieved a device specification, we have a wealth of information available—probably more than you’re likely to want at any one time.

NOTE:

As a general rule, access to a printer-specific information context shouldn’t be necessary. One of the strengths of Windows (any version) is being able to let the system handle device-specific details, such as the resolution and capabilities of a printer.

Device-Context Information

Device-context information describes the resolution and capacities of hardware output devices, such as the video display, a printer or plotter, a camera device, or even a metafile (a file of GDI function calls in binary format).

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

GDI Identifier Constants

The device context contains a variety of information that can be requested via the identifier constants defined in WinGDI.H. Many of these inquiries return integer values; others return word values composed of bit-flags that can be broken down, again using flag constants defined in WinGDI.H. Most of these constants are demonstrated in the DC demo. Table 12.1 summarizes the GDI information indexes and flag constants.

TABLE 12.1: GDI Information Indexes and Flag Constants

GDI Information Index and Flags Driver Version and Device Types

DRIVERVERSION
TECHNOLOGY
DT_PLOTTER
DT_RASDISPLAY
DT_RASPRINTER
DT_RASCAMERA
DT_CHARSTREAM
DT_METAFILE
DT_DISPFILE

Description

Device driver version
Device classification
Vector plotter
Raster display
Raster printer
Raster camera
Character-stream, PLP
Metafile, VDM
Display file

Device Dimensions

HORZSIZE
VERTSIZE
HORZRES
VERTRES
ASPECTX
ASPECTY
ASPECTXY

Description

Horizontal size (millimeters)
Vertical size (millimeters)
Horizontal width (pixels)
Vertical width (pixels)
Pixel width
Pixel height
Pixel hypotenuse

LOGPIXELSX	Pixels/logical inch (width)
LOGPIXELSY	Pixels/logical inch (height)
DESKTOPHORZRES ¹	Virtual desktop width (pixels)
DESKTOPVERTRES ¹	Virtual desktop height (pixels)
VREFRESH	Current vertical refresh rate in CPS (Hz)

Device Color Capabilities	Description
BITSPIXEL	Bits per pixel (color)
PLANES	Color planes
NUMBRUSHES	Device-specific brushes
NUMPENS	Device-specific pens
NUMMARKERS ⁵	Device-specific markers
NUMFONTS	Device-specific fonts
NUMCOLORS	Device-specific colors
SIZEPALETTE	Entries in physical palette
NUMRESERVED	Reserved entries in palette
COLORRES	Actual color resolution
SHADEBLENDCAPS	Shading and blending capabilities identified by following values:
SA_CONST_ALPHA	Handles SourceConstantAlpha member of BLENDFUNCTION structure
SA_GRAD_RECT	Supports gradient fill rectangles
SA_GRAD_TRI	Supports gradient fill triangles
SA_NONE	Not supported
SA_PIXEL_ALPHA	Supports per-pixel alpha in AlphaBlend
SA_PREMULT_ALPHA	Supports premultiplied alpha in AlphaBlend

Miscellaneous	Description
PDEVICESIZE	Device descriptor size required

Printer-Related Device Capabilities ²	Description
BLTALIGNMENT	Preferred horizontal drawing alignment in pixels (a value of 0 indicates an accelerated device with no alignment preferences)
PHYSICALWIDTH	Width (device units)
PHYSICALHEIGHT	Height (device units)
PHYSICALOFFSETX	x-margin, printable area
PHYSICALOFFSETY	y-margin, printable area
SCALINGFACTORX	x-axis scaling factor
SCALINGFACTORY	y-axis scaling factor

Graphics, Image, and Font-Handling Capabilities	Description
CURVECAPS	Curve capabilities
CC_NONE	Curves not supported
CC_CIRCLES	Circles
CC_PIE	Pie wedges
CC_CHORD	Chord arcs
CC_ELLIPSES	Ellipses
CC_WIDE	Wide lines
CC_STYLED	Styled lines
CC_WIDESTYLED	Wide styled lines

CC_INTERIORS	Interiors
CC_ROUNDRECT	Rounded rectangles
LINECAPS	Line capabilities
LC_NONE	Lines not supported
LC_POLYLINE	Polylines
LC_MARKER	Markers
LC_POLYMARKER	Polymarkers
LC_WIDE	Wide lines
LC_STYLED	Styled lines
LC_WIDESTYLED	Wide styled lines
LC_INTERIORS	Interiors
POLYGONALCAPS	Polygonal capabilities
PC_NONE	Polygonals not supported
PC_POLYGON	Polygons
PC_RECTANGLE	Rectangles
PC_WINDPOLYGON	Winding polygons
PC_SCANLINE	Scan lines
PC_WIDE	Wide borders
PC_STYLED	Styled borders
PC_WIDESTYLED	Wide styled borders
PC_INTERIORS	Interiors
TEBTCAPS	Text capabilities
TC_OP_CHARACTER	Character output precision
TC_OP_STROKE	Stroke output precision
TEBTCAPS,	Text capabilities
TC_CP_STROKE	Stroke clip precision
TC_CR_90	Character rotation, 90 degree
TC_CR_ANY	Character rotation, any angle
TC_SF_X_YINDEP	x/y independent scaling
TC_SA_DOUBLE	Double scaling
TC_SA_INTEGER	Integer scaling
TC_SA_CONTIN	Continuous scaling
TC_EA_DOUBLE	Doubled character scaling
TC_IA_ABLE	Italics
TC_UA_ABLE	Underline
TC_SO_ABLE	Strikeout
TC_RA_ABLE	Raster fonts
TC_VA_ABLE	Vector fonts
TC_RESERVED	Reserved, must be 0
TC_SCROLLBLT	Scroll using bltblock transfer is <i>not</i> supported
CLIPCAPS	Clipping capabilities
CP_NONE	Output clipping not supported
CP_RECTANGLE	Output clipped to rectangle
CP_REGION	Output clipped to region
RASTERCAPS	Raster capabilities
RC_NONE ⁴	Raster capabilities not supported
RC_BANDING	Banding support required
RASTERCAPS,	Raster capabilities
RC_BIGFONT ⁵	Fonts larger than 64KB supported
RC_BITBLT	Capable of transferring bitmaps (standard BitBlt)
RC_BITMAP64	Bitmaps larger than 64K supported

RC_DEVBITS ⁵	Supports device bitmaps
RC_DI_BITMAP	Supports SetDIBits and GetDIBits functions
RC_DIBTODEV	DIBitsToDevice support
RC_FLOODFILL	FloodFill support
RC_GDI20_OUTPUT	Supports Windows version 2.0 features
RC_GDI20_STATE ⁵	Includes a state block in the device context
RC_OP_DX_OUTPUT	Supports dev opaque and DX array
RC_PALETTE	Palette support
RC_SAVEBITMAP ⁵	Capable of saving bitmaps locally
RC_SCALING	Scaling supported
RC_STRETCHBLT	StretchBlt support
RC_STRETCHDIB	StretchDIBits support

¹The DESKTOPHORZRES and DESKTOPVERTRES values may be larger than the HORZRES and VERTRES values if the device supports a virtual desktop or multiple displays.

²These replace the appropriate Escapes from earlier versions.

³Effectively clipping capabilities are now universal and these queries are no longer needed.

⁴If RC_NONE is not defined in WinGDI.H, do not attempt to use this constant (it really isn't necessary anyway).

⁵A number of RASTERCAPS queries are now essentially obsolete and, while they may still be supported as queries, can simply be presumed to be supported without question.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The DC Demo: Reporting Device Capabilities

The *DC* demo demonstrates how device (hardware) information can be retrieved for the Windows device drivers. This demo uses a single function, `GetDeviceCaps`, to retrieve a list of integer values describing specific device capabilities. In many cases, the information returned is an integer value, such as the supported vertical or horizontal pixel size. In other cases, the value returned is a flag value that must be deciphered, as individual bits, to identify the presence or absence of specific capabilities.

NOTE:

In the *DC* demo, you will find that the information available has been grouped in related categories, principally for display convenience. Conventionally, these values would not be displayed at all; they would be requested as needed for use by an application. Or, far more often, this information would not be requested at all, except by the GDI to determine how to best map an application's display (or other output) to the available device. In general, neither the programmer nor the application should be concerned with any aspect of the physical device, leaving these to be handled by the Windows GDI.

`GetDeviceCaps` is called with two parameters: a device-context handle (`hDC`), which identifies a specific device such as a video display or printer, and an integer argument, which specifies the information requested.

In most cases, the capabilities reported by the `GetDeviceCaps` function are provided by graphics coprocessor devices that are incorporated into the output devices themselves. In a video display, this would be a coprocessor found on the video card. In hardcopy devices, the presence or absence of a graphics coprocessor depends partially on the age of the device and partially on the sophistication (and price) of the model. For example, older laser printers, though still capable of excellent print quality, often predate the development of graphics coprocessors. Other, more modern devices, such as ink-jet printers, may lack not only coprocessors but also sufficient memory for full-page graphics and, therefore, may require banding (downloading page images in bands) support provided by Windows.

NOTE:

The *DC* demo is included on the CD accompanying this book, in the Chapter 12 folder.

Device Palette (Color) Capabilities

The basic capabilities reported for the video display are shown in Figure 12.2. Figure 12.3 shows the capabilities reported for a printer device.



FIGURE 12.2 An example of CRT device information (XGA)



FIGURE 12.3 An example of printer device information

In general, when speaking of device palette capabilities, the first thought that comes to mind is of video display capabilities. And, in most cases, this is accurate. However, hardcopy devices that support palettes with more than two colors (black and white) are becoming increasingly popular. Therefore, even if we tend to speak of palettes as if we were referring only to the video device, remember that the video display is not the only color device available. It is, however, generally the most sophisticated device.

As a general rule, the *DC* demo reports video palettes in bits per pixel (BITSPIXEL) and color planes (PLANES). In most cases, one of these two elements will be reported as value 1, and only the remaining element is relevant in determining the device's supported color range. This limitation (or context, if you prefer) is mostly a matter of how the hardware's device driver is written and not a matter of how the actual physical device operates.

The *DC* demo also reports device colors (NUMCOLORS), which can be useful information. Figure 12.2 shows the device capabilities reported by the *DC* demo for a Matrox Pulsar video device that (deliberately, because of requirements set by other system uses) is restricted to a 256-color palette. Table 12.2 shows how the reported results vary according to equipment capabilities for different types of devices.

TABLE 12.2: Comparing Device Color Capabilities

Device Type	Color Planes	Bits/Pixel	Calculated Palette Size	Reported Palette Size	Possible Colors
EGA/VGA	4	1	16	16	256
SVGA	1	8	256	20	16,777,216
HighColor ¹	1	16	65,536	none	16,777,216
TrueColor ¹	1	24	16,777,216	none	16,777,216
Printer	1	1	2	2	2
Plotter	1	1	2	8	N/A

¹ HighColor and TrueColor color resolutions are independent of screen resolution size.

You may note a few discrepancies in the table, such as the fact that the SVGA palette size is calculated as 256 colors while the reported palette size is only 20. No, this is not an error; the reported palette size is the number of palette entries that are predefined, not the number of possible (calculated) palette entries. Thus, for an SVGA system, the palette size is 256 color entries, of which 20 have been reserved by the system (Windows) for standard palette colors. The remaining 236 palette entries are undefined and must be defined either by the application or, in some cases, by the user.

Likewise, for the HighColor video device at 16-bits-per-pixel, and TrueColor video at 24-bits-per-pixel, even though calculated palette sizes come out to 65,536 and 16,777,216, there is no reported palette simply because no palette is used. Instead, each pixel on the screen receives its own 16-bit or 24-bit color specification.

Another discrepancy appears in the entries for the plotter device, where a calculated palette size of 2 is at odds with the reported palette size of 8. This discrepancy occurs because the plotter reported in this table is a pen-carousel plotter, which has eight color pens but still has only one color plane and one bit per pixel. For the plotter, a specific instruction is required to select a color pen, and all subsequent instructions (until another pen is selected) are essentially monochromatic.

NOTE:

When you are reviewing device information, keep in mind that the information reported by the system for a specific device can be useful, but you must understand how these facts are derived and reported.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Version and Device Types

The DRIVERVERSION and TECHNOLOGY index arguments request the device driver version number and the device type. Driver version numbers are reported as word values in hexadecimal format. For example, a version number reported as 0x0103 identifies version 1, revision 3.

Seven device types are defined: vector plotter, raster display (CRT), raster printer (laser jet or ink jet), raster camera, character stream, metafile, and display file. Normally, of course, the video device will be a raster display. Depending on the equipment and configuration, however, the reported hardcopy device could be a vector plotter, raster printer, raster camera, or even a metafile or display device.

Size and Resolution

Device size and resolution cover a variety of elements, including physical size, pixel or dot resolutions, colors and palettes, and pixel or dot aspect ratios—in effect, all salient information about a device's reproduction capabilities. Table 12.3 shows the size and resolution data for four devices.

TABLE 12.3: Comparing Basic Device Resolutions

Data	Millennium II AGP/ NEC MultiSync LCD 1510V*	Panasonic KX-P6500	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Width (mm)	216	203	203	203
Width(pixels/dots)	1024	2400	2400	1440
Height (mm)	162	267	266	260
Height (pixel/raster lines/dots)	768	3150	3150	1846
Horizontal (pixel/dot aspect)	36	300	300	180

Vertical (pixel/dot aspect)	36	300	300	180
Diagonal (pixel/dot aspect)	51	425	425	255
Pixels per inch/dots per inch (horiz)	120	300	300	180
Pixels per inch/dots per inch (vert)	120	300	300	180
Color (bits per pixel)	16	1	1	1
Color (planes)	1	1	1	1
Device brushes	-1	-1	-1	-1
Device pens	-1	100	10	40
Device fonts	0	0	0	0
Device markers	0	4	0	0
Device colors	-1	2	2	8
Palette entries	0	0	2	16
Palette entries (reserved)	0	0	2	16
Color resolution (bits per pixel)	0	0	0	0

* *Some of these values depend on the monitor rather than the video card used.

Notice that the numbers shown do not always tell the entire story. For example, the PaintJet printer reports 1 color bit per pixel, 1 color plane, and 8 device colors; but, finally and correctly, it reports a 16-color palette.

TIP:

What the PaintJet report does not tell you is that these 16 palette colors can be assigned from a much wider range using RGB specifications.

Raster Capabilities

Because most output devices are raster devices (yes, this includes laser printers), RASTERCAPS reports on general device capabilities, such as banding (device memory dependent), bitmap handling, fill operations, scaling, and device palettes. Table 12.4 shows raster capabilities for some different devices.

TABLE 12.4: Comparing Raster Capabilities

RASTERCAPS	Matrox Millennium II APG video card	Panasonic KX-P6500	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Banding support required		ü		
Bitmap transfer	ü	ü	ü	ü
Bitmaps e 64K	ü	ü	ü	ü
SetDIBits and GetDIBits	ü	ü		
SetDIBitsToDevice	ü	ü		
Floodfill				
Windows 2.0 features	ü	ü		
Palettes				
Scaling				
Fonts e64KB	ü	ü		
StretchBlt	ü	ü	ü	ü
StretchDIBits	ü	ü	ü	ü

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Raster capabilities typically receive more sophisticated support from video devices than from printers. For either type of device, support may be partially determined by the amount of memory available on the video card or in the printer. Because all of the printers used for the example have more than adequate memory for full-page graphics, no banding support is required for any of these. In like fashion, the video card represented has 4MB of RAM and supports large bitmaps as well as large fonts, even though, in general, large fonts are not sent directly to display devices (normally, bitmapped images of text are transferred instead).

TIP:

The choice between downloading a font versus sending a bitmapped text image is determined by the printer type and, of course, the printer driver. Happily, in normal circumstances, this decision does not need to be made by the programmer. If you are designing printer drivers, however, some experimentation may be necessary to decide when and where the trade-offs occur.

Bitmap Operations

In the GetDeviceCaps function, bitmap operations are reported as capabilities under the heading of raster capabilities. From a graphics operation standpoint, however, these operations are also probably the most important device capabilities. Bitmaps inherently require more than a little handling to write to the screen or to move, superimpose, or manipulate in any other fashion.

Characters, whether bitmapped or stroked, are commonly written to the screen as foreground images only, and therefore generally leave the greater portion of the screen unaffected. A rough estimate suggests that 20 percent or less of the screen pixels are actually written during text operations. Because of this, character operations are inherently faster than bitmapped image operations, which require writing all pixels within the image area.

To offset this speed discrepancy, sophisticated graphics coprocessors include special hardware functions that are devoted to fast bitmap transfers, handling of bitmaps larger than 64KB, and bitmap-scaling operations. If any of these capabilities are supported, most common graphics operations are executed at much higher speeds than they can be with video equipment (or printers) that does not support these functions.

Clipping Capabilities

Clipping capabilities, which are reported by the CLIPCAPS request, define the ability of a device to clip drawing instructions to a specified region. Most devices have rectangular clipping capabilities. Region-clipping capabilities, permitting a more complex area definition, are also possible. Table 12.5 shows clipping capabilities for several devices.

TABLE 12.5: Comparing Clipping Capabilities

CLIPCAPS	Matrox Millennium II APG video card	Panasonic KX-P4400	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
No output clipping support				
Output clipped to rectangle	ü	ü	ü	ü
Output clipped to region				

Curve Capabilities

The curve capability flags (CURVECAPS) report the capabilities of the output device to handle its own curve definitions. These may include circles, arcs, pie wedges, and ellipses, as well as special borders. Table 12.6 shows the curve capabilities of several devices.

TABLE 12.6: Comparing Curve Capabilities

CURVECAPS	Matrox Millennium II APG video card	Panasonic KX-P4400	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Curves not supported				
Circles	ü		ü	ü
Pie wedges			ü	ü
Chord arcs			ü	ü
Ellipses	ü		ü	ü
Wide borders			ü	ü
Styled borders			ü	ü
Wide and styled borders			ü	ü
Interiors	ü		ü	ü
Rounded rectangles				

Line Capabilities

Line capability flags (LINECAPS) report on line-style support and some interior-fill operations. As with curve capabilities, line capabilities are widely (if not universally) supported. Although line capabilities are less calculation-intensive than curve operations, providing external support for line-drawing calculations still increases throughput for both display and hard-copy devices. Table 12.7 shows the line capabilities of some devices.

TABLE 12.7: Comparing Line Capabilities

LINECAPS	Matrox Millennium II APG video card	Panasonic KX-P4400	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Lines not supported				
Polylines	ü	ü	ü	ü
Markers			ü	ü
Polymarkers			ü	ü
Wide lines			ü	ü

Styled lines	ü	ü	ü	ü
Wide and styled lines			ü	ü
Interiors			ü	ü

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Polygon Capabilities

Polygon capability flags (POLYGONALCAPS) report on device capabilities for executing alternate and winding fill operations on polygon figures, as well as plain and styled borders and interiors.

NOTE:

A winding fill operation is slower than an alternate fill but also fills enclosed areas that might otherwise be omitted.

As with curve and line capabilities, polygon support is common but not universal. In terms of the calculations required, support for polygon capabilities probably falls somewhere between those for curves and those for lines, although the fill algorithms may run a close second to curves in terms of the CPU's workload. Again, external support for these tasks does increase throughput and display or output speed. Table 12.8 shows the polygon capabilities of some devices.

TABLE 12.8: Comparing Polygonal Capabilities

POLYGONALCAPS	Matrox Millennium II APG video card	Panasonic KX-P4400	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Polygonals not supported				
Alternate fill polygon			ü	ü
Rectangles	ü		ü	ü
Winding fill polygon			ü	ü
Scan lines	ü	ü	ü	ü
Wide borders			ü	ü
Styled borders			ü	ü
Wide and styled borders			ü	ü
Interiors	ü		ü	ü

Text Capabilities

Text capability flags (TEXTCAPS) report device capabilities for character output, including stroked output, character rotation and scaling, and clipping precision, as well as italic, boldface, underlining, and strikeout. Also reported are the handling capabilities for both raster and vector fonts.

In general, printers provide more sophisticated text capabilities than video cards. For either type of device, the text source may be more important than the device itself. A WYSIWYG (“what you see is what you get”) editor, such as Microsoft Word or CorelDraw, may provide character rotation regardless of the display or printer support. If the output device provides such support, the result is simply faster output operation, not new or different capabilities. Table 12.9 shows the text capabilities for some devices.

NOTE:

Note that these are device capabilities, not font characteristics. For font characteristics, refer to the LOGFONT structure for a specific font.

TABLE 12.9: Comparing Text Capabilities

TEXTCAPS	Matrox Millennium II APG video card	Panasonic KX-P4400	HP LaserJet IID Printer	HP PaintJet XL Printer (standard)
Character output precision			ü	ü
Stroke output precision			ü	ü
Stroke clip precision	ü		ü	ü
90-degree character rotation				
Any character rotation				
Independent x/y scaling				
Doubled character scaling				
Integer multiple scaling				
Any multiple scaling				
Double-weight characters				ü
Italic				
Underlining			ü	ü
Strikeout			ü	ü
Raster fonts	ü			
Vector fonts			ü	ü

Mapping Modes

Under DOS, all graphics operations (with some rare exceptions) use a single mapping mode in which the logical unit is the screen pixel. The screen origin—the 0,0 point—is located in the upper left corner, which is a convention established by and held over from text display and video memory-mapping conventions. Although applications might create both text and graphics windows, the use of this same mapping-mode convention was almost as inevitable as the sun rising in the east. Things have changed under Windows.

Windows Mapping Modes

Windows supports eight separate mapping modes, each providing a different set of conventions appropriate for different circumstances. Another difference from the mapping mode used by DOS is that the modes (with the exception of MM_TEXT, which corresponds to the DOS standard) use the lower left corner—not the upper left—as the default screen origin when they do not use a variable origin point. The eight mapping modes under Windows are listed in Table 12.10.

TABLE 12.10: Windows Mapping Modes

Mapping Mode	value	Logical Units	Default Origins	
			x-axis	y-axis

MM_TEXT	1	Pixel	Left	Top
MM_LOMETRIC	2	0.1 mm	Left	Bottom
MM_HIMETRIC	3	0.01 mm	Left	Bottom
MM_LOENGLISH	4	0.01 inch	Left	Bottom
MM_HIENGLISH	5	0.001 inch	Left	Bottom
MM_TWIPS	6	1/1440 inch	Left	Bottom
MM_ISOTROPIC	7	Variable($x=y$)	Variable	Variable
MM_ANISOTROPIC	8	Variable ($x!=y$)	Variable	Variable

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Default Text Mode

Beginning with the familiar, the MM_TEXT mode (default) corresponds to the DOS graphics mode, allowing the application to operate in terms of pixel positions (virtual or physical device pixels). In the MM_TEXT mode, the default origin lies at the upper-left corner of the screen with the x- and y-axis coordinates increasing to the right and down. Figure 12.4 illustrates the MM_TEXT mode.



FIGURE 12.4 The MM_TEXT mode

The MM_TEXT mode is used by default if no other mapping mode has been selected. As its name suggests, this mode is optimized for text displays—that is, for text displays following European/American conventions with the text flowing from left to right and top to bottom. Your applications are not required to change modes to perform their specific tasks. This means that any operation can be accomplished in MM_TEXT mode, if you are willing to perform the necessary coordinate conversions yourself, just not always as conveniently. Convenience is no small consideration and is the real reason for providing such a variety of mapping modes.

The Cartesian Coordinate Modes

The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS modes follow the familiar Cartesian coordinate system, with coordinate values increasing up and to the right of the origin point. These five modes differ in their logical units. They provide measurements appropriate to applications that need to draw in physically meaningful units: English, metric, or typesetting units.

The MM_HIMETRIC and MM_LOMETRIC modes use logical units of 0.01mm and 0.1mm, respectively, to provide high- and low-resolution imaging. Figure 12.5 shows an example of the MM_LOMETRIC mode. Similarly, the MM_HIENGLISH and MM_LOENGLISH mapping modes use units of 0.01 and 0.001 inch, again providing high and low resolution. Together, these four modes adequately provide for a wide variety of circumstances to suit the needs of engineers, scientists, and artists. (Scaling, of course, can be applied as necessary.)



FIGURE 12.5 The MM_LOMETRIC mode

The fifth mapping mode, MM_TWIPS, requires a bit more explanation. The logical units are 1/1440 inch, which (while unfamiliar to an engineer or scientist) just happens to be 1/20th of a printer's point (72 points = 1 inch). Using MM_TWIPS mode, a 10-point typeface can be drawn 200 logical units in height, providing sufficient detail to render even the most complex typeface, regardless of whether the actual display device can support a similar resolution.

The Isotropic and Anisotropic Modes

The MM_ISOTROPIC and MM_ANISOTROPIC modes provide both variable logical units and variable origin points. They can be adapted to provide any scale or arrangements that might arise and that are not covered by one of the standard modes.

For example, using the MM_ANISOTROPIC mode, where the x- and y-axes can be assigned different scales, the vertical direction might be mapped in thousands of dollars (or the numerical equivalent) and the horizontal direction could represent dates to create a business graph. Equally practical, the MM_ISOTROPIC mode could be used with measurements defined as astronomical units with the 0,0 origin at the center to plot the orbital path of a comet on its passage through the solar system. In both cases, the GDI would provide conversion from the internal modal units to the actual physical display, but would not require these conversions to be handled by the application.

Remember, however, that the MM_ISOTROPIC mode has the same logical units assigned to both the x- and y-axes. This mode is useful when preserving the exact shape of an image is important. Figure 12.6 shows an example of the MM_ISOTROPIC mode. The MM_ANISOTROPIC mode may have each axis scaled differently. Furthermore, both modes may have the origin point located anywhere within the application window or even somewhere outside the application window entirely.



FIGURE 12.6 The MM_ISOTROPIC mode

Getting and Setting Mapping Modes

The SetMapMode and GetMapMode functions are provided to set or retrieve the eight defined mapping modes.

```
SetMapMode( hdc, nMapMode );
nMapMode = GetMapMode( hdc );
```

The nMapMode argument used to call SetMapMode would, normally, be one of the predefined constants, but it could be simply an integer argument in the corresponding range. The value returned by SetMapMode reports the previous (existing) mapping mode.

As with SetMapMode, the value returned by GetMapMode will correspond to one of the defined mapping modes, or if there is an error, will return as zero. Until a different mapping mode is explicitly selected, the initial (default) mode will always be MM_TEXT, mapped in pixels with the 0,0 origin at the upper-left corner of the application window, just as in DOS.

After a different mapping mode is selected, the values returned by SetMapMode and GetMapMode will correspond to that mode. For example, after MM_LOMETRIC is selected, the origin point is at the lower-left corner of the application window, with coordinates increasing up and right. At the same time, instead of specifying coordinates in pixels, coordinates are specified in 0.1mm increments, which on the average SVGA monitor provides a full-screen vertical resolution of 1560 units and 2080 units horizontally (assuming the 208mm × 156mm display reported by the DC demo).

Remember, however, that the actual physical display mode, assuming standard SVGA resolution, is only 640 pixels wide and 480 pixels high. This means that, horizontally and vertically, 3.25 logical units are mapped to each physical unit, or in overall terms, a total of 10.56 (3.25²) logical pixels are mapped to each physical pixel.

Table 12.11 shows the logical size and corresponding inches and millimeters for several output devices and modes, arranged in descending order of magnitude.

TABLE 12.11: Device and Mode Resolutions

Device	Logical Size	Unit	Inches	Millimeters
EGA video (H/V)	640 / 350	Pixels	0.012795 / 0.017547	0.3250 / 0.4457*
VGA video (H/V)	640 / 480	Pixels	0.012795	0.3250
MM_LOENGLISH	0.01	Inches	0.010000	0.2540
SVGA video (H/V)	1024 / 768	Pixels	0.079960	0.2031
MM_LOMETRIC	0.1	MM	0.003940	0.1000
LaserJet printer	300	DPI	0.003333	0.0846
MM_HIENGLISH	0.001	Inches	0.001000	0.0254
High-resolution SVGA	1280 / 1024	Pixels	0.008515	0.0191
MM_TWIPS	1/20	Points	0.000694	0.0176
Typesetter	2000	DPI	0.000500	0.0127
MM_HIMETRIC	0.01	MM	0.000394	0.0100

*The physical measurements (inches and millimeters) for video modes are based on standard monitor screen sizes as specified by the manufacturers with a display area approximately 10.9 inches wide and 7.9 inches high (a nominal 14-inch monitor). Because actual physical units may vary, depending both on monitor adjustments and on larger physical screen sizes, the values shown are for comparative resolutions only.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Coordinate Translations: Viewport versus Window

Mapping the logical, virtual display to the physical display is not the application's concern, because all relevant operations are handled by the GDI. What is the application's concern—or, more properly, the programmer's concern—is selecting the mapping mode appropriate to the task's requirements, and then setting the origins, window, and viewport extents accordingly.

Under Windows, each application is contending with two separate coordinate systems that, combined, form the client window display. These two reference systems have been given the titles *viewport* and *window*—a nomenclature that is responsible for more than a little confusion.

The term *window* is used to refer to the application's client window, meaning the area of the screen where the application's display actually appears. And, for an even longer time, the term *viewport* has been used, particularly in non-Windows contexts, to refer to a clipping region or a boundary restricting graphics operations.

Under Windows, and specifically when referring to mapping modes, the term *viewport* does indeed refer to the screen display where the physical device coordinates must and do apply. When referring to *viewport* coordinates on the screen, the reference is in terms of pixels, and the origin is always at the upper-left corner (just like under DOS).

The term *window*, however, refers to the virtual space where the application is executing its drawing operations. The window coordinates are expressed in logical coordinates. Furthermore, all GDI functions accept coordinates as logical, not physical, units.

The mapping modes are simply a description of how Windows (the system) translates coordinates from the *window*, which is using logical coordinates, to the *viewport*, which uses device (hardware) coordinates. The conversion from window coordinates (*xWindow,yWindow*) to viewport (screen) coordinates (*xViewport,yViewport*) uses two formulas:

$$xViewport = xViewOrg + ((xWindow - xWinOrg) * \frac{xViewExt}{xWinExt})$$

$$yViewPort = yViewOrg + ((yWindow - yWinOrg) * \frac{yViewExt}{yWinExt})$$

The additional parameters used in these formulas are the variables describing the selected mapping mode:

x_ViewOrg/y_ViewOrg Describe the viewport origin in device or screen coordinates

x_WinOrg/y_WinOrg Describe the window origin in local, logical coordinates

x_ViewExt/y_ViewExt and **x_WinExt/y_WinExt** Define a conversion ratio between “size” in virtual units of the window and an equally fictional “size” for the viewport, expressed in device units

If both the viewport and window were using the same scalar coordinate systems (pixel units) and origin points (0,0 at upper left), these formulas could be greatly simplified as:

$$\begin{aligned} xViewPort &= xViewOrg + xWindow \\ yViewPort &= yViewOrg + yWindow \end{aligned}$$

However, such a simplification would destroy the very flexibility that Windows has introduced by using mapping modes (although, essentially, this relationship is found in the default MM_TEXT mode).

Also, when an application does require conversion from its logical coordinate context to its physical device context (or vice versa), Windows provides two functions: DPTOLP and LPtoDP. The DPTOLP function converts device points to logical points:

```
DPTOLP( hdc, &points, 1 );
```

The Points argument is an array of POINT structures to be converted. The hdc argument specifies the device context, and the third parameter indicates how many points should be converted.

The reverse process, from logical points to device points, is accomplished by calling the LPtoDP function, which has the same arguments:

```
LPtoDP( hdc, &points, 1 );
```

As another example, assume that an application needs to convert the client window rectangle to device coordinates. This would be accomplished as:

```
GetClientRect( hwnd, &rect );
LPtoDP( hdc, (LPPPOINT) &rect, sizeof(rect) / sizeof(POINT) );
```

TIP:

Remember that most of the time, Windows will provide all the translation required between the viewport and window coordinate systems without application provisions. But if you need to handle these, these functions are available.

Setting Window and Viewpoint Origins

Under Windows, all mapping modes have variable origin points, even though six of these modes conventionally use the default origins indicated in Table 12.10. These defaults can be changed by using the SetViewportOrgEx and SetWindowOrgEx functions.

The SetViewportOrgEx function sets the viewport (device context) origin and is invoked as:

```
SetViewportOrgEx( hdc, xPos, yPos, &point );
```

The xPos and yPos arguments are in device-context units (pixels for the video device). Point is returned reporting the previous origin coordinates, with the y-coordinate in the high-order word and the x-coordinate in the low-order word. Optionally, this fourth parameter can be passed as NULL if no return value is desired.

The SetWindowOrgEx function provides a similar flexibility within the virtual context and is invoked as:

```
SetWindowOrgEx( hdc, xPos, yPos, &point );
```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

With this function, xPos and yPos are expressed in logical units appropriate to the mapping mode selected. Again, an optional Point returns the previous origin coordinates.

NOTE:

In Windows 3.x, similar services are supplied by the SetViewportOrg and SetWindowOrg functions. However, these have different calling parameters and return values.

Remember that changing the viewport or window origins has no effect on the position of the client window on the screen. Each is, in effect, a fiction providing an adjustable offset that affects how and which portion of an application's display is mapped from the virtual to the physical display. (To change the window position on the screen, use the SetWindowPos function.) Also, in general, only one of these two functions is used at any time, since they perform what are, in essence, redundant functions.

Setting the Window and Viewport Extents

The SetWindowExtEx and SetViewportExtEx functions are used to set the window and viewport extents. They set the scale and offset sizes, which provide the mapping ratios (see the formulas shown earlier). These ratios determine how an application's client window image is mapped to the physical (device) viewport context. These two functions are valid only when the mapping mode is set to MM_ISOTROPIC or MM_ANISOTROPIC; they are ignored if any other mapping mode is in effect.

The SetWindowExtEx function specifies the window size in logical units and is called as:

```
SetWindowExtEx( hdc, xSize, ySize, &oldSize );
```

The xSize and ySize parameters specify the window extent (size) in logical units. The fourth parameter (OldSize) is returned with the previous window extent. If the original settings are not needed, this argument can be passed as NULL, and nothing will be returned.

The SetViewportExtEx function is called in similar fashion to specify the viewport extent in device units.

```
SetViewportExtEx( hdc, xSize, ySize, &oldSize );
```

Two constraints apply when the MM_ISOTROPIC mapping mode is in effect:

- SetWindowExtEx must be called before calling SetViewportExtEx.
- The ySize parameter is ignored by both functions in favor of the xSize setting, keeping the aspect ratio constant.

In MM_ANISOTROPIC mode, the xSize and ySize parameters can be specified independently, setting different scales along each axis.

The ratios between the x- and y-viewport extents and the x- and y-window extents define how much the GDI should stretch or compress units in the logical coordinate system to fit units in the device (physical) coordinate system. In isotropic mode, the x- and y-axes always maintain the same ratios; in anisotropic mode, these ratios can be determined independently.

As an example, consider an application window using the anisotropic mode with the window x-extent set to 200 and the viewport x-extent at 400. In this situation, the GDI would map two logical units (along the x-axis) to four device units. Continuing with the assumption that the window y-extent is also set at 200 and the viewport y-extent at 100, the GDI would map two logical units to one device unit along the y-axis.

However, since no provision has been made to alter this, the y-axis in the window increases from bottom to top; the same y-axis in the viewport increases from top to bottom. The result is that the image mapped is, in effect, inverted. Of course, if this is not desired, either the viewport or window y-axis can be assigned a negative value, thus inverting the inversion. On the other hand, if both are negative, the situation remains unchanged.

The Modes Demo: Mapping Mode Effects

The *Modes* demo demonstrates how the mapping modes affect the window's size (in logical units) as well the window's origin point. Figure 12.7 shows the *Modes* main screen.



FIGURE 12.7 The *Modes* demo

NOTE:

The *Modes* demo is included on the CD accompanying this book, in the Chapter 12 folder.

The main menu offers three selections:

Modes This option displays a submenu listing the eight mapping modes.

Origin This mode offers the origin choices of upper left, center, or lower left.

Aspect When the isotropic or anisotropic mode has been selected, this option displays the dialog box shown in Figure 12.8, which allows you to change the horizontal and vertical viewport and window extents. With other modes, this option is grayed.

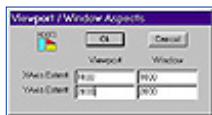


FIGURE 12.8 The Viewport/Windows Aspects dialog box in the *Modes* demo

The *Modes* demo demonstrates two menu-handling procedures that you may find useful. The first checks and unchecks menu items according to selection; the second changes the grayed state of the Aspects menu item, enabling this selection only when the isotropic or anisotropic mapping modes are selected.

Summary

The demos described in this chapter, *DC* and *Modes*, provide an introduction to the GDI. Because the *DC* demo is intended simply to show how device capabilities can be queried, this program is provided only in a conventional Windows C-code format; no MFC/C++ equivalent is provided. Although actual applications might query specific device capabilities for the purpose of setting some internal handling, they would not normally report these capabilities to the user.

NOTE:

For additional information and examples of using graphics API capabilities refer to Chapters S11 through S20 in the Graphics Utilities section of the CD accompanying this book.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 13

DirectX and OpenGL Graphics—An Overview

- Using DirectX components
- Attaining high visual quality and performance with OpenGL
- Fahrenheit: The next generation for graphics?

While most programmers will find conventional graphics capabilities adequate for their applications, there are circumstances and applications that demand high performance graphics operations. While multimedia and game programs may be the first examples to come to mind, other examples include computer-aided design (CAD) programs that commonly use three-dimensional representation—whether as wire-frame outlines or by depicting surfaces—to display shapes. In like fashion, scientific modeling programs represent molecular shapes in 3-D or depict flow conditions, thermal gradients, or neutron flux using complex graphic representations. If these examples are too esoteric, simply consider the weather maps on the evening news where 3-D cloudscape follow synthesized surface contours. (The infamous and ubiquitous “dancing baby,” of course, is simply one more example of a computer-animated graphic.)

While a variety of software packages have been and are used to create various types of graphics, the two principal tools for the PC platform today for high-end graphics are *DirectX* and *OpenGL*.

NOTE:

This chapter offers only a brief introduction to both DirectX and OpenGL with a few notes on where examples and further information can be found. Discussing either API in any depth or even discussing the principal features using development examples would require a generously sized book, while providing complete coverage for both could not be encompassed in less than two volumes. Fortunately, however, there are a number of excellent manuals available for both DirectX and OpenGL development.

DirectX

DirectX, part of Microsoft’s Visual Studio/Visual C++, is a group of COM-based objects (COM is discussed

in Chapters 27 through 29) offering a consistent interface through the Windows operating system to the hardware capabilities of a PC. (In contrast, OpenGL is platform-independent.)

The principal advantage of DirectX is that its APIs and functions allow the developer to concentrate on application features without becoming embroiled in the details and differences between the hardware configurations of target platforms. By using DirectX, applications become not only independent of the platform's video accelerator chipset but also independent of whatever network, sound, mouse, joystick, or other peripherals may be present.

DirectX is composed of seven principal components, discussed in the following sections.

DirectDraw

DirectX's *DirectDraw* component offers direct access to the platform's video hardware. Provisions include allowing image data to be stored in either video or system memory and access to hardware page flipping and blitting capabilities. Other encapsulated graphic functionality includes video buffers (surfaces), clippers, video overlays, and palettes.

Direct3D

DirectX's *Direct3D* component is the big item in gaming circles, where developers rely on Direct3D capabilities to generate complex visual objects and effects.

The Direct3D component is similar to the DirectDraw component, providing direct access to the video 3-D accelerator capabilities. Direct3D offers two different modes: *Immediate mode*, a low-level interface for game developers, and *Retained mode*, a high-level interface for multimedia developers.

DirectSound

DirectX's *DirectSound* component provides direct access to the platform's sound hardware. DirectSound offers a primary sound buffer for mixing and secondary buffers for storing sound data. DirectSound can be used to mix buffer contents or custom sound-mixing algorithms can be written with direct output to the primary sound buffer.

In support of virtual reality or spatial modeling, three-dimensional directional sound positioning is supported through the *IDirectSound3DBuffer* and *IDirectSound3DListener* interfaces.

DirectMusic

DirectMusic, introduced with DirectX 6.0, provides access to platform music capabilities, going beyond the functions supported by the DirectSound component. DirectMusic includes direct support for the MIDI file format.

DirectInput

DirectX's *DirectInput* component provide direct access to such input devices as the mouse and keyboard as well as joystick and force-feedback devices popular with gamers (see the Diff1 example provided in Visual Studio 6.0). Additional support for newer game devices is expected.

DirectPlay

DirectX's *DirectPlay* component is used to include network support for applications and provides handling for TCP/IP, IPX, serial, and modem communications. In addition, DirectPlay offers "lobbies" (see the Bellhop demo provided in Visual Studio 6.0) that allow users to connect with each other while concealing the underlying mechanisms and protocols.

DirectSetup

DirectX's *DirectSetup* component offers simple provisions to add DirectX setup functionality to an installation program. Combined with Autoplay—which automatically loads a specific program from a CD—DirectSetup can be used as the basis for an installation program.

Direct3D Retained Mode

DirectX's *Direct3D Retained Mode* provides support for real-time, three-dimensional graphics allowing real-time manipulation for 3-D images.

DirectAnimation

DirectX's *DirectAnimation* provides support for diverse media types including 2-D vector graphics, 3-D graphics, sprites, audio, and video, and a uniform time and event model. DirectAnimation is a COM API with an underlying runtime engine.

DirectShow

DirectX's *DirectShow* is an architecture controlling and processing streams of multimedia data through custom or built-in filters. DirectShow can also use the set of media streaming interfaces to stream media data without creating filters.

DirectShow uses video and audio hardware cards supporting the DirectX APIs allowing users to play digital movies and various sound formats (includes MPEG, AVI, MOV, and WAV formats).

DirectX Transform

DirectX Transform makes it possible to add image filters and transitions by providing a set of rules about how to process the inputs to produce output image; this output can be a modified version of a single input or a combination of multiple inputs. DirectX Transform enhances simple animations by replacing frame animation with rule-based animation.

DirectX and the Pentium III

DirectX (version 6.1) offers new support for calculating the geometry and lighting of 3-D objects by routing these tasks to the Pentium III. The Pentium III processor's new multimedia instructions, the *Internet Streaming SIMD Instructions* (previously *Katmai New Instructions*; see Chapter 18, "Multimedia MMX for the Pentium III"), are optimized for geometry and lighting tasks. After processing, the Pentium III passes its calculations to the system's graphics chip for rendering.

DirectX also includes support for 3DNow!, the multimedia instructions in the newest AMD K6 CPUs.

DirectX 7.0

With the release of DirectX 7.0, a number of new features have been added.

These include:

- A new type library for Visual Basic supporting DirectX features
- Direct3D Immediate Mode API offers new features including:
 - Device-state blocks
 - Geometry blending
 - Cubic environment mapping
 - User-defined clipping planes
- Direct3DX Utility Library—functions as a helper layer simplifying common 3-D graphics tasks and sits on top of Direct3D and DirectDraw.
- DirectMusic supporting the Downloadable Sounds level 2 standard
- DirectInput supports:
 - Eight mouse buttons
 - Exclusive keyboard access
 - Delayed start for force-feedback devices
 - Read/write for force-feedback effect files
 - Force Editor application for creating force-feedback effects
- DirectSound with improved hardware voice management
- DirectSound with improved 3-D audio processing algorithms and improved CPU utilization
- DirectDraw adds stereo support for active devices
- DirectPlay provides improved ripple launching

- Improved documentation with dynamic filtering for language-specific information
- Additional sample applications

Playing with DirectX

The Visual Studio 6.0 set offers a wide variety of DirectX example programs. More than 40 examples are located on MSDN CD 1—refer to DirectX in the Help index to copy the demo programs to your hard drive for experimentation.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

These examples range from relatively simple animations using multiple bitmap images (see the Donut example where 60 images are contained in a single bitmap file) to complex generated images (see the Duel example that uses robotic pugilists). Also included are examples showing how to read system capabilities (see Figure 13.1 from the DXView example), how to play .WAV files (see the DSShow example), how to apply textures to a surface (see Figure 13.2 from the Globe example), and how to morph shapes (see the Morph example).

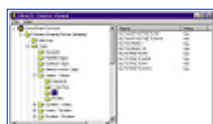


FIGURE 13.1 DirectX Device Capabilities using DXView

A couple of caveats are in order here. First, while most or all of the examples will compile as is, quite a few of the examples require media files to execute. For example, the Globe demo requires the Sphere3.x (x of mesh surface) file to provide the spherical surface and the Tex1.ppm (portable pixel map) file to provide the texture map shown in Figure 13.2.

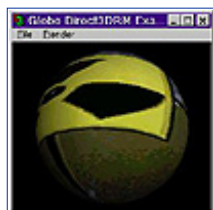


FIGURE 13.2 Globe applies texture to a sphere

While these image and mesh files (as well as a number of .WAV and .BMP files) are included on the CD, all of these are found in a \Media subdirectory. To use the example programs, you may either copy the appropriate media files to the application directory or include the media directory in your path.

As a second caveat, a few of the demo programs will execute only if special equipment—such as a force-feedback joystick—is present. While this is understandable, less comprehensible is the reason why many of the demos terminated after reporting that a 640x480 display could not be generated or that DirectDraw initialization failed.

Finally, a few of the demos appear to function ... but do nothing of any particular interest.

Still, in this potpourri of examples, it should be possible for you to explore and discover some suggestion of the capabilities provided by DirectX.

DirectX versus OpenGL

In the gaming world, competition between Microsoft's DirectX and the SGI-initiated open standard OpenGL is every bit as hot and heavy as the action in today's popular video games. The competition, however, is not exclusively between the two 3-D APIs, since there are other chipsets (hardware video) supporting their own specific APIs.

While OpenGL has been popular for many years, Microsoft engineers are openly criticized for three characteristics of DirectX:

- DirectX is an attempt to reengineer the wheel but to do so in a proprietary format (that is, to make it Windows compatible only, while OpenGL is platform independent).
- DirectX lacks capabilities that have been available in OpenGL for several generations even though DirectX version 7 has added new features.
- DirectX is difficult to use and badly documented, while OpenGL is widely supported, well documented, and easy to develop. Again, DirectX version 7 offers improvements over previous versions, but the criticisms remain valid.

OpenGL

Since OpenGL was first introduced in 1992, it has become the most widely used and supported API for 2-D and 3-D graphics applications, functioning across a wide variety of computer platforms. Providing a broad base of rendering, texture mapping special effects, and powerful visualization functions, OpenGL's ability to support innovative applications and to foster rapid application development has made this a favorite with programmers. Among the most important aspects of OpenGL is that developments can be carried out across a variety of desktop and workstation platforms rather than being limited to the Windows operating system.

While OpenGL—like ActiveX—is popular with game developers, OpenGL is also widely used for more serious applications, such as:

- **CAD/CAM** See, for example, EdgeCAM 4.0 from Pathtrace Engineering (www.pathtrace.com)
- **Flight training** See IFR Trainer 3.0 from Safeline Systems (www.safeline.com)
- **Real-time simulations** See Total Airspace and Airport Modeller (TAAM) from The Preston Group (www.preston.com)

These are, of course, only a very few of the applications using OpenGL. Links to a wide variety of examples can be found at www.opengl.org. In general, OpenGL has a proven track record, with an enviable series of characteristics, some of which are discussed in the following sections.

High Visual Quality and Performance

OpenGL provides support for any visual computing application requiring maximum performance—from 3-D animation to CAD to visual simulation. Applications can exploit high-quality, high-performance OpenGL capabilities, allowing developers in such diverse markets as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2-D and 3-D graphics.

NOTE:

The graphics features supported by OpenGL and the mechanisms used are similar to those provided by DirectX.

Advantages in OpenGL

From a developer's standpoint, OpenGL has a variety of strengths and advantages:

- OpenGL is an industry standard. Guided by an independent consortium, the OpenGL Architecture Review Board oversees the OpenGL specification, making this the only truly open, vendor-neutral, and multiplatform graphics standard.
- OpenGL has been stable for more than seven years across a wide variety of platforms. Additions to

the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.

- OpenGL applications offer reliability and provide consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system. Supported platforms and operating systems include:
 - All Unix workstations
 - BeOS
 - Linux
 - Mac OS
 - OPENStep
 - OS/2
 - Python
 - Windows 95/98/NT/2000
- Supported windowing systems include:
 - Presentation Manager
 - Win32
 - X/Window System
- OpenGL applications and services can be called from a variety of languages, including:
 - Ada
 - C
 - C++
 - Fortran
 - Java
- OpenGL offers complete independence from network protocols and topologies.
- OpenGL offers scalability with applications that can execute on systems running the gamut from consumer electronics to PCs, workstations, and supercomputers. Consequently, applications can scale to any class of machine that the developer chooses to target.

Disadvantages of OpenGL

The principal disadvantages of OpenGL are few, as contrasted with DirectX, but OpenGL does lack sophisticated support for such peripherals as force-feedback joystick devices. And OpenGL does not, at present, provide 3-D audio, only graphics.

The design of OpenGL, however, does not preclude support for such capabilities. Instead, the OpenGL specification allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. If you have a critical need for 3-D audio or force-feedback joysticks, you could either look around to see if someone is already supporting these devices, or you could look into creating your own as an OpenGL extension.

Or, of course, you could switch to DirectX.

TIP:

For Windows developers, OpenGL is a standard component distributed with all Windows versions. The principal DLL is found as `x:\WinNT\system32\OpenGL32.dll` while header files for Visual C++ can be found under `..Developer Tools\VC98\Include\GL` (or the equivalent directory for versions other than Visual Studio 6.0). Documentation for OpenGL as well as downloadable libraries and a wealth of useful support can be found on the Internet through www.opengl.org. Support for other operating systems such as Linux is also available.

Fahrenheit: The Next Generation for Graphics?

According to current reports, Microsoft and SGI are now cooperating to create a new graphics standard under the code name *Fahrenheit*. To quote from SGI's Web site:

“SGI and Microsoft have formed a strategic alliance to define the future of graphics. The two companies will

collaborate on a graphics development project code-named "Fahrenheit" to provide a common set of low-level and high-level APIs that integrate SGI and Microsoft concepts for visualization and graphics for future versions of DirectX for Windows and SGI platforms. This evolution of the DirectX architecture and APIs will further strengthen the role of Windows as a premier platform for consumer, professional and commercial graphics applications. As a reciprocal component of this alliance, elements of the Fahrenheit technologies will be delivered on current and future SGI hardware systems. Until these technologies are delivered, SGI and Microsoft will work together to support the development of graphics applications for professionals through the OpenGL API and the development of graphics applications for consumers through the Direct3D API."

The end result, assuming Fahrenheit becomes an actual product, may be that both DirectX and OpenGL will be replaced by a single open-standard graphics library/development platform.

Additional information is available from SGI at www.sgi.com/fahrenheit/home.html or through Microsoft's Web site.

Summary

As graphics become increasingly important (and when was the last time you used a purely text application?), DirectX and OpenGL are the two principal contenders to supporting high-end graphics, although a third standard, Fahrenheit, is a potential future contender. Of the two existing standards, OpenGL offers the wider scope for cross-platform compatibility. In contrast, DirectX offers freedom within the Windows environment, but is also widely criticized as another attempt by Redmond to lock the world into Microsoft dependency.

The choice between these two contenders, or the choice to follow the emerging Fahrenheit standard, can be decided only on the basis of your application's needs. However, planning should take into account future needs as well as current requirements.

In the next chapter, we'll look at Windows Multimedia support and devices with particular emphasis on support for sound devices.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 14

Multimedia Operations

- Windows multimedia features
- Commands for playing waveform sounds
- Multimedia control and file operations functions
- A Sound Recorder demo for playing, recording, and mixing sounds

Integration has been a guiding principle behind the design of each version of Windows. Programs can exchange data, send each other messages and commands, and cooperate in combining linked and embedded objects to create compound documents. The various methods of linking communications and services have occupied the past several chapters. Now we conclude with a discussion of multimedia operations, which further the goal of integration by embracing new ranges of data.

Audio and video data in many formats now have a place in the general PC market. For about the price of a large hard disk, you can add a soundboard and a CD-ROM drive to your computer and play audio CDs, watch movies, record sounds to the disk, and create animation sequences. Multimedia seems to promise most for the education and entertainment fields, but business, too, will appreciate the new flexibility and variety of choices in presenting information.

This chapter explains the general support Windows provides for multimedia applications. You will learn the high-level commands for operating any multimedia device and the file I/O commands for reading and writing many kinds of multimedia data. You will also see how to play sounds, record them, change their volume, and combine them.

Windows Multimedia Support

Windows multimedia support is a collection of capabilities for dealing with audio and visual peripherals. Its purpose is to allow the integration of many different data formats in a single system environment. The Windows multimedia features include three different components: audio and visual hardware devices, drivers for those devices, and a generalized API that translates programming commands into instructions for any

multimedia driver.

Multimedia Devices

Multimedia operations are data-intensive; they make great demands on the CPU to process large quantities of data rapidly. The devices that store the data must meet certain function and protocol standards in order to work effectively.

To encourage the proliferation of multimedia-capable systems, Microsoft consulted with a number of hardware and software companies to develop a Multimedia PC specification establishing minimum system capabilities for multimedia computing. Hardware and software products compatible with this standard carry the MPC trademark. For example, to carry the MPC mark, a CD-ROM drive must transfer data at a minimum rate of 150KB per second without utilizing more than 40 percent of the CPU's capacity. It must also have an average seek time of 1 second.

Windows 2000 recognizes the multimedia device types listed in Table 14.1.

TABLE 14.1: Multimedia Device Types Recognized by Windows 2000

Multimedia Type	Device Type
cdaudio	Audio CD player
dat	Digital audio tape player
digitalvideo	Digital video (not GDI) in a window
mmovie	Animation movie files
other	Undefined MCI device
overlay	Overlay device (analog video in a window)
scanner	Image scanner
sequencer	MIDI sequencer
vcr	Controls a VCRdevice
videodisc	Videodisc player
waveaudio	Device that plays digitized waveform sounds

The list of drivers suggests the range of hardware and data formats that Windows now expects to encounter. These types of devices and data were formerly inaccessible to most Intel-based PC users. Besides these drivers, multimedia also brings enhanced display drivers for high-resolution adapters and gray-scale VGA, the Sound Recorder and Media Player applications, and several applets in the Control Panel for installing devices and setting system sounds.

Multimedia Services

Beyond the hardware and drivers, multimedia services include a layer of software defining (predictably) a device-independent interface for programs to use multimedia. For example, a single set of commands will play sounds on any waveform device. Under Windows 2000 (and the other Win32 versions), the multimedia services reside in WinMM.DLL. The layer of Win32 that interprets multimedia commands is called WinMM.

NOTE:

A program that uses the WinMM commands must include the `Mmsystem.H` header file and link with the `Winmm.LIB` library.

Four Command Sets

The system provides four different ways to manage multimedia services: two high-level command sets, one low-level command set, and a set of file I/O commands.

The low-level and the high-level commands control the same multimedia devices. The low-level commands are more powerful, and the high-level commands are more convenient. To record a sound, for example, the low-level functions make you repeatedly send the device an empty buffer and wait for it to come back full. But the low-level functions will also let you mix a new sound with an old sound as you record, scale the pitch

and playback rates, change the device volume setting, record a MIDI song, and send custom messages defined by the driver. Also, since all the high-level commands are implemented internally through the low-level commands, you can get better performance by calling the low-level commands directly.

Low-level commands interact with drivers for particular devices. The more generalized high-level commands interact with drivers for logical devices. Windows 2000 comes with seven: a CD player, a digital-video player, a MIDI sequencer, a VCRplayer, a video overlay player, a videodisc player, and a waveform audio player. These generic drivers translate high-level commands into low-level function calls for particular drivers. The high-level API defined by the generic drivers is called the *Media Control Interface (MCI)*. MCI commands shield you from many small details of managing data streams, but at the expense of some flexibility. The high-level commands give the same kind of control that, for example, `wsprintf` gives for string output. Sometimes you really do need low-level commands like `lstrcat` and `_fcvt`, but the high-level commands are usually much easier to use.

Only specialized programs require the low-level multimedia functions. The high-level functions can play MIDI files, movies, vcxs and videodiscs, and CD-ROMs, and they can record as well as play waveform sounds.

The MCI supports two parallel sets of high-level MCI functions: a command interface and a message interface. Command strings and command messages do the same thing, but strings are useful for authoring systems where the user writes command scripts to control a device. The function `mciSendCommand` sends drivers messages like `MCI_OPEN` and `MCI_PLAY`. The parallel function `mciSendString` sends strings like “open c:\sounds\harp.wav” and “play waveaudio to 500.”

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Besides the low-level commands, the MCI commands, and the MCI strings, a fourth command set facilitates reading and writing with multimedia data files. The *Multimedia I/O (MMIO)* commands understand the organization of files in the standard *Resource Interchange File Format (RIFF)* format (discussed later in this chapter) and also perform buffering, a useful optimization for data-intensive multimedia programs.

The Multimedia Timer

Since timing is often critical in multimedia, particularly for playing MIDI music and for coordinating different devices during a presentation, WinMM also includes enhanced timer services. The multimedia timer does not send WM_TIMER messages; instead, it is based on interrupts. The CPU regularly receives interrupt signals from the computer's timer chip, and the multimedia timer invokes a callback function from your program during those interrupts.

Interrupt-driven timer signals are much more regular because no time is lost waiting for the application's message queue to empty. Furthermore, the multimedia timer is accurate down to about 10 (MIPS) or 16 (Intel) milliseconds, but the smallest effective interval for SetTimer is about 55 milliseconds, and even that resolution isn't guaranteed because of message queue delays. The timer resolution varies from system to system; you can determine the resolution by calling timeGetDevCaps.

NOTE:

The drawback of real-time interrupt processing is that it can significantly slow other applications and degrade system performance.

Multimedia Animation

WinMM includes animation capabilities. By opening an mmmovie device, you can play animation files called *movies*. Multimedia animation does not provide new GDI functions to create moving images. It works very much like the audio device, translating a data file into output. The movie player reads from a RIFF file (one containing RMMP format chunks).

Movie files can support casts, scores, inks, transitions, palette effects, audio, and other animation structures. You open the movie-player device with the MCI_OPEN command message, play the movie with MCI_PLAY,

and finish with MCI_CLOSE. Some other command messages are specific to movie files; MCI_STEP, for example, changes the current position forward or backward a set number of movie frames. MCI_WINDOW sets or adjusts the window where the movie appears. But in general, the MCI commands for movies work the same way as the commands for sound, and when you have learned one set, you can easily learn the other.

Sound Data Formats

Digitized sounds for the PC generally come in one of three common forms. One is the *Compact Disc–Digital Audio* format (also called *Red Book audio*). Commercial CDs use this data-intensive format to store high-quality digitized sound. Each second of sound consumes 176KB of disk space.

Another more compact storage format is defined by the *Musical Instrument Digital Interface (MIDI)*. MIDI is a standard protocol for communication between musical instruments and computers. MIDI files contain instructions for a synthesizer to play a piece of music. MIDI sound files take up less room and produce good-quality sound, but recording them requires MIDI hardware.

A third format, *waveform* files, produces adequate sound without a synthesizer and consumes less disk space than the CD–Digital Audio format. *Waveform audio* is a technique for re-creating sound waves by sampling the original sound at discrete intervals and recording a digital representation of each sample.

To store sound as waveform data, a digitizer measures the sound at frequent intervals. Each measurement forms a snapshot called a *sample*. With smaller intervals and more frequent samples, the sound quality improves. Sound travels in waves; by sampling frequently, you can plot more points on the wave and reproduce it more accurately. WinMM supports three sampling rates: 11.025kHz, 22.05kHz, and 44.1kHz (1kHz equals 1,000 times per second; 44.1kHz is 44,100 times per second). In a .WAV file digitized with a sampling rate of 11.025kHz, each millisecond contains about 11 samples.

The human ear stops perceiving high-pitched sounds when they reach a frequency near 20kHz. For a recording to capture a sound, it must sample at a rate at least twice the frequency of the sound, so a sampling rate of 44.1kHz captures the full range of perceptible frequencies. Commercial audio compact discs sample at 44.1kHz. The lower sampling rates, which are fractions of 44.1kHz, reproduce sound less well but take up less storage room.

Three Easy Ways to Play Sounds

Waveform files conventionally have the .WAV extension. To produce waveform sounds, most programs will rely on one of three simple commands, which work best with short .WAV files:

MessageBeep Plays only sounds configured in the registry for warnings and errors.

sndPlaySound Plays sounds directly from .WAV files or from memory buffers.

PlaySound New in Win32, it resembles sndPlaySound but differs in two respects: It does not play sounds from memory, and it does play sounds stored as resources of type WAVE.

MessageBeep

The MessageBeep command takes one parameter naming one of five system sounds configured in the Control Panel. By pairing every call to MessageBox with a MessageBeep, you can make your program play sounds the user selects to indicate different levels of warnings. If MessageBox displays the MB_ICONHAND icon, MessageBeep should pass MB_ICONHAND as its parameter. The sound produced depends on the SystemHand entry in the system registry.

TIP:

You should allow the user to disable your program's message beeps. If many errors occur, the repeated yellow-alert sirens and broken dishes crashing may become irritating.

Table 14.2 lists the possible parameter values and their corresponding registry entries.

TABLE 14.2: MessageBeep Parameter Values

Parameter Value	Registry Entry
0xFFFFFFFF	Standard beep through PC speaker
MB_ICONASTERISK	SystemAsterisk

MB_ICONEXCLAMATION
MB_ICONHAND
MB_ICONQUESTION
MB_OK

SystemExclamation
SystemHand
SystemQuestion
SystemDefault

Using the Control Panel or the Registry Editor, the user may associate any .WAV file with these signals (see Chapter 8, “Using the Registry,” for more information about the Registry Editor).

Like all the sound functions, MessageBeep requires an appropriate device driver in order to play a waveform sound. The normal PC speaker is not an adequate device for multimedia.

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

sndPlaySound

With the `sndPlaySound` command, you can play any system sound named in the registry and configured from the Control Panel (there may be others besides the standard five), or you can play .WAV files directly.

```
BOOL sndPlaySound( LPCTSTR lpszSoundName,    // file or registry key
                   UINT uFlags                // SND_ option flags
```

The first parameter names a registry entry such as `SystemStart` or `SystemQuestion`; alternatively, it may contain a full path name pointing to a .WAV file. `sndPlaySound` requires enough memory to load the full sound into memory. It works best with sound files no larger than about 100KB.

The second parameter expects a flag controlling how the sound is played. Here are some possible values:

SND_MEMORY Identifies the first parameter as a pointer to an object in memory and not to a filename or system sound.

SND_SYNC Finishes playing sound before returning control to the program.

SND_ASYNC Returns control to the program immediately and plays sound in the background.

SND_ASYNC|SND_LOOP Returns control to the program immediately and plays the sound continuously in the background until the program calls `sndPlaySound` with `NULL` for the first parameter.

SND_NODEFAULT Instructs the function to make no noise at all if for any reason it cannot find or play the sound. Normally, `sndPlaySound` feels obligated to produce a noise of some sort on every call, and if all else fails, it will at least play the `SystemDefault` sound.

SND_NOSTOP If another sound is already playing, the function returns `FALSE` without playing the requested sound.

PlaySound

To play sounds compiled as resources, `PlaySound` is the best choice. (`sndPlaySound` can also play sounds from the program's resources, but only if you load them into memory first and then set the `SND_MEMORY` flag.)

```
BOOL PlaySound( LPCTSTR lpszSoundName,    // file or resource name
```

```
HANDLE hModule,           // source for resource sounds
DWORD dwFlags );         // sound type and option flags
```

The function interprets the first parameter according to the option flags:

SND_APPLICATION Sound is played using an application-specific association.

SND_ALIAS Plays a sound from the system registry. The first parameter is an alias from the registry, such as SystemAsterisk or SystemHand.

SND_ALIAS_ID The pszSound parameter identified a predefined sound.

SND_ASYNC Sound is played asynchronously with PlaySound returning immediately after beginning the sound. To terminate an asynchronously played sound, PlaySound is called with pszSound set as NULL.

SND_FILENAME Plays a sound from a .WAV file, just as sndPlaySound does. The first parameter points to a filename.

SND_LOOP Sound plays repeatedly until PlaySound is called with pszSound set as NULL. The SND_ASYNC flag must be specified to indicate an asynchronous sound event.

SND_MEMORY The pszSound parameter points to an image of a sound in memory.

SND_NODEFAULT No default sound event is used—if the specified sound is not found, PlaySound returns without playing the default sound.

SND_NOSTOP If another sound is already playing, the function returns FALSE without playing the requested sound. If SND_NOSTOP is not specified, PlaySound attempts to stop the current sound so that the device can be used to play the new sound.

SND_NOWAIT If the audio driver is busy, PlaySound returns immediately without playing the sound.

SND_PURGE Terminates sounds for the calling task. If pszSound specifies a particular sound, only that sound is stopped; if pszSound is null, all sounds are stopped. To terminate SND_RESOURCE events, an instance handle must be specified as well.

SND_RESOURCE Plays a sound from a program's resources. The first parameter is a resource ID string, possibly returned from the MAKEINTRESOURCE macro.

SND_SYNC Specifies synchronous playback of a sound event with PlaySound returning after the sound event completes.

The second parameter, hModule, is ignored unless dwFlags includes SND_RESOURCE, in which case hModule identifies the program whose resources contain the WAVE data named in lpszSoundName. The handle may belong to an instance rather than a module and may be acquired, for example, from GetModuleHandle, LoadLibrary, or GetWindowLong.

Windows does not define a WAVE keyword to use in resource files the way you use ICON or BITMAP, but you can always define your own resource types as:

```
<resname> WAVE <filename> // add sound to program's resources
```

<resname> is a name you choose for your resource, and <filename> points to a .WAV file. PlaySound always looks for resources identified as type WAVE.

PlaySound (and MessageBeep and sndPlaySound) is simple to use, but it has limitations. In order to control where in a sound the playback starts, to record sounds, to mix them and change their volume, and to save new sound files, you need more commands. The MCI is the easiest way to program for multimedia, as described in the following section.

Media Control Interface (MCI) Operations

MCI operations take the form of command messages sent to devices. Generally, you begin an operation by opening a device; then you send commands, such as MCI_PLAY or MCI_STOP, to make the device play, stop, record, or rewind; and finally you close the device.

The most important and most versatile of the MCI functions is mciSendCommand. This function is to multimedia what Escape is to printing: a route for sending any of many possible signals to a device. mciSendCommand expects four parameters:

```
MCIERROR mciSendCommand( MCIDEVICEID mciDeviceID,
                          // mm device identifier
```

```
UINT uMessage,          // command message number
DWORD dwFlags,          // flags modifying command
DWORD dwParamBlock );  // info structure
```

The first parameter, mciDeviceID, addresses a particular device. When you open a device, mciSendCommand gives you a device ID; in subsequent commands, the device ID tells Windows where to deliver the message.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

The second parameter, `uMessage`, is a constant like `MCI_PLAY` or `MCI_STOP`. *ShowWave*, the demo described in this chapter, sends the following messages:

MCI_OPEN Opens a device (to begin an interaction)

MCI_CLOSE Closes a device (to end an interaction)

MCI_SET Changes device settings

MCI_PLAY Begins playback

MCI_STOP Interrupts current action

MCI_RECORD Begins recording

MCI_SAVE Saves a recorded sound in a file

Other messages might, for example, make the device pause, seek a location in the device element, or retrieve information about the device.

The third parameter, `dwFlags`, usually combines several bit flags that help Windows interpret the command. The set of possible flags varies for each message, but a few are common to all messages. For example, `mciSendCommand` normally works asynchronously. When it initiates a device operation, it doesn't wait for the device to finish. It returns immediately and the device continues to operate in the background. If you want to know when the operation ends, setting the `MCI_NOTIFY` flag causes WinMM to send you a termination message. For example, you might want to close the audio device when a sound finishes playing. On the other hand, sometimes you don't want to proceed until you are certain the device operation succeeded. The `MCI_WAIT` flag forces the command to run synchronously. Program execution stops at `mciSendCommand` until the device finishes the requested task.

The final parameter for `mciSendCommand`, `dwParamBlock`, also varies from message to message. It is always a structured variable holding either information the device may need to execute the command or empty fields for information the device may return after executing the command. Here are the parameter block data structures needed for the *ShowWave* demo:

Data Structure

`MCI_OPEN_PARMS`

`MCI_SET_PARMS`

`MCI_PLAY_PARMS`

Associated Message

`MCI_OPEN`

`MCI_SET`

`MCI_PLAY`

MCI_RECORD_PARMS
MCI_SAVE_PARMS

MCI_RECORD
MCI_SAVE

The fields of these structures might hold a filename, positions in the file at which to start or stop playing, a device ID, or the address of a callback function to receive the asynchronous completion message. We'll consider each structure in more detail as we encounter it in the *ShowWave* demo.

NOTE:

One consideration shaping the design of the MCI was clearly extensibility. As other devices and other technologies find their way into Windows PCs, the set of MCI command messages and parameter block structures can easily expand to accommodate them. New drivers can define their own messages and structures.

Multimedia File I/O Functions

Multimedia data files conform to the standard RIFF format. Multimedia programmers need to understand the structure of a RIFF file and to learn the MMIO functions for reading and writing them.

RIFF Files

The *Resource Interchange File Format (RIFF)* protocol is a tagged file structure, meaning that a file can be divided into a series of irregular sections marked off by *tags*, or short strings. The tags in RIFF files are four-character codes, such as “RIFF”, “INFO”, and “PAL ”. (The fourth character in “PAL ” is a space.) Each tag begins a *chunk*. The most important chunks begin with the tag “RIFF”. RIFF chunks are allowed to contain other chunks, sometimes called *subchunks*. RIFF files always begin with a RIFF chunk, and all the remaining data is organized as subchunks of the first one.

Every chunk has three parts: a tag, a size, and some binary data. The tag tells what kind of data follows. The size, a DWORD, tells how much data the chunk contains. At the end of the data comes the tag for the next chunk (if any). A waveform file always has at least two subchunks—one for the format and one for the sound data—and may have more. Some chunks might carry copyright and version information; others might hold a list of *cues*, locations in the file that coordinate with events in some other chunk or file.

RIFF chunks differ from most others in that their data fields (in the binary data section) always begin with another four-letter code indicating the file's contents. The “RIFF” tag identifies a RIFF file, and the form code tells you to expect subchunks appropriate for a waveform (“WAVE”), a MIDI sound (“RMID”), a DIB (“RDIB”), a movie file (“RMMP”), or a palette file (“PAL”).

Since RIFF files need so many of these four-character codes, there's a macro for creating them: mmioFOURCC. This command stores a RIFF tag in one field of a chunk-information structure:

```
MMCKINFO mmckinfo.ckid = mmioFOURCC( 'R', 'I', 'F', 'F' );
```

The MMCKINFO structure holds information describing a single chunk. When reading data, the system fills out fields describing the current chunk for you. When writing data, you fill out the information for the system to store.

```
typedef struct _MMCKINFO    /* RIFF chunk information data structure */
{
    FOURCC  ckid;           // chunk ID
    DWORD   cksize;         // chunk size
    FOURCC  fccType;        // form type or list type
    DWORD   dwDataOffset;   // offset of data portion of chunk
    DWORD   dwFlags;        // flags used by MMIO functions
} MMCKINFO;
```

FOURCC is a new data type based on the DWORD type. Each character in the code fills one of the 4 bytes in a DWORD. The third field, fccType, is the form tag we said follows every RIFF tag. The fccType field is irrelevant for non-RIFF chunks because they don't have forms. Figure 14.1 illustrates parent chunks (or superchunks) and subchunks in a RIFF file.

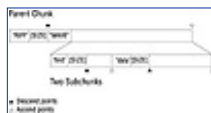


FIGURE 14.1 Structure of a RIFF file

The Multimedia File I/O Functions

You've already encountered two sets of file I/O functions, one in the C runtime libraries and one in the Windows API. The WinMM API includes yet another set, which has special features for chunky RIFF files. The multimedia file I/O functions understand chunks better than other functions do. In addition, they allow buffered file access. (The Windows 2000 system caches file I/O by default, but functions that do their own additional buffering can still improve performance.)

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



► **KEYWORD SEARCH**

GO!

► [Search Tips](#)

► [Advanced Search](#)

► **PUBLICATION LOOKUP**

GO!

Title

► **JUMP TO TOPIC**



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

File Operations

The command that opens a file also controls the buffer settings:

```
HMMIO mmioOpen( LPTSTR      lpzFilename,    // name of file to open
                  LPMMIOINFO lpmmioinfo,    // place to put file info
                  DWORD       fdwOpen );     // option flags
```

The first parameter names the file, and the second stores information about its current state. Unless you want to change a default setting, such as the size of the I/O buffer (8KB), lpmmioinfo should be NULL. The third parameter contains a variety of option flags. Here are some of them:

MMIO_READ Allows reading only

MMIO_WRITE Allows writing only

MMIO_READWRITE Allows reading and writing

MMIO_CREATE Creates a new file

MMIO_DELETE Deletes an existing file

MMIO_EXCLUSIVE Prevents other programs from using the file

MMIO_DENYWRITE Prevents other programs from changing the file

MMIO_ALLOCBUF Enables buffered I/O

In the C libraries, `fopen` begins buffered I/O and `_open` begins unbuffered I/O. The `MMIO_ALLOCBUF` flag makes the same distinction for the multimedia file I/O procedures. The system responds by allocating a default buffer of 8KB. (To make the buffer larger or smaller, set a value in the `MMIOINFO` structure or call `mmioSetBuffer`.)

`mmioOpen` returns a handle of type `HMMIO`, meaning a handle to a multimedia file. Multimedia file handles are not compatible with other file handles; don't use them with the other C or Win32 file functions.

The functions `mmioRead`, `mmioWrite`, and `mmioClose` perform easily recognizable file operations.

Chunk Operations

A few other I/O functions deal specifically with RIFF data chunks. To put a new chunk in a file, call `mmioCreateChunk`. This command writes a chunk header, including the tag, the size, and—for RIFF and LIST chunks—a form code as well. It leaves the file pointer on the byte where you will begin writing the chunk's data with `mmioWrite`.

```
MMRESULT mmioCreateChunk( HMMIO hmmio,          // handle of RIFF file
                          LPMMCKINFO lpmmcki,    // description of new
                                              // chunk
                          UINT uOptions );       // creation options
```

To write a RIFF or LIST superchunk, set an option flag, either `MMIO_CREATE-RIFF` or `MMIO_CREATELIST`.

Moving the file pointer from chunk to chunk calls for `mmioDescend` and `mmioAscend`. *Descending* into a chunk means advancing the file pointer past the tag and size fields to the beginning of the chunk's binary data. *Ascending* from a chunk means advancing the pointer to the end of its data.

```
MMRESULT mmioDescend( HMMIO hmmio,          // handle of RIFF file
                     LPMMCKINFO lpmmcki,    // place to put chunk
                                              // info
                     LPMMCKINFO lpmmckiParent, // optional parent
                     // struct
                     UINT uSearch );        // search option flags
MMRESULT mmioAscend( HMMIO hmmio,          // handle of RIFF file
                    LPMMCKINFO lpmmcki,    // place to put chunk
                    // info
                    UINT uReserved );      // reserved; must be
// zero
```

After each descent, `mmioDescend` returns information about the chunk through the `MMCKINFO` parameter. You can also make `mmioDescend` search for a chunk of a certain type and descend into it. To initiate a search, the last parameter should contain `MMIO_FINDCHUNK`, `MMIO_FINDLIST`, or `MMIO_FINDRIFF`. The search begins at the current file position and stops at the end of the file. `mmioAscend`, besides advancing to the end of a chunk, helps build new chunks. `mmioAscend` is called after you write new data, when it pads the chunk to an even byte boundary and writes the data size in the chunk's header.

The PCMWAVEFORMAT Structure

Every waveform in a RIFF file is required to contain a chunk tagged "fmt". (Lowercase tags indicate subchunks in a larger form.) The *ShowWave* demo reads the format subchunk. The *ShowWave* demo reads the format information to confirm that the sound is playable. It also remembers the sampling rate (`nSamplesPerSecond`) for calculating file positions and scrollbar ranges.

```
/* general waveform format (information common to all formats) */
typedef struct waveform_tag
{
    WORD wFormatTag;           // format type
    WORD nChannels;            // number of channels (1 = mono; 2 = stereo)
    DWORD nSamplesPerSec;      // sample rate
    DWORD nAvgBytesPerSec;     // for buffer estimation
    WORD nBlockAlign;          // block size of data
} WAVEFORMAT;

/* specific waveform format structure for PCM data */
typedef struct pcmwaveformat_tag
{
    WAVEFORMAT wf;
    WORD wBitsPerSample;
} PCMWAVEFORMAT;
```

PCM (pulse control modulation) is the only format category currently defined for .WAV files, so the value in the `wFormatTag` field of a `WAVEFORMAT` structure should be `WAVE_FORMAT_PCM`.

The `PCMWAVEFORMAT` structure adds to the general wave data a single field for bits per sample; this describes the space required to hold the data for a single sound sample. The common values on personal computers are 8 and 16 bits. A monaural wave sound sampled for one second at 11kHz and 8 bits per sample contains 11,000 different samples of 8 bits each, for a total of about 11KB. A stereo waveform samples in two channels simultaneously. If each channel records 8 bits at a time, a single full sample is 16 bits. A one-second 11kHz stereo waveform with a `wBitsPerSample` value of 8 would fill 22KB.

The ShowWave Demo: A Sound Recorder Clone

The *ShowWave* demo imitates the Sound Recorder that comes with Windows 2000 by reading and writing waveform files, playing and recording wave sounds, mixing sounds from several files, and adjusting the volume of a sound. Without a sound card, you can still run the program to open, close, mix, scroll through, and save sound files. However, the program will be deaf

and dumb, unable to play or record anything.

ShowWave is made up of several modules. Here they are in the order in which we'll discuss them:

- Mci.C sends commands to audio device.
- Mmio.C reads and writes waveform data.
- WinMain.C contains WinMain and the About box procedure.
- ShowWave.C responds to dialog box controls.
- GraphWin.C manages the custom control used for the program's graph.

The first two modules, Mci.C and Mmio.C, contain general procedures for performing basic sound operations such as playback and record. WinMain.C registers and creates the program's window and runs the About box. The fourth module, ShowWave.C, calls the appropriate functions in response to input from the user. GraphWin.C manages the sound graph at the center of the program's window, visible in Figure 14.2. When the user scrolls with the scrollbar, the graph display shows the sound wave in different parts of the file. To paint the graph display, we create a custom control and write a window procedure for it. (Another solution would be to subclass a standard control.)



FIGURE 14.2 The ShowWave program

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The *ShowWave* demo is on the CD accompanying this book, in the Chapter 14 folder.

The *ShowWave* Header Files and Resource Script

The *ShowWave* header files and resource script are found on the CD accompanying this book. A few oddities may strike you in the main dialog box's resource template; the MENU keyword hasn't been used previously in a dialog resource but dialog boxes may have menus just as overlapping windows do. Also, the dialog template refers to two custom window classes: GraphClass and ShowWaveClass. Defining new classes for a dialog box control and for the dialog itself makes it possible to assign nonstandard properties to both windows.

The control in the center of the program window represents the current wave sound as a graph. When the user scrolls through the file, the graph changes to represent different parts of the sound. The custom class for the graph window lets you write the window's paint procedure for graphing the sound. The custom class for the main dialog window lets you assign the dialog box an icon to display when minimized.

The MCI Module

In the MCI module, we've isolated all the mciSendCommand function calls and built a separate routine for each command message. All the procedures are short and most follow this basic pattern:

1. Initialize a parameter block.
2. Send a command.
3. Check for errors.
4. Return a value.

The module's eight procedures are described at the top of the Mci.C file:

PUBLIC FUNCTIONS

OpenDevice	open audio device
CloseDevice	close audio device
SetTimeFormat	choose millisecond time format
BeginPlay	begin sound playback
StopPlay	end sound playback
BeginRecord	begin recording
SaveRecord	save recording

Opening and Closing a Device

Opening a device is like opening a file; it announces your intention to exchange information with some piece of hardware and tells the system to create whatever internal structures are needed to manage the interaction. The system gives you a device ID number that, like a file handle, identifies your partner in the exchange. When the interaction ends, you close the device, the system releases any related memory resources, and the device ID becomes invalid.

All multimedia devices respond to the MCI_OPEN and MCI_CLOSE messages. (The other three messages to which all drivers must respond are MCI_GETDEVCAPS, MCI_STATUS, and MCI_INFO, all of which request information about the device.)

Every MCI_OPEN command is accompanied by an MCI_OPEN_PARMS structure, defined in Mmsystem.H:

```
// parameter block for MCI_OPEN command message
typedef struct tagMCI_OPEN_PARMS
{
    DWORD dwCallback;           // window handle
    MCIDEVICEID wDeviceID;      // number identifying device
    LPCTSTR lpstrDeviceType;    // type of device to open
    LPCTSTR lpstrElementName;   // input element for device
    LPCTSTR lpstrAlias;         // optional
} MCI_OPEN_PARMS;
```

The dwCallback field appears in all the parameter structures. It works in tandem with the MCI_NOTIFY flag. Any mciSendCommand function call that asks for a notification message must include a window handle in the low-order word of the dwCallback field. This way, when the operation ends, the system can send an MM_MCINOTIFY message to the window that you named. You'll see how to answer the notification message when we discuss the ShowWave.C module.

The wDeviceID field must be empty when you open a device; WinMM assigns an ID to the device you open and places the ID in the wDeviceID field. After opening any device, you will want to save the ID number.

NOTE:

When Microsoft first released the Multimedia Extensions as a separate product enhancing Windows 3.0, the second and third fields were declared to be type WORD. In Windows 3.1 they changed to the polymorphic type UINT, and in Win32 the ID field changed again to the newly defined MCIDEVICEID type. For backward compatibility, however, both fields still incongruously retain the w prefix. In the transition from Windows 3.1 to Win32, the MCI_OPEN_PARMS structure also lost an unused field, wReserved0.

The lpstrDeviceType field names the sort of device you need for your data. The type name comes from the system registry, where you find entries like these:

```
AVIVideo : REG_SZ : mciavi32.dll
WaveAudio : REG_SZ : mciwave.dll
Sequencer : REG_SZ : mciseq.dll
CDAudio : REG_SZ : mcicda.dll
```

TIP:

To find these multimedia device entries with the Registry Editor, go to HKEY_LOCAL_MACHINE and descend through SOFTWARE to Microsoft to Windows NT to CurrentVersion to MCI and MCI32.

ShowWave requests a device of type WaveAudio in order to play .WAV files.

The lpstrElementName field designates a data source for a compound device. Windows distinguishes between *simple* devices and *compound* devices. A simple device doesn't need the name of a file in order to operate; a compound device does. For example, a program cannot choose what a CD player will play; the player plays only whatever CD the drive contains. A CD player is a simple device. A waveform sound driver, on the other hand, might play any of many different files currently available in the system; you must specify the file. The waveaudio device is always a compound device.

A *device element* is whatever input or output medium your program connects with the device. The device

element is usually a file, so the lpstrElementName field usually contains a filename.

The final field, lpstrAlias, allows you to provide a synonym for naming the device you open. Aliases matter only for the MCI string command interface.

You don't need to fill out all the fields in the parameter block. You might, for example, provide only the element name and let the system choose a matching device by looking at the file's extension—a waveaudio device for a .WAV file, a sequencer for a .MID file. Or if you just want information about the device, you might open it by supplying the type without any element. The flags parameter of the mciSendCommand function tells the system which fields to read. Here is an example:

```
//-----  
//      OPEN DEVICE  
//      Open a waveaudio device  
//-----  
BOOL OpenDevice( HWND hWnd, LPSTR lpszFileName,  
                MCIDEVICEID *lpmciDevice )  
{  
    DWORD dwRet;  
    MCI_OPEN_PARMS mciOpenParms;  
    // open the compound device  
    mciOpenParms.lpstrDeviceType = "waveaudio";  
    mciOpenParms.lpstrElementName = lpszFileName;  
    dwRet = mciSendCommand( 0,                                // device ID  
                           MCI_OPEN,                          // command  
                           MCI_OPEN_TYPE | MCI_OPEN_ELEMENT, // flags  
                           (DWORD)(LPVOID)&mciOpenParms );    // param blk  
  
    if( dwRet != 0 )  
    {  
        ReportMCIError( hWnd, dwRet );  
        return( FALSE );  
    }  
  
    // set return values  
    *lpmciDevice = mciOpenParms.wDeviceID;  
    return( TRUE );  
}
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The first parameter for `mciSendCommand` can only be zero because the device is not open and has not yet been assigned an ID. The third parameter combines two flags. The first, `MCI_OPEN_TYPE`, tells the system to read the `lpstrDeviceType` field of the parameter block because we have put a string there. The second flag, `MCI_OPEN_ELEMENT`, says to read the `lpstrElementName` field as well. Because we have omitted the `MCI_OPEN_ALIAS` flag, the system will ignore any value in the `lpstrAlias` field.

Our `OpenDevice` procedure returns `TRUE` or `FALSE` to indicate its success; if it succeeds, it also returns the device ID in its third parameter. The device ID will be needed for subsequent operations, such as closing the device:

```
//-----  
//  CLOSE DEVICE  
//  Close a multimedia device  
//-----  
void CloseDevice( HWND hWnd, MCIDeviceID mciDevice )  
{  
    DWORD dwRet;  
  
    dwRet = mciSendCommand( mciDevice, MCI_CLOSE, MCI_WAIT,  
                           (DWORD)NULL );  
    if( dwRet != 0 )  
        ReportMCIError( hWnd, dwRet );  
    return;  
}
```

`CloseDevice` expects a device ID as part of its input. No other input is needed; the `MCI_CLOSE` command doesn't even use a parameter block.

Setting the Time Format

When *ShowWave* asks the waveaudio device to play a sound, it always specifies a location in the file from which to begin. With the program's scrollbar, the user can move to any part of the file before starting playback. *ShowWave* and the driver need to agree on units for measuring the file.

The possible units for waveform files are bytes, samples, and milliseconds. Measuring files in bytes makes intuitive sense. As explained earlier in the chapter, a sample is a discrete instant of digitized sound. To measure a file in samples means counting each individual snapshot of the sound wave. Samples are taken at a constant rate, so every millisecond of sound contains the same number of samples. A sound recorded with a sampling rate of 22.5kHz contains about 22 samples in every millisecond. Because milliseconds mean more to most users than do samples or bytes, *ShowWave* chooses the MM_FORMAT_MILLISECONDS format. Choosing a format means sending the MCI_SET command message with the MCI_SET_PARMS parameter block:

```
// parameter block for MCI_SET command message
typedef struct tagMCI_SET_PARMS
{
    DWORD dwCallback;        // window for MM_MCINOTIFY message
    DWORD dwTimeFormat;      // time format constant
    DWORD dwAudio;           // audio output channel
} MCI_SET_PARMS;
```

dwTimeFormat may be MM_FORMAT_BYTES, MM_FORMAT_SAMPLES, or MM_FORMAT_MILLISECONDS. *ShowWave* doesn't play stereo, so we ignore the dwAudio field.

```
//-----
//  SET TIME FORMAT
//  Set time format.  Use milliseconds (not bytes or samples).
//-----
BOOL SetTimeFormat( HWND hWnd, MCIDeviceID mciDevice )
{
    DWORD dwRet;
    MCI_SET_PARMS mciSetParms;
    // set time format to milliseconds
    mciSetParms.dwTimeFormat = MCI_FORMAT_MILLISECONDS;
    dwRet = mciSendCommand( mciDevice, MCI_SET, MCI_SET_TIME_FORMAT,
                           (DWORD)(LPVOID)&mciSetParms );
    if( dwRet != 0 )
    {
        ReportMCLError( hWnd, dwRet );
        return( FALSE );
    }
    return( TRUE );           // success
}
```

The MCI_SET_TIME_FORMAT flag tells the system to read the value in the dwTimeFormat field of mciSetParms.

Playing a Sound

For modularity, PlayBack makes no assumptions about the device settings. It resets the time format before each operation. The MCI_PLAY command initiates playback, and its parameter block is called MCI_PLAY_PARMS:

```
// parameter block for MCI_PLAY command message
typedef struct tagMCI_PLAY_PARMS
{
    DWORD dwCallback;        // window for MM_MCINOTIFY
    DWORD dwFrom;            // starting point
    DWORD dwTo;              // ending point
} MCI_PLAY_PARMS;
```

By default, the Play command starts at the current position in the file and plays to the end, but dwFrom and dwTo, if they are flagged, direct WinMM to start and stop at other points. You may express the starting and stopping points in bytes, samples, or milliseconds, but you should tell the driver in advance which units to expect. (By default, drivers work in milliseconds.)

```

//-----
//   BEGIN PLAYBACK
//-----
BOOL BeginPlay ( HWND hWnd, MCIDEVICEID mciDevice, DWORD dwFrom )
{
    DWORD dwRet;
    MCI_PLAY_PARMS mciPlayParms;

    // set time format to milliseconds
    if( ! SetTimeFormat( hWnd, mciDevice ) )
        return( FALSE );
    // The callback window will be notified with an MM_MCINOTIFY
    // message when playback is complete. At that time, the window
    // procedure closes the device.
    mciPlayParms.dwCallback = (DWORD)(LPVOID) hWnd;
    mciPlayParms.dwFrom = dwFrom;
    dwRet = mciSendCommand( mciDevice, MCI_PLAY, MCI_FROM | MCI_NOTIFY,
                           (DWORD)(LPVOID)&mciPlayParms );

    if( dwRet != 0 )
    {
        ReportMCIError( hWnd, dwRet );
        return( FALSE );
    }
    return( TRUE ); // success
}

```

The MCI_FROM flag signals the presence of a value in the dwFrom field. The MCI_NOTIFY flag tells the system to send a message when the sound stops playing. Sounds can be quite long, so we let WinMM take over and continue to play the sound in the background while *ShowWave* moves on to the next procedure. When WinMM reaches the end of the .WAV file, it addresses an MM_MCINOTIFY message to the window named in the dwCallback field. Look for an MM_MCINOTIFY message handler when we reach *ShowWave_WndProc*. It is the completion routine for asynchronous multimedia operations.

The notify message won't arrive until after the wave device reaches the dwTo point or the end of the file. In its wParam, the message carries a result code indicating whether the operation finished normally, was interrupted or superseded by another command to the device, or failed from a device error. The low word of the lParam carries the device ID.

Stopping a Sound

The MCI_STOP command interrupts an operation already in progress. If the user begins playing a long sound and then decides not to listen after all, clicking on the Stop button sends an MCI_STOP command to abort the playback. Like the MCI_CLOSE message, the MCI_STOP command uses no parameter block.

```

//-----
//   STOP PLAY
//   Terminate playback
//-----
void StopPlay( HWND hWnd, MCIDEVICEID mciDevice )
{
    DWORD dwRet;

    dwRet = mciSendCommand( mciDevice, MCI_STOP, MCI_WAIT,
                           (DWORD)NULL );

    if( dwRet != 0 )
        ReportMCIError( hWnd, dwRet );
    return;
}

```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

If you sent MCI_STOP with MCI_NOTIFY instead of MCI_WAIT, the window procedure would receive *two* notification messages. The first, MCI_NOTIFY_ABORTED, would tell you that playback ended before reaching the terminal point. The second, MCI_NOTIFY_SUCCESSFUL, would indicate successful completion of the Stop command.

Recording a Sound

Soundboards generally include a jack so you can plug a microphone directly into your computer and record straight to disk. The MCI_RECORD message directs the sound device to accept input from a microphone.

```
// parameter block for MCI_RECORD command message
typedef struct tagMCI_RECORD_PARMS
{
    DWORD dwCallback;    // window for MM_MCINOTIFY
    DWORD dwFrom;        // starting point
    DWORD dwTo;          // ending point
} MCI_RECORD_PARMS;
```

The dwFrom and dwTo fields name points in an existing file where the recorded information should be written. In a new file, only the dwTo field matters; new files must always begin at zero. Without the MCI_TO flag and a dwTo value, recording continues until either the disk fills up or the driver receives a Stop command. (To get a new file, give MCI_OPEN a null string, "", for the element name.)

```
//-----
//   BEGIN RECORD
//-----
BOOL BeginRecord( HWND hWnd, MCIDEVICEID mciDevice, DWORD dwTo )
{
    DWORD dwRet;
    MCI_RECORD_PARMS mciRecordParms;

    // set time format to milliseconds
```



```

if( ! SetTimeFormat( hWnd, mciDevice ) )
    return( FALSE );
// Begin recording for the specified number of milliseconds.
// The callback window will be notified with an MM_MCINOTIFY
// message when recording is complete. At that time, the window
// procedure saves the recording and closes the device.
mciRecordParms.dwCallback = (DWORD)(LPVOID) hWnd;
mciRecordParms.dwTo = dwTo;
dwRet = mciSendCommand( mciDevice, MCI_RECORD, MCI_TO | MCI_NOTIFY,
                        (DWORD)(LPVOID) &mciRecordParms );

if( dwRet != 0 )
{
    ReportMCIError( hWnd, dwRet );
    return( FALSE );
}
return( TRUE ); // success
}

```

Saving a Recorded Sound

The MCI_SAVE command instructs a driver to save the current recording to disk. If you record and then close the application without sending MCI_SAVE, the data will be lost.

```

// parameter block for MCI_SAVE command message
typedef struct tagMCI_SAVE_PARMS
{
    DWORD dwCallback; // window for MM_MCINOTIFY
    LPCTSTR lpfilename; // name of disk file
} MCI_SAVE_PARMS;

```

The string in lpfilename names the output file.

```

//-----
//  SAVE RECORD
//  Save recording
//-----
BOOL SaveRecord( HWND hWnd, MCIDEVICEID mciDevice, LPSTR lpszFileName )
{
    DWORD dwRet;
    MCI_SAVE_PARMS mciSaveParms;

    // Save the recording to the specified file. Wait for
    // the operation to complete before continuing.
    mciSaveParms.lpfilename = lpszFileName;
    dwRet = mciSendCommand( mciDevice, MCI_SAVE,
                            MCI_SAVE_FILE | MCI_WAIT,
                            (DWORD)(LPVOID)&mciSaveParms );

    if( dwRet != 0 )
    {
        ReportMCIError( hWnd, dwRet );
        return( FALSE );
    }
    return( TRUE ); // success
}

```

Handling Errors

The last function in the MCI module handles errors in any of the other functions. It puts up a message box telling the user what happened.

The error procedure needs two strings: it loads the first, the program title for the caption bar, from the string

table; it gets the second, an error message, directly from MCI. The `mciGetErrorString` function retrieves a string describing a WinMM error. `mciSendCommand` returns detailed error codes that *ShowWave* dutifully stores in its `dwResult` variable. If the return value is not 0, an error has occurred and *ShowWave* calls `ReportMCIErr`. Given the `dwResult` error code, `mciGetErrorString` returns an appropriate string.

The `Mmsystem.H` file defines about 90 different error codes. Some of them, like `MCIERR_INVALID_DEVICE_ID`, can happen any time; others, like `MCIERR_CANNOT_LOAD_DRIVER`, arise only during a specific command (in this case, `Open`); still others are peculiar to one device. `MCIERR_WAVES_OUTPUTSINUSE`, for example, indicates that all waveform devices are already busy.

```
//-----  
//  REPORT MCI ERROR  
//  Report given MCI error to the user  
//-----  
static void ReportMCIErr( HWND hwnd, DWORD dwError )  
{  
    HINSTANCE hInstance;  
    char szErrStr[MAXERRORLENGTH];  
    char szCaption[MAX_RSRC_STRING_LEN];  
  
    hInstance = GetWindowInstance( hwnd );  
    LoadString( hInstance, IDS_CAPTION, szCaption, sizeof(szCaption) );  
    mciGetErrorString( dwError, szErrStr, sizeof(szErrStr) );  
    MessageBox( hwnd, szErrStr, szCaption, MB_ICONEXCLAMATION | MB_OK );  
    return;  
}
```

The MMIO Module

If *ShowWave* only played and recorded sounds, it wouldn't need the MMIO module. Several of its functions, however, require the program to manipulate the data in sound files directly. Most obviously, to draw the sound wave, it must read samples from the .WAV file. Also, since the user can modify sounds in memory by mixing them or changing the volume, sometimes *ShowWave* must save data into a new file. The MMIO module contains one function to read wave data, one to write wave data, and one to handle file errors.

Reading the .WAV File

The `ReadWaveData` procedure loads all the data from a .WAV file into memory. It performs the following steps:

1. Opens the file.
2. Finds the WAVE chunk.
3. Locates the `fmt` subchunk and confirms that the sound is in a suitable format.
4. Finds the data subchunk and loads it into memory.
5. Closes the file.

```
//-----  
//  READ WAVE DATA  
//  Read waveform data from a RIFF file into a memory buffer.  
//  
//  RETURN  
//  TRUE if we successfully fill the buffer, otherwise FALSE.  
//  If the function returns TRUE, then the last three parameters  
//  return information about the new buffer.  
//-----  
BOOL ReadWaveData( HWND hwnd,  
                   LPSTR lpszFileName,  
                   LPSTR *lplpWaveData,           // points to buffer  
                   DWORD *lpdwWaveDataSize,        // size of buffer  
                   DWORD *lpdwSamplesPerSec )      // sampling rate  
{  
    HMMIO          hmmio;                          // file handle
```

```

MMCKINFO      mmckinfoWave;      // description of "WAVE" chunk
MMCKINFO      mmckinfoFmt;       // description of "fmt " chunk
MMCKINFO      mmckinfoData;      // description of "data" chunk
PCMWAVEFORMAT pcmWaveFormat;     // contents of "fmt " chunk
LONG          lFmtSize;          // size of "fmt " chunk
LONG          lDataSize;         // size of "data" chunk
LPSTR         lpData;            // pointer to data buffer
    // open the given file for reading using multimedia file I/O
hmmio = mmioOpen( lpszFileName, NULL, MMIO_ALLOCBUF | MMIO_READ );
if (hmmio == NULL)
{
    ReportError( hWnd, IDS_CANTOPENFILE );
    return( FALSE );
}

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The mmioOpen command takes three parameters: a filename, a structure for extra parameters, and some operation flags. The extra parameters matter only for changing the size of the file I/O buffer, for opening a memory file, or for naming a custom I/O procedure to read the file. Since ReadWaveData does none of these, the parameter is NULL.

MMIO_ALLOCBUF turns on I/O buffering. The other flag, MMIO_READ, opens the file for reading only. mmioWrite will return an error for files opened with MMIO_READ.

```
// locate a chunk with a "WAVE" form type
mmckinfoWave.fccType = mmioFOURCC('W','A','V','E');
if (mmioDescend( hmmio, &mmckinfoWave, NULL, MMIO_FINDRIFF ) != 0)
{
    ReportError( hWnd, IDS_NOTWAVEFILE );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
// find the format subchunk
mmckinfoFmt.ckid = mmioFOURCC('f','m','t',' ');
if( mmioDescend( hmmio, &mmckinfoFmt, &mmckinfoWave,
                MMIO_FINDCHUNK ) != 0)
{
    ReportError( hWnd, IDS_CORRUPTEDFILE );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
```

After opening the file, we next locate and verify the data. The first mmioDescend command looks for a "RIFF" tag followed by a WAVE code. If that works, the second command looks for the waveform's format subchunk.

To find the first chunk, we fill out only one field in the chunk info structure: fccType. The form type we seek is WAVE. The ckid (chunk ID) field should be RIFF, but the MMIO_FINDRIFF flag adequately describes that part of our target. The Descend command also recognizes three other flags: MMIO_FINDCHUNK,

MMIO_FINDRIFF, and MMIO_FINDLIST. In effect, the FINDCHUNK flag says to search for whatever is in the ckid field, and the other flags say to match the fccType field with a RIFF or LIST chunk.

mmioDescend takes four parameters: an HMMIO file handle, a description of the target chunk, a description of its parent chunk, and some operation flags. RIFF chunks don't have parents, so we leave the third field NULL, but the format chunk is always a subchunk of some parent. Only RIFF and LIST chunks can have subchunks.

NOTE:

Please pardon the mixed metaphors for chunk relationships. The terminology comes from the Microsoft manuals. Perhaps *superchunk* would be clearer than *parent*.

To find the format subchunk, we put "fmt" in the target information structure and "WAVE" in the parent information structure. mmioDescend will stop looking for "fmt" if it reaches the end of the current WAVE chunk. In this case, the file is unusable, perhaps corrupted, because you cannot interpret a WAVE without its format specifications.

The second Descend command left the file pointer at the beginning of the data in the format subchunk. Next, we load the format information into memory for verification:

```
// read the format subchunk
lFmtSize = (LONG)sizeof( pcmWaveFormat );
if( mmioRead( hmmio, (LPSTR)&amppcmWaveFormat, lFmtSize )
    != lFmtSize )
{
    ReportError( hWnd,IDS_CANTREADFORMAT );
    mmioClose( hmmio, 0 );
    return( FALSE );
}

// ascend out of the format subchunk
if( mmioAscend( hmmio, &ammckinfoFmt, 0 ) != 0 )
{
    ReportError( hWnd, IDS_CANTREADFORMAT );
    mmioClose( hmmio, 0 );
    return( FALSE );
}

// make sure the sound file is an 8-bit mono PCM WAVE file */
if( ( pcmWaveFormat.wf.wFormatTag != WAVE_FORMAT_PCM ) ||
    ( pcmWaveFormat.wf.nChannels != 1 ) ||
    ( pcmWaveFormat.wBitsPerSample != 8 ) )
{
    ReportError( hWnd, IDS_UNSUPPORTEDFORMAT );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
```

mmioRead expects a file handle, a pointer to a memory buffer, and a byte quantity. lFmtSize contains the number of bytes in a PCMWAVEFORMAT structure, and mmioRead loads that many bytes from the disk.

The Ascend command advances the file-position pointer past the last byte of the format chunk, ready for the next file operation. mmioAscend takes only three parameters because it never needs to think about the enclosing superchunk in order to find the end of a subchunk.

For clarity, we've limited ShowWave to one-channel sounds with 8 bits per pixel. To allow other ratings, you could add a few variables and modify the scrollbar code (see "Scrolling While Playing or Recording" later in this chapter).

We've verified the data format. Now we can load it into memory. We'll find the data subchunk, determine its size, allocate a memory buffer for it, and read the data into the buffer.

```
// find the data subchunk
mmckinfoData.ckid = mmioFOURCC('d','a','t','a');
if( mmioDescend( hmmio, &ammckinfoData, &ammckinfoWave,
```

```

        MMIO_FINDCHUNK ) != 0 )
    {
        ReportError( hWnd, IDS_CORRUPTEDFILE );
        mmioClose( hmmio, 0 );
        return( FALSE );
    }
    // get the size of the data subchunk
    lDataSize = (LONG)mmckinfoData.cksize;
    if( lDataSize == 0 )
    {
        ReportError( hWnd, IDS_NOWAVEDATA );
        mmioClose( hmmio, 0 );
        return( FALSE );
    }
    // allocate and lock memory for the waveform data
    lpData = GlobalAllocPtr( GMEM_MOVEABLE, lDataSize );
    if( ! lpData )
    {
        ReportError( hWnd, IDS_OUTOFMEMORY );
        mmioClose( hmmio, 0 );
        return( FALSE );
    }
    // read the data subchunk
    if( mmioRead( hmmio, (LPSTR)lpData, lDataSize ) != lDataSize )
    {
        ReportError( hWnd, IDS_CANTREADDATA );
        GlobalFreePtr( lpData );
        mmioClose( hmmio, 0 );
        return( FALSE );
    }
}

```

Finding the data chunk is just like finding the fmt chunk. mmioDescend fills the mmckinfoData variable with information that includes the size of the data. lDataSize tells us how much space to allocate from memory and how many bytes to read from the file.

To finish ReadWaveData, we close the file and return through the procedure's parameters three values: a pointer to the new memory object, the number of data bytes in the object, and the sampling rate:

```

    // close the file
    mmioClose( hmmio, 0 );
    // set return variables
    *lpWaveData = lpData;
    *lpdwWaveDataSize = (DWORD)lDataSize;
    *lpdwSamplesPerSec = pcmWaveFormat.wf.nSamplesPerSec;
    return( TRUE );
}

```

WARNING:

After closing the .wav audio file, do not free the pointer to the retrieved data, because the data is retained in memory for further use. When the audio waveform is replayed, instead of reopening the file to read the data again, the audio player plays from memory.

[Previous](#)
[Table of Contents](#)
[Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Writing the .WAV File

WriteWaveData transfers a wave sound from a memory buffer to a disk file. When the user modifies an existing sound or records a new one, WriteWaveData saves the result. It performs these steps:

1. Opens the file.
2. Creates the RIFF superchunk with a WAVE format type.
3. Creates the fmt subchunk; fills in its size and data fields.
4. Creates the data subchunk; fills in its size and data fields.
5. Ascends to the end of the file, causing the total size to be written in for the superchunk.
6. Closes the file.

```
//-----
//  WRITE WAVE DATA
//  Transfer waveform data from a memory buffer to a disk file
//-----
BOOL WriteWaveData( HWND hWnd,           // main window
                    LPSTR lpszFileName,   // destination file
                    LPSTR lpWaveData,     // data source buffer
                    DWORD dwWaveDataSize, // size of data in buffer
                    DWORD dwSamplesPerSec ) // sampling rate
{
    HMMIO      hmmio;           // file handle
    MMCKINFO   mmckinfoWave;    // description of "WAVE" chunk
    MMCKINFO   mmckinfoFmt;     // description of "fmt " chunk
    MMCKINFO   mmckinfoData;    // description of "data" chunk
    PCMWAVEFORMAT pcmWaveFormat; // contents of "fmt " chunk
    LONG       lFmtSize;        // size of "fmt " chunk
    LONG       lDataSize;       // size of "data" chunk

    // open the given file for writing using multimedia file I/O
    hmmio = mmioOpen( lpszFileName, NULL,
```

```

                                MMIO_ALLOCBUF | MMIO_WRITE | MMIO_CREATE );
if( hmmio == NULL )
{
    ReportError( hWnd, IDS_CANTOPENFILE );
    return( FALSE );
}
    // create a "RIFF" chunk with a "WAVE" form type
mmckinfoWave.fccType = mmioFOURCC( 'W','A','V','E' );
if( mmioCreateChunk( hmmio, &mmckinfoWave, MMIO_CREATERIFF )
    != 0 )
{
    ReportError( hWnd, IDS_CANTWRITEWAVE );
    mmioClose( hmmio, 0 );
    return( FALSE );
}

```

The mmioOpen command tells the system we want to buffer our file operations, write without reading, and create the file if it doesn't already exist. mmioCreateChunk expects three parameters: a file handle, a structure describing the new chunk, and an optional flag for creating superchunks.

The MMCKINFO structure has a field called dwDataOffset. mmioCreateChunk returns a value there telling where in the file the new chunk's data area begins. It also leaves the file pointer on the first byte of the new data area.

mmioCreateChunk cannot insert new chunks into the middle of a file. If the file pointer is not at the end of the file, old data will be overwritten.

Having established the main RIFF chunk, we next create and initialize the format subchunk:

```

    // store size of the format subchunk
lFmtSize = (LONG)sizeof( pcmWaveFormat );
    // Create the format subchunk.
    // Since we know the size of this chunk, specify it in the
    // MMCKINFO structure so MMIO doesn't have to seek back and
    // set the chunk size after ascending from the chunk.
mmckinfoFmt.ckid = mmioFOURCC( 'f', 'm', 't', ' ' );
mmckinfoFmt.cksize = lFmtSize;
if (mmioCreateChunk( hmmio, &mmckinfoFmt, 0 ) != 0)
{
    ReportError( hWnd, IDS_CANTWRITEFORMAT );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
    // initialize PCMWaveFormat structure
pcmWaveFormat.wf.wFormatTag      = WAVE_FORMAT_PCM;
pcmWaveFormat.wf.nChannels       = 1;
pcmWaveFormat.wf.nSamplesPerSec  = dwSamplesPerSec;
pcmWaveFormat.wf.nAvgBytesPerSec = dwSamplesPerSec;
pcmWaveFormat.wf.nBlockAlign     = 1;
pcmWaveFormat.wBitsPerSample     = 8;
    // write the format subchunk
if( mmioWrite( hmmio, (LPSTR)&pcmWaveFormat, lFmtSize )
    != lFmtSize )
{
    ReportError( hWnd, IDS_CANTWRITEFORMAT );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
    // ascend out of the format subchunk
if( mmioAscend( hmmio, &mmckinfoFmt, 0 ) != 0 )
{
    ReportError( hWnd, IDS_CANTWRITEFORMAT );
}

```

```

        mmioClose( hmmio, 0 );
        return( FALSE );
    }

```

Remember that every chunk contains a tag, a size, and some data. `mmioCreateChunk` leaves a space for the size, but if the `cksize` field is zero, then the space remains blank until the next `mmioAscend` seals off the new chunk. Normally, `mmioAscend` must calculate the data size, move back to the size field and fill it in, and then move forward to the end of the data. By providing the size, we avoid the extra backward motion, saving the time of two disk accesses.

The value in the global variable `dwSamplesPerSec` defaults to 22,050 (22.05kHz), but it changes whenever `ReadWaveData` loads a new file. (Choosing New from the File menu resets the value.) Because *ShowWave* restricts itself to one-channel sounds with 8 bits per sample, every sample always contains 1 byte. This is why we can put the same value in the `nSamplesPerSec` and `nAvgBytesPerSec` fields of the `pcmWaveFormat` variable.

The `nBlockAlign` field tells how many bytes one sample fills. The size of a sample must be rounded up to the nearest byte. A 12-bit-per-sample sound, for example, would align on 2-byte boundaries. Four bits would be wasted in each block, but when they are loaded into memory, the extra padding speeds up data access. The CPU always fetches information from memory in whole bytes and words, not bits.

You may wonder why we call `mmioAscend` when the write operation has already moved the file position to the end of the format data. Again, the answer has to do with alignment and access speed. A chunk's data area must always contain an even number of bytes so that it aligns on word (2-byte) boundaries. If the data contains an odd number of bytes, the final `mmioAscend` adds padding. Otherwise, it has no effect. `mmioCreateChunk` should nearly always be followed eventually by `mmioAscend`.

The format chunk written, we next perform the same steps to create the data chunk:

```

        // store size of the "data" subchunk
        lDataSize = (LONG)dwWaveDataSize;
        // create the "data" subchunk that holds the waveform samples
        mmckinfoData.ckid = mmioFOURCC( 'd', 'a', 't', 'a' );
        mmckinfoFmt.cksize = lDataSize;
        if( mmioCreateChunk( hmmio, &mpmmckinfoData, 0 ) != 0 )
        {
            ReportError( hWnd, IDS_CANTWRITEDATA );
            mmioClose( hmmio, 0 );
            return( FALSE );
        }
        // write the "data" subchunk
        if( mmioWrite( hmmio, lpWaveData, lDataSize ) != lDataSize )
        {
            ReportError( hWnd, IDS_CANTWRITEDATA );
            mmioClose( hmmio, 0 );
            return( FALSE );
        }
        // ascend out of the "data" subchunk
        if( mmioAscend( hmmio, &mpmmckinfoData, 0 ) != 0 )
        {
            ReportError( hWnd, IDS_CANTWRITEDATA );
            mmioClose( hmmio, 0 );
            return( FALSE );
        }
    }

```

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

That mmioAscend command moves to the end of the data subchunk, but remember we are still inside the RIFF superchunk. We've called mmioCreateChunk three times but mmioAscend only twice. One more to go:

```
// ascend out of the "WAVE" chunk--causes size to be written
if( mmioAscend( hmmio, &ammckinfoWave, 0 ) != 0 )
{
    ReportError( hWnd, IDS_CANTWRITEWAVE );
    mmioClose( hmmio, 0 );
    return( FALSE );
}
// close the file
mmioClose( hmmio, 0 );
return( TRUE );
}
```

Before creating each subchunk, we put a size value in the cksize field, so that WinMM knew the chunk size from the beginning. But for the first chunk, the parent chunk, we provided only a format type (WAVE). The final mmioAscend completes the creation of the first chunk. It computes the size and records it right after the "RIFF" tag at the beginning of the file.

Handling Errors

mciGetErrorString works only with the error returns from mciSendCommand; the file I/O procedures have no equivalent function for error messages. We put our own messages in ShowWave.RC and wrote ReportError to display them:

```
//-----
// REPORT ERROR
// Report given error to the user
//-----
static void ReportError( HWND hWnd, int iErrorID )
{
```

```

HINSTANCE hInstance;
char szErrStr[MAX_RSRC_STRING_LEN];
char szCaption[MAX_RSRC_STRING_LEN];

hInstance = GetWindowInstance( hWnd );
LoadString( hInstance, iErrorID, szErrStr, sizeof(szErrStr) );
LoadString( hInstance, IDS_CAPTION, szCaption,
            sizeof(szCaption) );
MessageBox( hWnd, szErrStr, szCaption,
            MB_ICONEXCLAMATION | MB_OK );
return;
}

```

The WinMain Module

WinMain registers window classes for the main window and the sound graph control. The custom window classes let us paint the control window and assign an icon to the dialog window. This module also contains the About box procedure.

The ShowWave Module

The MCI and MMIO modules provide a modular set of tools that any program might use to manipulate .WAV files. The ShowWave.C module runs the program's main window and a modal dialog box. In response to commands from the user, it calls functions from the other modules to play and record sounds.

ShowWave_WndProc mixes characteristics of a window procedure and a dialog procedure. Like a window procedure, it calls DefWindowProc and returns an LRESULT; like a dialog procedure, it receives WM_INITDIALOG rather than WM_CREATE. Because the window is launched by the DialogBox command, it initializes like a dialog box. Because the window has its own custom window class, it must have its own window procedure and does not use the default DefDlgProc processing.

ShowWave stores newly recorded sounds in the temporary file until they have names, so the dialog initialization handler generates a name for the file by calling GetTempPath and GetTempFileName. These commands generate a full path with a unique filename suitable for storing temporary data. If the environment defines a TEMP variable, the path takes it into account. The second and third parameters of GetTempFileName are for alphabetic and numeric elements to be combined in the filename.

SWT stands for ShowWave Temporary. Since we haven't provided a number, Windows will append digits drawn from the current system time to create a unique name. The new name is returned in the final parameter.

Responding to Commands

Some of the WM_COMMAND messages come from the menu, some from buttons on the dialog box. A different procedure handles each command. StopPlay appeared earlier in the MCI module.

```

//-----
//  SHOWWAVE_ONCOMMAND
//      Handle WM_COMMAND messages here.  Respond to actions from
//      each dialog control and from the menu.
//-----
static void ShowWave_OnCommand( HWND hDlg, int iCmd,
                               HWND hCtl, UINT uCode )
{
    switch( iCmd )
    {
        case IDM_NEW:                // clear data buffer
            NewWaveFile( hDlg );
            break;

        case IDM_OPEN:               // load a disk file

```

```

        OpenWaveFile( hDlg );
        break;

case IDM_SAVE:                                // save to disk file
case IDM_SAVEAS:
    SaveWaveFile( hDlg, (iCmd==IDM_SAVEAS) );
    break;

case IDM_EXIT:                                // end program
    FORWARD_WM_CLOSE( hDlg, PostMessage );
    break;

case IDM_MIXWITHFILE:                          // mix two sounds
    MixWithFile( hDlg );
    break;

case IDM_INCREASEVOLUME:                       // make louder
case IDM_DECREASEVOLUME:                       // make softer
    ChangeVolume( hDlg, iCmd==IDM_INCREASEVOLUME );
    break;

case IDM_ABOUT:                                // show About box
    DialogBox( GetWindowInstance(hDlg),
               MAKEINTRESOURCE(DLG_ABOUT),
               hDlg, About_DlgProc );
    break;

case IDD_PLAY:                                // play a sound
    PlayWaveFile( hDlg );
    break;

case IDD_RECORD:                              // record a sound
    RecordWaveFile( hDlg );
    break;

case IDD_STOP:                                // interrupt device
    if( mciDevice )
        StopPlay( hDlg, mciDevice );
    break;

default:
    break;
}
return;
UNREFERENCED_PARAMETER( hCtl );
UNREFERENCED_PARAMETER( uCode );
}

```

Scrolling the Wave Image

Windows translates scrollbar input into one of eight SB_ notification codes delivered through the first parameter of a WM_HSCROLL message. The eight signals reflect the actions described in Figure 14.3.

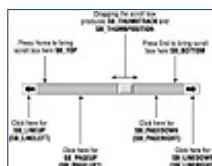


FIGURE 14.3 Scrollbar commands

Each program decides for itself what the signals mean. The SB_ signals are named for their most common

application, paging through a document. SB_TOP and SB_BOTTOM indicate opposite ends of the data. SB_LINEUP and SB_LINEDOWN move through the data by the smallest permissible increment, usually 1. The page-scrolling messages move at whatever intermediate increment the programmer decides is convenient.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

The *ShowWave* scrollbar's range measures the current sound in hundredths of a second. Line commands scroll in tenths of a second; page commands scroll in full seconds. When the new position is reached, we store it in *dwCurrentSample*, move the scrollbar thumb, update the dialog text that says how many seconds we have progressed into the file, and call another procedure to repaint the wave graph. The value in *dwCurrentSample* measures the current position in samples. The assignment statement converts hundredths of a second to samples. It assumes each sample contains 8 bits.

```
//-----  
//  SHOWWAVE_ONHSCROLL  
//      Process WM_HSCROLL messages here.  Advance through the sound  
//      file according to the user's signals from the scrollbar.  
//-----  
static void ShowWave_OnHScroll( HWND hDlg,  HWND hCtl,  
                               UINT uCode, int  iPos )  
{  
    int iMinPos, iMaxPos;  
    ScrollBar_GetRange( hCtl, &iMinPos, &iMaxPos );  
    switch( uCode )  
    {  
        case SB_LINEUP:          iHScrollPos -= 11;          break;  
        case SB_LINEDOWN:        iHScrollPos += 11;          break;  
        case SB_PAGEUP:          iHScrollPos -= 101;         break;  
        case SB_PAGEDOWN:        iHScrollPos += 101;         break;  
        case SB_TOP:              iHScrollPos = iMinPos;      break;  
        case SB_BOTTOM:          iHScrollPos = iMaxPos;       break;  
        case SB_THUMBPOSITION:    iHScrollPos = iPos;         break;  
        case SB_THUMBTRACK:       iHScrollPos = iPos;         break;  
        default:                  return;  
    }  
    // update scrollbar thumb  
    iHScrollPos = max( iMinPos, min(iHScrollPos, iMaxPos) );  
    ScrollBar_SetPos( hCtl, iHScrollPos, TRUE );  
}
```

```

        // set current sample
        dwCurrentSample = (iHScrollPos*dwSamplesPerSec) / 100;
        // set position and length text
        UpdateTimeText( hDlg );
        // paint the waveform data
        PaintWaveData( hDlg );
        return;
}

```

Scrolling While Playing or Recording~ShowWave plays or records a sound, we want the scrollbar thumb to move forward and the sound wave to scroll with it. As you'll see in a minute, *ShowWave* always begins a timer at the same time it begins playing or recording. ShowWave_OnTimer responds to the WM_TIMER messages.

```

//-----
//  SHOWWAVE_ONTIMER
//      Handle WM_TIMER messages here.  Update display to show
//      current position in sound file.
//-----
void ShowWave_OnTimer( HWND hDlg, UINT uID )
{
    int iMaxPos;
    HWND hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );
    if( uID == TMRPLAY ) // set the new scrollbar position
        FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_LINEDOWN,
                               0, SendMessage );
    else
    {
        // set the new waveform data
        dwWaveDataSize += dwSamplesPerSec/10;
        // set the new scrollbar range
        iMaxPos = (int)((dwWaveDataSize*100) / dwSamplesPerSec );
        ScrollBar_SetRange( hwndScrollBar, 0, iMaxPos, FALSE );
        // set the new scrollbar position
        FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_BOTTOM,
                               0, SendMessage );
    }
    return;
}

```

The timer messages arrive 10 times each second. The uID value is set when the timer begins and will be either TMRPLAY or TMRRECORD. If the timer is marking progress during a Play operation, we send ourselves a scroll message to advance the thumb one-tenth of a second further into the sound wave.

While *ShowWave* records, the size of the file through which we're scrolling changes. The scrollbar thumb is always at the right end of the bar because we're always at the end of the recorded data, but as more data comes in, the file size increases and the scrollbar range must increase, too. First we update the global variable dwWaveDataSize, adding to it the number of bytes received every tenth of a second. (Remember that with 8 bits per sample, each sample adds 1 byte to the file size.) Then we convert the new file size into hundredths of a second and set that as the scrollbar's new maximum range. And again we send ourselves another scroll message to keep the thumb at the end of the bar.

Ending a Play or Record Operation

Playing and recording continue in the background until the user clicks on Stop or the device reaches the end of its input element. When the action ends, WinMM sends the MM_MCINOTIFY message that we requested with the MCI_NOTIFY flag. The Windowsx.H file does not include message-handler macros for any multimedia messages, so we wrote HANDLE_MM_MCINOTIFY and put it in ShowWave.H. When the notify message comes, we need to close the device and kill the timer.

```

//-----
//  SHOWWAVE_ONMCINOTIFY
//      Handle MM_MCINOTIFY messages here.  Clean up after a

```

```
//      playback or recording operation terminates.
//-----
static void ShowWave_OnMCINotify( HWND hDlg, UINT uCode,
                                MCIDEVICEID mciDevice )
{
    int iMaxPos;
    HWND hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );
    if( uTimerID == TMRPLAY )
    {
        // close devices
        CloseAudioDevice( hDlg );
        CloseTimerDevice( hDlg );
        if( uCode == MCI_NOTIFY_SUCCESSFUL )
            FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_BOTTOM,
                                0, SendMessage );
    }
    else
    {
        // save recording and close devices
        SaveRecord( hDlg, mciDevice, szTempFileName );
        CloseAudioDevice( hDlg );
        CloseTimerDevice( hDlg );
        // set file dirty flag
        bIsFileDirty = TRUE;
        // read new waveform data
        ReadWaveData( hDlg, szTempFileName, &lpWaveData,
                      &dwWaveDataSize, &dwSamplesPerSec );
        // set the new scrollbar range
        iMaxPos = (int)( ( dwWaveDataSize * 100 ) / dwSamplesPerSec );
        ScrollBar_SetRange( hwndScrollBar, 0, iMaxPos, FALSE );
        // set the new scrollbar position
        FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_BOTTOM,
                            0, SendMessage );
    }
    return;
    UNREFERENCED_PARAMETER( mciDevice );
}
}
```

This time, we determine which action has stopped by testing a global variable, `uTimerID`, which we set when the timer started. If the device was playing sound and the sound ended successfully, we finish the scroll action by advancing the thumb to the end of the bar.

When *ShowWave* records a sound, the new data accumulates in a temporary file. When the recording ends, the program lifts the entire sound into a memory buffer so it can be the current sound. Since the user has not yet given the new sound a name and saved it, we mark the current file “dirty.” And again we finish the scroll operation by updating the scroll range and moving the thumb to the end.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

Ending the Program

The last few message-handling functions help the program close down without losing data or leaving objects behind.

```
//-----  
//  SHOWWAVE_ONQUERYENDSESSION  
//      Handle WM_QUERYENDSESSION messages here.  Give the user a  
//      chance to save any open file before program ends.  
//-----  
static BOOL ShowWave_OnQueryEndSession ( HWND hDlg )  
{  
    return( QuerySave(hDlg) );  
}  
//-----  
//  SHOWWAVE_ONDESTROY  
//      Handle WM_DESTROY message here.  Close all devices.  
//-----  
static void ShowWave_OnDestroy ( HWND hDlg )  
{  
    OFSTRUCT of;  
    DeleteBrush( hbrBackgnd );  
    FreeGlobalWaveData( ); // release buffer  
    CloseAudioDevice( hDlg ); // close devices  
    CloseTimerDevice( hDlg );  
    OpenFile( szTempFileName, &amp;of, OF_DELETE ); // delete temp file  
    PostQuitMessage( 0 ); // send WM_QUIT  
    return;  
}
```

To clean up, we delete the background brush created in ShowWave_OnInitDialog, release the buffer where the current sound is held, close the audio device if it's open, kill the timer if it's running, and delete the

temporary file we opened on initializing. Here are the cleanup functions:

```
//-----  
//  QUERY SAVE  
//      Ask user to confirm loss of unsaved file before closing  
//-----  
BOOL QuerySave ( HWND hDlg )  
{  
    HINSTANCE hInstance;  
    char szText[MAX_RSRC_STRING_LEN];  
    char szCaption[MAX_RSRC_STRING_LEN];  
    char szFormat[MAX_RSRC_STRING_LEN];  
    int iRet;  
  
    // is file dirty?  
    if( ! bIsFileDirty )  
        return( TRUE );  
    // see if user wants to save the modifications  
    hInstance = GetWindowInstance( hDlg );  
    LoadString( hInstance, IDS_CAPTION, szCaption,  
                sizeof(szCaption) );  
    LoadString( hInstance, IDS_CONFIRMCLOSE, szFormat,  
                sizeof(szFormat) );  
    wsprintf( szText, szFormat, (LPSTR)szFileTitle );  
    iRet = MessageBox( hDlg, szText, szCaption,  
                      MB_YESNOCANCEL | MB_ICONQUESTION );  
    if( iRet == IDYES )  
        FORWARD_WM_COMMAND( hDlg, IDM_SAVE, NULL, 0, SendMessage );  
    return( iRet != IDCANCEL );  
}  
//-----  
//  FREE GLOBAL WAVE DATA  
//      Free storage associated with the global waveform data  
//-----  
static void FreeGlobalWaveData( void )  
{  
    if( lpWaveData )  
    {  
        GlobalFreePtr( lpWaveData );  
        lpWaveData = NULL;  
    }  
    return;  
}  
//-----  
//  CLOSE AUDIO DEVICE  
//      Close an opened waveform audio device  
//-----  
static void CloseAudioDevice ( HWND hDlg )  
{  
    if( mciDevice )  
    {  
        CloseDevice( hDlg, mciDevice );  
        mciDevice = 0;  
    }  
    return;  
}  
//-----  
//  CLOSE TIMER DEVICE  
//      Kill an active timer; stop receiving WM_TIMER messages  
//-----
```

```

static void CloseTimerDevice ( HWND hDlg )
{
    if( uTimerID )
    {
        KillTimer( hDlg, uTimerID );
        uTimerID = 0;
    }
    return;
}

```

QuerySave tests bIsFileDirty to see whether the current sound has been saved. If not, we put up a message box asking the user what to do. The user chooses Yes (to save), No (to end without saving), or Cancel (to avoid ending after all). The function returns TRUE unless the user cancels.

Displaying Information in Static Controls

The next procedures change static controls in the dialog box to make them show current information.

```

//-----
//    PAINT WAVE DATA
//    Repaint the dialog box's GraphClass display control
//-----
static void PaintWaveData ( HWND hDlg )
{
    HWND hwndShowWave = GetDlgItem( hDlg, IDD_SHOWWAVE );
    InvalidateRect( hwndShowWave, NULL, TRUE );
    UpdateWindow( hwndShowWave );
    return;
}
//-----
//    UPDATE FILE TITLE TEXT
//    Set a new filename in dialog box's static filename control
//-----
static void UpdateFileTitleText ( HWND hDlg )
{
    Static_SetText( GetDlgItem(hDlg, IDD_FILETITLE), szFileTitle );
    return;
}
//-----
//    UPDATE TIME TEXT
//    Update the static dialog controls that show the scroll
//    thumb's current position and the playing time for the
//    current .WAV file
//-----
static void UpdateTimeText ( HWND hDlg )
{
    DWORD dwFrac;
    UINT  uSecs, uTenthSecs;
    char  szText[MAX_RSRC_STRING_LEN];
    char  szFormat[MAX_RSRC_STRING_LEN];
    // get the format string
    LoadString( GetWindowInstance(hDlg), IDS_TIMEFMT, szFormat,
        sizeof(szFormat) );
    // update position text
    dwFrac    = ((dwCurrentSample*100) / dwSamplesPerSec) / 10;
    uSecs     = (UINT)(dwFrac / 10);
    uTenthSecs = (UINT)(dwFrac % 10);
    wsprintf( szText, szFormat, uSecs, uTenthSecs );
    Static_SetText( GetDlgItem(hDlg, IDD_POSITION), szText );
    // update length text
    dwFrac    = ((dwWaveDataSize*100) / dwSamplesPerSec) / 10;

```



```

    uSecs      = (UINT)(dwFrac / 10);
    uTenthSecs = (UINT)(dwFrac % 10);
    wsprintf( szText, szFormat, uSecs, uTenthSecs );
    Static_SetText( GetDlgItem(hDlg, IDD_LENGTH), szText );
    return;
}

```

Resetting the Program

When the user chooses New from the File menu, the program must discard any current data and reset all its variables. NewWaveFile also updates the static dialog box controls and effectively disables the scrollbar by making its range very small.

```

//-----
//  NEW WAVE FILE
//      Start work on a new .WAV file.  Reset variables and update
//      display.  Called in response to the New command.
//-----
static void NewWaveFile ( HWND hDlg )
{
    HINSTANCE hInstance;
    HWND hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );

    // close the old .WAV file
    if( ! QuerySave( hDlg ) )
        return;
    // set filename and title
    hInstance = GetWindowInstance( hDlg );
    LoadString( hInstance, IDS_UNTITLED, szFileTitle,
        sizeof(szFileTitle) );
    szFileName[0] = '\\0';
    FreeGlobalWaveData();           // delete old waveform data
    bIsFileDirty = FALSE;           // set file dirty flag
    UpdateFileTitleText( hDlg );    // set filename text
    // set the new waveform data
    dwCurrentSample = 0;
    lpWaveData      = NULL;
    dwWaveDataSize  = 0;
    dwSamplesPerSec = 22050;
    // set the new scrollbar range
    ScrollBar_SetRange( hwndScrollBar, 0, 1, FALSE );
    // set the new scrollbar position
    FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_TOP, 0, SendMessage );
    return;
}

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.


```

ofn.lpstrFile           = lpstrFile;
ofn.nMaxFile           = iMaxFileNmLen;
ofn.lpstrFileName      = lpstrFileTitle;
ofn.nMaxFileName       = iMaxFileNameLen;
ofn.lpstrInitialDir    = NULL;
ofn.lpstrTitle          = NULL;
ofn.nFileOffset        = 0;
ofn.nFileExtension     = 0;
ofn.lpstrDefExt         = szOFNDefExt;
ofn.lCustData          = 0;
ofn.lpfHook            = NULL;
ofn.lpTemplateName     = NULL;
    // invoke the common dialog box
if( bOpenName )        // open a file
{
    ofn.Flags = OFN_HIDEREADONLY | OFN_PATHMUSTEXIST |
                OFN_FILEMUSTEXIST;
    return( GetOpenFileName(&ofn) );
}
else                    // Save As...
{
    ofn.Flags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT;
    return( GetSaveFileName(&ofn) );
}
}

```

Opening a Data File

When the user chooses Open from the File menu, the procedure asks the user to choose a file and then passes the name to ReadWaveData in the MMIO module. When the program loads new data, it resets the scrollbar range and pushes the thumb back to zero.

```

//-----
//  OPEN WAVE FILE
//      Open a new .WAV file
//-----
static void OpenWaveFile ( HWND hDlg )
{
    int iMaxPos;
    HWND hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );
    // close the old .WAV file
    if( ! QuerySave( hDlg ) )
        return;
    // get a .WAV file to open
    if( ! GetFileName( hDlg, TRUE, szFileName, sizeof(szFileName),
                      szFileTitle, sizeof(szFileTitle) ) )
        return;
    FreeGlobalWaveData();           // delete old waveform data
    bIsFileDirty = FALSE;           // set file dirty flag
    UpdateFileTitleText( hDlg );    // set filename text
    // read new waveform data
    dwCurrentSample = 0;
    ReadWaveData( hDlg, szFileName, &lpWaveData, &pdwWaveDataSize,
                  &pdwSamplesPerSec );
    // set the new scrollbar range
    iMaxPos = (int)(( dwWaveDataSize * 100 ) / dwSamplesPerSec );
    ScrollBar_SetRange( hwndScrollBar, 0, iMaxPos, FALSE );
    // set the new scrollbar position
    FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_TOP, 0, SendMessage );
    return;
}

```

Saving New Data

With the procedures we've already defined, writing a .WAV file to disk is easy. SaveWaveFile spends most of its time ensuring that we have a filename and that we don't write over important data.

```
//-----  
//  SAVE WAVE FILE  
//      Save waveform data to disk.  If the second parameter is TRUE,  
//      or if the current data does not yet have a filename, this  
//      procedure will request a name.  
//-----  
static void SaveWaveFile( HWND hDlg, BOOL bAskForName )  
{  
    HINSTANCE hInstance;  
    BOOL bSave;  
    char szText[MAX_RSRC_STRING_LEN];  
    char szCaption[MAX_RSRC_STRING_LEN];  
    int iRet;  
  
    // anything to save?  
    if( ! lpWaveData ) return;  
    // if renaming or no name, prompt user for name  
    if( ( bAskForName ) || ( szFileName[0] == '\0' ) )  
    { // get the name of the .WAV file to save as  
        bSave = GetFileName( hDlg, FALSE,  
            szFileName, sizeof(szFileName),  
            szFileTitle, sizeof(szFileTitle) );  
    }  
    else  
    { // no new name; confirm overwriting old file  
        hInstance = GetWindowInstance( hDlg );  
        LoadString( hInstance, IDS_OVERWRITE, szText, sizeof(szText) );  
        LoadString( hInstance, IDS_CAPTION, szCaption,  
            sizeof(szCaption) );  
        iRet = MessageBox( hDlg, szText, szCaption,  
            MB_YESNO | MB_ICONQUESTION );  
        bSave = (iRet == IDYES);  
    }  
    // save to the .WAV file  
    if( bSave )  
    {  
        bIsFileDirty = FALSE; // set file dirty flag  
        UpdateFileTitleText( hDlg ); // set filename text  
        //write out the waveform data  
        WriteWaveData( hDlg, szFileName, lpWaveData, dwWaveDataSize,  
            dwSamplesPerSec );  
    }  
    return;  
}
```

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

Mixing Two Sounds Together

ShowWave's Edit menu includes a Mix command. To mix one sound with another, you average them; that is, you add together each pair of samples and divide by two. When you play the combined sounds, you should hear both components simultaneously.

```
//-----  
// MIX WITH FILE  
// Mix a .WAV file into the current waveform data, combining  
// the data from both into a single recording. Mixing begins  
// at the current point in the current file. Mixing stops if  
// we reach the end of the current file; it will not extend  
// the current file.  
//-----  
static void MixWithFile ( HWND hDlg )  
{  
    HINSTANCE hInstance;  
    char szMixFile[_MAX_FNAME]; // name of second .WAV file  
    LPSTR lpMix; // data from second file  
    DWORD dwMixSize; // size of new data  
    DWORD dwMixSPS; // samples per second  
    char szErrStr[MAX_RSRC_STRING_LEN];  
    char szCaption[MAX_RSRC_STRING_LEN];  
    LPSTR lpDest, lpSrc; // pointers for data transfer  
    DWORD dw; // loop variable  
    int iMaxPos; // scrollbar range  
    HWND hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );  
  
    // get a .WAV file to mix with  
    if( ! GetFileName( hDlg, TRUE, szMixFile,  
                      sizeof(szMixFile), NULL, 0 ) )  
        return; // no filename
```

```

        // read waveform data
    if( ! ReadWaveData( hDlg, szMixFile, &lpMix, &pdwMixSize,
                        &pdwMixSPS ) ) // error reading data
        return;
    if( ! lpWaveData )                // mix data
    {
        bIsFileDirty = TRUE;          // set file dirty flag
        // set the new waveform data
        dwCurrentSample = 0;
        lpWaveData      = lpMix;
        dwWaveDataSize  = dwMixSize;
        dwSamplesPerSec = dwMixSPS;
        // set the new scrollbar range
        iMaxPos = (int)((dwWaveDataSize*100) / dwSamplesPerSec );
        ScrollBar_SetRange( hwndScrollBar, 0, iMaxPos, FALSE );
        // set the new scrollbar position
        FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_TOP,
                             0, SendMessage );
    }
    else
    {
        // for demo, use only matching frequencies
        if( dwSamplesPerSec != dwMixSPS )
        {
            hInstance = GetWindowInstance( hDlg );
            LoadString( hInstance, IDS_BADFREQUENCY, szErrStr,
                        sizeof(szErrStr) );
            LoadString( hInstance, IDS_CAPTION, szCaption,
                        sizeof(szCaption) );
            MessageBox( hDlg, szErrStr, szCaption,
                        MB_ICONEXCLAMATION | MB_OK );
            GlobalFreePtr( lpMix );
            return;
        }
        // mix new file into waveform at current position (lpDest)
        lpSrc = lpMix;
        lpDest = lpWaveData + dwCurrentSample;
        for( dw=0; dw<(dwWaveDataSize-dwCurrentSample); dw++ )
        {
            // merge one source and destination sample
            *lpDest = (BYTE)(((int)(BYTE)*lpDest + (BYTE)*lpSrc) / 2 );
            lpSrc++;
            lpDest++; // increment transfer pointers
            if( lpSrc >= ( lpMix + dwMixSize ) )
                break; // reached end of original data
        }
        // clean up
        GlobalFreePtr( lpMix ); // free memory
        bIsFileDirty = TRUE;    // set file dirty flag
        PaintWaveData( hDlg );  // paint the new waveform data
    }
    return;
}

```

MixWithFile begins by asking for the name of a .WAV file to open and reading the data into a second memory buffer, lpMix. If there is no current sound (if the user has not already opened another file), the new “mixed” sound simply becomes the current sound. We mark the new file as unsaved, set the global variables, and give the scrollbar new range values based on the length of the new sound.

But normally, the user will already have loaded a sound and we’ll need to combine two sets of data. For demonstration purposes, *ShowWave* mixes only sounds that have the same sampling rate. To mix sounds with different rates, you would skip over some samples in the faster sound. For example, if one had a

sampling rate of 11 and the other of 22, you would average every other sample from the second sound with one sample from the first.

The for loop that averages bytes together starts with the current position in the current sound. The user may already have played or scrolled partway through the file. The loop continues averaging bytes until it reaches the end of either sound, and then it stops. The new sound is cut off if it extends past the end of the old one. You could allow mixing to expand the current sound by calling `GlobalReallocPtr` to expand the `lpWaveData` buffer.

The line that averages two samples performs some typecasting to ensure that the compiler uses integers (which have 2 bytes) when it multiplies and divides. Even though the answer always fits in 1 byte, the intermediate product of two samples often overflows that limit.

Changing the Volume

The Effects menu lets the user make the current sound louder or softer. Like mixing, this effect involves modifying the samples mathematically. To understand the calculation, consider the diagram of a sound wave shown in Figure 14.4. The wave undulates above and below a middle point, called the *baseline*. When the wave swings very high and low, far away from the baseline, the sound it makes is loud. Quiet sounds have low peaks and shallow troughs. You can make a quiet sound loud by raising the peaks and lowering the troughs.



FIGURE 14.4 A sound wave undulates around its baseline.

When a digitizer samples a sound wave, it notes where the wave falls in relation to the baseline at any given moment. When recording with 8 bits per sample, the range of the largest wave is divided into 256 different regions. Each sample names one of the regions. Sixteen-bit samples perceive 65,536 regions in the same amplitude and so record much finer distinctions. *ShowWave* works with 8-bit samples, so a value of 128 represents the baseline.

To change the volume of a sound sample, first determine its distance from the baseline:

$$\text{WaveHeight} = \text{sample} - \text{baseline}$$

Samples with values less than 128 produce negative heights, indicating a trough in the wave. To increase the volume by 25 percent, multiply each height by 1.25 (125/100):

$$\text{LouderHeight} = (\text{WaveHeight} * 125) / 100;$$

Finally, we need to be sure the new value doesn't fall outside the possible range of 8-bit values. Amplitudes greater than 255 or less than 0 are not permitted.

$$\text{LouderSample} = \text{LouderHeight} + \text{baseline}$$

$$\text{LouderSample} = \max(0, \min(\text{LouderSample}, 255))$$

For 16-bit samples, make *LouderSample* type `LONG` and change the second calculation to this:

$$\text{LouderSample} = \max(-32768, \min(\text{LouderSample}, 32767))$$

ShowWave can either increase or decrease the volume by 25 percent. To decrease the volume, multiply each height by 0.75.

```
//-----  
//  CHANGE VOLUME  
//      Increase/decrease the volume of the waveform data playback  
//-----  
static void ChangeVolume( HWND hDlg, BOOL bIncrease )  
{  
    LPSTR lp;  
    DWORD dw;  
    int    iTmp, iFactor;  
  
    //anything to change?  
    if( ! lpWaveData ) return;  
    // change the volume of the waveform data
```

```
lp      = lpWaveData;
iFactor = (bIncrease ? 125 : 75);
for( dw=0; dw<dwWaveDataSize; dw++ )
{
    iTmp = (((int)(BYTE)(*lp) - MIDVALUE) * iFactor) / 100;
    *lp = (BYTE)max( MINVALUE, min(iTmp+MIDVALUE, MAXVALUE) );
    lp++;
}
// set file dirty flag
bIsFileDirty = TRUE;
// paint the new waveform data
PaintWaveData( hDlg );
return;
}
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#) [Table of Contents](#) [Next](#)

Playing a Sound

BeginPlay from the MCI module plays a sound, but PlayWaveFile performs some additional housekeeping chores before calling that core function.

```
//-----  
//  PLAY WAVE FILE  
//      Play waveform data  
//-----  
static void PlayWaveFile ( HWND hDlg )  
{  
    LPSTR lpsz;  
    DWORD dwFrom;  
    int    iMinPos, iMaxPos;  
    HWND  hwndScrollBar = GetDlgItem( hDlg, IDD_SCROLLBAR );  
  
    // anything to play?  
    if( ! lpWaveData )  
        return;  
    // get waveform file to play  
    if( ! bIsFileDirty )  
        lpsz = szFileName;  
    else  
    { // temporarily store waveform data to disk for playing  
        WriteWaveData( hDlg, szTempFileName, lpWaveData,  
                        dwWaveDataSize, dwSamplesPerSec );  
        lpsz = szTempFileName;  
    }  
  
    // if current position is end of sound, reset to beginning  
    ScrollBar_GetRange( hwndScrollBar, &iMinPos, &iMaxPos );  
    if( iHScrollPos == iMaxPos )  
        FORWARD_WM_HSCROLL( hDlg, hwndScrollBar, SB_TOP,
```

```

                                0, SendMessage );
    // convert current sample position to milliseconds
    dwFrom = (dwCurrentSample*1000) / dwSamplesPerSec;
    // play waveform file
    if( OpenDevice( hDlg, lpsz, &amcmciDevice ) )
    {
        if( ! BeginPlay( hDlg, mciDevice, dwFrom ) )
        {
            CloseAudioDevice( hDlg );
            return;
        }
        // set timer to update display
        uTimerID = TMRPLAY;
        SetTimer( hDlg, uTimerID, 100, NULL );
    }
    return;
}

```

Since `BeginPlay` must play from a disk file, first we need to decide which file to pass it. If the sound in memory has not changed since the user loaded it, *ShowWave* reads from the sound's original file. But if the user has changed the sound by mixing it or adjusting its volume, *ShowWave* deposits the sound in the temporary file we created on initialization.

If the user has scrolled partway through the current sound, we should start playing at the current position. The static variable `iHScrollPos` remembers where the scrollbar thumb is. If the user has scrolled to the end of the file, `PlayWaveFile` sends a message to reset the thumb at the beginning. `ShowWave_OnHScroll` answers the message, updating both `iHScrollPos` and `dwCurrentSample` to reflect the new file position.

Having guaranteed that the starting point is not the end of the file, `PlayWaveFile` opens the audio device and calls `BeginPlay`. The sound starts, and `BeginPlay` returns immediately. The `SetTimer` command causes Windows to send us `WM_TIMER` messages every tenth of a second while the sound continues to play. You already saw how `ShowWave_OnTimer` responds to each message by advancing the scrollbar thumb and scrolling the wave graph.

Recording a Sound

The code for recording closely resembles the code for playing sound. The opening chores differ, however. Although the high-level audio commands allow you to insert newly recorded sound into an existing wave sound, not all devices support the `MCI_RECORD_INSERT` flag with the `MCI_RECORD` command. For simplicity, *ShowWave* insists on recording to an empty file, and the opening `if` statement enforces the restriction.

We have also somewhat arbitrarily limited the recording to 20 seconds. The user can interrupt the recording any time before that by clicking the Stop button. If you prefer to leave the recording time open-ended, modify `BeginPlay` by removing the `MCI_TO` flag. (Or you might conditionally remove the flag only if the `dwTo` parameter is zero.)

```

//-----
//   RECORD WAVE FILE
//       Record waveform data
//-----
static void RecordWaveFile ( HWND hDlg )
{
    HINSTANCE hInstance;
    char szErrStr[MAX_RSRC_STRING_LEN];
    char szCaption[MAX_RSRC_STRING_LEN];

    // for demo purposes, record only onto new .WAV files
    if( lpWaveData )
    {
        hInstance = GetWindowInstance( hDlg );
        LoadString( hInstance, IDS_BADRECORDFILE, szErrStr,

```

```

        sizeof(szErrStr) );
    LoadString( hInstance, IDS_CAPTION, szCaption,
        sizeof(szCaption) );
    MessageBox( hDlg, szErrStr, szCaption,
        MB_ICONEXCLAMATION | MB_OK );
    return;
}
// record waveform data into a new file
if( OpenDevice( hDlg, "", &mciDevice ) )
{
    // set recording to stop after 20 seconds
    if( ! BeginRecord( hDlg, mciDevice, 20000 ) )
    {
        CloseAudioDevice( hDlg );
        return;
    }
    // set timer to update display
    uTimerID = TMRRECORD;
    SetTimer( hDlg, uTimerID, 100, NULL );
}
return;
}

```

The GraphWin Module

The *ShowWave* demo uses a custom window class, *GraphClass*, to display the sound wave. The custom control is defined in the resource script thus:

```
CONTROL    "", IDD_SHOWWAVE, "GraphClass", 0x0000, 53, 22, 76, 23
```

GraphClass names a window class that *ShowWave* registers when it initializes. The *Graphwin.C* module contains the three procedures that support our *GraphClass* window: *RegisterGraphClass* tells the system about the new class when the program begins, *Graph_WndProc* receives messages for the control, and *Graph_OnPaint* draws the sound wave.

```

//-----
//  REGISTER GRAPH CLASS
//      Register the window class for the dialog box's wave graph
//      control window.  The main dialog box's resource template
//      names this window class for one of its controls.  This
//      procedure must be called before CreateDialog.
//-----
BOOL RegisterGraphClass ( HINSTANCE hInstance )
{
    WNDCLASS wc;

    wc.style          = 0;
    wc.lpfnWndProc    = Graph_WndProc;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hIcon          = NULL;
    wc.hCursor        = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground  = GetStockBrush( BLACK_BRUSH );
    wc.lpszMenuName   = NULL;
    wc.lpszClassName  = szGraphClass;
    return( RegisterClass(&wc) );
}

//-----

```

```
//      GRAPH WNDPROC
//      Receive messages for the main dialog box's sound graph
//      window.
//-----
LRESULT WINAPI Graph_WndProc( HWND    hWnd,    UINT    uMsg,
                              WPARAM wParam, LPARAM lParam )
{
    switch (uMsg)
    {
        HANDLE_MSG( hWnd, WM_PAINT, Graph_OnPaint );
        // paint the green sound graph line
        default:
            return( DefWindowProc(hWnd, uMsg, wParam, lParam) );
    }
}
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

The window procedure intercepts only one message, WM_PAINT, and in every other case, the window accepts the standard message responses from the default window procedure. (Dialog-box controls are not themselves dialog boxes, so they call DefWindowProc and not DefDlgProc.)

Our GraphClass window isn't really a full-blown control. Real controls must be careful with global or static variables because, unlike child windows, they do not have separate data segments for each window instance. Changing one static variable for one control would change that variable for all controls of the same class. Also, a control should answer the WM_GETDLGCODE message to tell the parent dialog box what keyboard input it wants. (GraphClass doesn't want any, and would respond with DLGC_STATIC.)

TIP:

A tightly designed custom control can be compiled into a DLL and made available to all applications, including a dialog editor. To do this, you would need to write and export a small set of standard control functions.

Drawing the Sound Wave

Rather than connecting points on the wave curve, *ShowWave* represents each sample as a vertical line. The wave height indicated in the sample determines the height of the vertical line. Each line extends an equal distance above and below the baseline, coloring in the space over and under each wave. Solid shapes show up better in the small graph window than a single wave line could. In Figure 14.5, you can see how the vertical lines fill the wave.



FIGURE 14.5 How *ShowWave* draws the sound wave

```
//-----
//  GRAPH_ONPAINT
//      Handle WM_PAINT messages for the Graph window.  Draw
//      the green wave graph.
//-----
static void Graph_OnPaint( HWND hWnd )
```



```

{
    PAINTSTRUCT ps;
    HDC    hDC;
    HPEN    hPen, hpenOld;        // green pen for drawing columns
    RECT    rClient;              // size of graph control window
    int     iBase;                // vertical position of baseline
    LPSTR    lp;                  // points to one sample in wave data
    DWORD    dwMaxStart;          // maximum value for starting point
    int     iYScale;              // vertical scaling factor
    int     iCol;                 // horizontal position of a column
    int     iColHeight;           // height of a column

    // begin paint processing
    hDC = BeginPaint( hWnd, &amps );

```

An earlier version of this program called the `GetDeviceCaps` function to query the `NUMCOLORS` value to decide whether the device could support color or only black and white (B/W). At the time this original version was written, there were still some B/W video systems around, but `TrueColor` and `HighColor` video were virtually unknown. The original test was written as:

```

    // create a pen for drawing graph
    if( GetDeviceCaps( hDC, NUMCOLORS ) > 2 )

```

Today, however, `TrueColor` and `HighColor` video systems will report `NUMCOLORS` as 0 simply because they do not rely on color palettes. Instead, a more appropriate test is to query the `BITSPIXEL` value, thus:

```

    if( GetDeviceCaps( hDC, BITSPIXEL ) > 2 )
        hPen = CreatePen( PS_SOLID, 1, RGB_GREEN ); // color display
    else
        hPen = CreatePen( PS_SOLID, 1, RGB_WHITE ); // mono display
    if( hPen )
    {
        // select the pen
        hpenOld = SelectPen( hDC, hPen );
        // draw the waveform baseline
        GetClientRect( hWnd, &rClient );
        iBase = RECTHEIGHT(rClient) / 2;
        MoveToEx( hDC, 0, iBase, NULL );
        LineTo( hDC, (int)rClient.right, iBase );
        // graph waveform data
        if( lpWaveData )
        {
            // set the current sample position in the waveform data
            dwMaxStart = dwWaveDataSize - RECTWIDTH(rClient) - 1;
            lp = lpWaveData + min( dwCurrentSample, dwMaxStart );
            // determine the height scaling factor
            iYScale = ( ( MAXVALUE - MINVALUE ) + 1 ) / // amplitude
                      ( RECTHEIGHT(rClient) - 4 ); // control height
            // Subtracting 4 from the height ensures a small
            // margin above and below the biggest waves
            /* paint samples from the waveform data */
            for( iCol=(int)rClient.left; iCol<=(int)rClient.right;
                iCol++ )
            {
                iColHeight = ( (int)(BYTE)(*lp++) - MIDVALUE ) / iYScale;
                if( iColHeight != 0 )
                {
                    // figure absolute value of column height
                    if( iColHeight < 0 )
                        iColHeight = -iColHeight;
                    // draw line from below base to above base
                    MoveToEx( hDC, iCol, iBase - iColHeight, NULL );
                    LineTo( hDC, iCol, iBase + ( iColHeight + 1 ) );
                }
            }
        }
    }
}

```

```

    }
    // restore the DC and delete the pen */
    SelectPen( hDC, hpenOld );
    DeletePen( hPen );
}
// end paint processing
EndPaint( hWnd, &amps );
return;
}

```

Graph_OnPaint represents the baseline with a horizontal line bisecting the graph window. The first vertical line on the left side of the control window will represent the current sample. If the user has scrolled to a point near the end of the file and only a few samples remain, the wave graph might not extend all the way across the window. To avoid leaving part of the graph empty, the program imposes a maximum value for the starting point. If the graph control is 50 pixels wide, for example, the graph must not begin less than 50 samples from the end.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The heights must be scaled to fit inside the window. A line extending the full height of the window must represent the maximum possible amplitude. With 8-bit samples, the maximum is 256. *ShowWave* calculates the line height with the following formulas:

Scale factor = maximum amplitude / window height

Column height = sample height / scale factor

To preserve a small margin of 2 pixels below and above the tallest waves, *ShowWave* subtracts 4 from the window height when figuring the scale factor.

The loop that draws each column begins by figuring the sample height and scaling it down to fit in the window. It draws each column by moving to a point below the baseline and drawing upward to a point an equal distance above the baseline. Because the columns extend on both sides of the baseline, the height of each column is twice the height of the sample.

We could call the C library function `abs` to get the absolute value of `iColHeight`, but then the compiler would add library code to the .EXE file. The `if` statement takes up less memory.

Some Improvements for the *ShowWave* Program

You might want to do some experimenting to see how you can improve the demo multimedia program. Here we present some ideas, but you probably can think of some others.

Although the 8-bit sample size is common, you might want to allow *ShowWave* to work with 16-bit samples as well. We've described several places where *ShowWave* makes calculations that assume 8-bit samples, such as in changing the volume, drawing the sound wave, and moving the scrollbar thumb. You would need to change any place that assumes a sample is a `BYTE` or refers to the manifest constants `MINVALUE`, `MIDVALUE`, and `MAXVALUE`. Sixteen-bit samples range in value from -32,768 to 32,767, with a midpoint of zero.

You also know enough to add stereo sound. `MixWithFile` and `ChangeVolume` would need to do everything twice—once for the left channel and once for the right. To draw the sound wave, you could average both channels together or let the user choose which channel to see. In a stereo data chunk, the channels are interleaved:

Sample one: channel 0 sample, channel 1 sample

Sample two: channel 0 sample, channel 1 sample

You might also modify `MixWithFile` to permit expanding the original sound. To do this, you would need to reallocate the buffer periodically.

The `MCI_OPEN` command reloads the waveaudio driver each time you open the device. From working with printers, you know that loading the driver causes a noticeable lag. *ShowWave* could load the driver once on opening and close it once at the end. In between, it would open and close individual device elements. The very first Open command would specify a device type but no device element; subsequent commands would open and close with an element name. The final close would again omit the element.

Several programs may open one device simultaneously, so holding it open won't cause trouble. (Whether different programs can share the same device *element* depends on the flags they use with `MCI_OPEN`.)

With `EnableWindow` and `EnableMenuItem`, you could disable buttons and menu commands not currently available. For example, all the buttons should be disabled until the user loads a sound. Stop should be disabled unless a playback or record message is in progress.

Finally, you could add `MessageBeep` commands to the program's error messages. Those who favor restraint in interface design will limit audio signals to the more critical errors or allow the user to choose whether and when to hear error sounds.

Summary

This chapter began with a general discussion of the hardware for multimedia applications and the WinMM translation layer that permits Windows programs to manipulate multimedia files in a device-independent manner. Windows works with files that contain data for wave sounds, MIDI music, animation, and analog video, among other formats.

WinMM was designed to be extensible. When other data types appear, new chunk formats can expand the RIFF standards. The MCI commands could send new messages, and their parameter structures could easily acquire new fields to accommodate more input. The high-level MCI commands establish a general protocol for opening and operating many very different devices.

For the demo discussed in this chapter, we chose to work with waveform audio because its hardware requirements are less demanding; many machines now have inexpensive sound cards. The *ShowWave* demo demonstrates a full range of MCI command messages for opening and closing a device, playing and recording sounds, and interrupting operations. The MMIO module demonstrates the group of WinMM functions that facilitate reading and writing chunks of a RIFF file.

In the next chapter, we'll take a brief look at some of the newest multimedia (MMX) functions supported by the new Pentium III CPUs.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 15

Multimedia MMX for the Pentium III

- Intel's MMX and SIMD extensions
- Floating-point instructions for SIMD
- Integer instructions for SIMD
- Cacheability Control instructions for SIMD
- SIMD State Management operations
- Resources for the SIMD extensions

With the release of the Pentium III (in mid-1999), Intel offered new MMX technology. Originally—that is, during development—these instructions were known as the *Katmai New Instructions*, but these are more properly referred to as the *Internet Streaming SIMD Extensions* where SIMD is the acronym for *Single Instruction, Multiple Data*. For convenience, however, we'll simply refer to these as the *SIMD instructions*.

NOTE:

The SIMD acronym is pronounced "Sim-Dee."

Unlike most topics in this book that use C/C++ examples and are high-level programs, the SIMD instructions are more relevant to low-level assembly language operations. This is not because assembly language subroutines cannot be called from C/C++ programs, and not because there is any impediment to using inline assembly language routines. It is simply because the intent and purpose of the SIMD instructions is to execute processes at the highest possible speeds. Ergo, the SIMD instructions would normally be used in creating library procedures that might be called from programs written in a higher-level language.

However, the SIMD instructions offer a distinct departure from the more customary single-instruction, single-data operations that have previously been the mainstay of graphic-intensive operations and particularly of 3-D graphic operations. Therefore, this chapter will offer an overview of the new SIMD instructions together with notes on where to find further information and examples. We will not attempt to provide

working examples using the SIMD instruction set.

Intel and MMX

Intel's MMX services consist of a set of 57 general-purpose integer instructions and four data types, which were aimed specifically at serving the requirements of multimedia and communications applications. The SIMD extensions comprise 70 new instructions, along with eight 64-bit wide Intel MMX technology registers. These 70 new instructions include single-instruction, multiple-data operations for floating-point calculations and additional SIMD integer and cacheability control instructions. The SIMD extensions fall into four primary categories:

- SIMD single-precision floating-point instructions
- Additional SIMD integer instructions
- Cacheability control instructions
- State management instructions

Technologies and applications that can benefit from the SIMD extensions include:

- 3-D animation
- Advanced imaging
- Speech recognition
- Streaming audio and video

For graphic applications, the benefits mean that higher-resolution and higher-quality images can be viewed and manipulated, including high-quality audio, MPEG2 video, and simultaneous MPEG2 encoding and decoding.

For speech recognition applications, the SIMD extensions offer reduced CPU workloads plus higher accuracy and faster response times.

Support for SIMD

When Intel released their original MMX instruction set, the advanced graphics support was readily welcomed by developers but failed to fulfill early promises, primarily because of the lack of tools for developing MMX-capable applications.

The new SIMD extensions should be less vulnerable to such perceived shortcomings because Microsoft's DirectX technology (version 6.1 or later) already includes provisions to make use of the new Pentium III instructions. Thus, when the geometry and lighting of 3-D objects needs to be calculated, DirectX will route those tasks to the Pentium III by invoking the SIMD instructions. In like fashion, DirectX software also built in support for AMD's multimedia instruction set known as 3DNow!

You should keep in mind, however, that the SIMD extensions have little to offer to improve 2-D business applications. Instead, these new instructions are primarily aimed at the computation-intensive needs found in 3-D applications.

SIMD Floating-Point Instructions

SIMD floating-point instructions operate on a newly defined 32-bit floating-point data type, single precision floating-point values with a normalized range from 2^{-126} to 2^{127} . Significant improvements in performance are achieved by using single instructions to process as many as four 32-bit data elements in parallel. The four 32-bit values are packed into a single 128-bit value for operations. Figure 15.1 shows how the packed data type is contained in a 128-bit register.



FIGURE 15.1 Packing single precision floating-point values

SIMD single-precision floating-point instructions utilize direct access to eight new 128-bit general-purpose registers identified as XMM0 through XMM7. These registers are used for data calculations only, while memory addressing continues to use the integer registers and the existing Intel architecture addressing modes.

NOTE:

In earlier Intel MMX operations, floating-point numbers were represented as 80-bit extended precision values.

Floating-point instructions include *packed operations* and *scalar operations*. Packed operations are identified by the suffix *ps* as, for example, `addps`. In packed operations, all four single precision values are subjected to the same operation.

By contrast, scalar operations are identified by the suffix *ss* as, for example `addss`. In scalar operations, only the least significant data elements of the two operands are used and the upper three data elements in the destination register remain unaffected.

The SIMD Floating-Point Operations

The floating-point instructions fall into 12 categories:

- Basic arithmetic
- Square root calculations
- Fast approximation operations
- Min/max comparisons
- Shuffle
- Unpack
- Data movement
- Move mask from floating point(s) to integer(s)
- Compare and set mask
- Logical instructions
- Compare and set EFLAGS
- Conversion operations

Basic Arithmetic

`mulps source, destination`

SIMD arithmetic operations include addition, subtraction, multiplication, and division, as both packed and scalar operations. While the source operand for such instructions can be either an operand in memory or an XMM register (128-bits), the destination operand must always be an XMM register. Basic arithmetic operations include:

`addps addss`
`subps subss`
`mulps mulss`
`divps divss`

Square Root Calculations

`sqrtps source, destination`

Square root instructions calculate the square root value of the four floating-point values in the source operand and write the results to the destination. While the source may be either an operand in memory or an XMM register, the destination must be an XMM register. Both packed and scalar operations are supported:

`sqrtps sqrtss`

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Fast Approximation Operations

rcpps source, destination

Fast approximation operations provide either square root or reciprocal values for the four floating-point values in the source operand. These values are found using on-chip lookup tables and can be used when full precision results are not needed. The approximation operations provide 11 bits of precision and do not generate numeric exceptions. Supported approximation operations are:

rcpps rcpss
rsqrtps rsqrtss

Min/Max Comparisons

minps source, destination

The minimum and maximum comparison instructions perform comparisons between the four floating-point values in the source and destination operands, writing the results to the destination operand. Again, the first argument may be either an operand in memory or an XMM register, but the destination must be an XMM register. Supported comparisons are:

minps minss
maxps maxss

Shuffle

shufps source, destination, pattern

The shuffle instruction (shufps) selects two of the four floating-point data elements from each operand, copying the elements from the first operand to the lower two elements in the destination and copying the elements from the second operand to the higher two elements in the destination. Again, the first argument may be either an operand in memory or an XMM register, but the destination must be an XMM register. Alternately, both of the sources can be a single register, allowing the order of the four floating-point values to

be rearranged.

The pattern argument is a byte value that is treated as four 2-bit arguments, with each identifying a position (0..3) in the two registers. Using a single register, the shuffle operation supports three rearrangements titled *broadcast*, *rotate*, and *swap*.

A *broadcast* operation copies any one data element from the source to all four positions in the destination:

- A value of 00h (00 00 00 00) broadcasts the least significant data element.
- A pattern value of 55h (01 01 01 01) broadcasts the second data element.
- A pattern value of AAh (10 10 10 10) broadcasts the third data element.
- A pattern value of FFh (11 11 11 11) broadcasts the most significant data element.

A *rotate* operation can perform either a right or left rotation of the data elements. The pattern 93h (10 01 00 11) rotates the four elements to the left, moving the most significant data element to the least significant position. The pattern 39h (00 11 10 01) rotates the elements to the right, moving the least significant data element to the most significant position.

A *swap* operation uses the pattern 1Bh (00 01 10 11) to create a mirror image of the source operand by swapping the upper and lower values and the second and third values.

Unpack

`unpcklps source, destination`

The unpack instruction can be called as unpack high (`unpckhps`) or unpack low (`unpcklps`). The unpack high instruction copies and interleaves the two higher floating-point data elements from each of the two operands, placing the result in the destination register. In like fashion, unpack low copies and interleaves the two lower floating-point elements. While the source element may be either an operand in memory or an XMM register, the destination must be an XMM register. Also, even though 64 bits are ignored, data references in memory must be 128-bit aligned. Supported unpack operations are:

`unpckhps unpcklps`

Data Movement

`movaps [memaddr], xmm2`
`movaps xmm1, [memaddr]`

The data movement instructions are used to move either packed or scalar floating-point data elements between memory and XMM registers or between XMM registers. Data movement is supported to and from both aligned memory locations (`movaps`, `movhps`, `movlps`) and unaligned locations (`movups`). Copying to or from unaligned memory locations, however, requires more cycles to execute than copying to or from aligned operations. Seven data movement operations are supported:

<code>movaps</code>	Moves four floating-point elements between an XMM register and memory or between two XMM registers. The location in memory must be 128-bit aligned.
<code>movups</code>	Moves four floating-point elements between an XMM register and memory or between two XMM registers. The location in memory is not required to be 128-bit aligned.
<code>movhps</code>	Moves two floating-point elements between the two higher data elements in an XMM register and memory.
<code>movlps</code>	Moves two floating-point elements between the two lower data elements in an XMM register and memory.
<code>movss</code>	Moves 32 bits (one floating-point data element) between memory and the least significant element in an XMM register or between two XMM registers. When <code>movss</code> is used to copy data to an XMM register, the upper 96 bits are cleared.
<code>movhlps</code>	Moves the two higher floating-point data elements from the source to the two lower data elements in the destination. Both source and destination must be XMM registers.

movlhps

Moves the two lower floating-point data elements from the source to the two higher data elements in the destination. Both source and destination must be XMM registers.

For operations to or from memory, the linear (memory) address always corresponds to the address of the least-significant byte of the referenced memory data.

Move Mask from Floating Point(s) to Integer(s)

movmskps eax, XMMn

The move mask from floating point to integer instruction (movmskps) copies the sign bit (the most significant bit) from each of the four data elements in an XMM register to the four least significant bits in an integer register. The upper 28 bits of the integer register are cleared.

Compare and Set Mask

cmpltps source, XMMn

The compare and set mask instructions comprise a set of 16 Boolean comparisons:

cmpeqps	cmpeqss	Equal to
cmpltps	cmpltss	Less than
cmpleps	cmpless	Less than or equal to
cmpneps	cmpness	Not equal to
cmpnltps	cmpnltps	Not less than
cmpnleps	cmpnless	Not less than or equal to
cmpordps	cmpordss	Ordered comparison
cmpunordps	cmpunordss	Unordered comparison

The comparisons are made between the contents of the source and the contents of the destination. While the source element may be either an operand in memory or an XMM register, the destination must be an XMM register. If the results of the comparison are TRUE, a mask of all ones is written to the corresponding element in the destination register. If FALSE, a mask of all zeros is written.

NOTE:

When two operands are compared but either or both operands are non-numerical, an unordered comparison is performed on the entire 128-bit value.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Logical Instructions

`andps xmm1, xmm2`

The logical instructions are used to perform bitwise logical operations on two packed floating-point data elements. These are commonly used to:

- Return an absolute value
- Change the sign of the value (negation)
- Perform conditional comparison operations

The four logical operations are:

<code>andps</code>	Performs a logical AND operation between two 128-bit operands, setting destination bits to one if and only if both operand bits are ones.
<code>andnps</code>	Performs a logical NAND operation between two 128-bit operands by inverting the destination operand and then performing an AND operation.
<code>orps</code>	Performs a logical OR operation between two 128-bit operands, setting destination bits to zero if and only if both operand bits are zeros.
<code>xorps</code>	Performs an exclusive OR operation between two 128-bit operands, setting destination bits to one if the corresponding operand bits are not the same.

The source operand may be either an operand in memory or an XMM register, while the destination must be an XMM register.

Compare and Set EFLAGS

`comiss xmm1, xmm2`

The compare and set EFLAGS instructions are used to perform unordered comparisons between scalar floating-point data elements with the results placed in the ZF, PF, and CF status flags in the EFLAGS register. The OF, SF, and AF flags are all set to zeros. Comparison results can report greater-than, less-than, or equal conditions:

EFLAGS			
result	ZF	PF	CF
Unordered	1	1	1
Less-than	0	0	1
Greater-than	0	0	0
Equal	1	0	1

Conversion Operations

`cvtps2pi eax, xmm1`

The conversion operations instructions perform conversions between packed or scalar 32-bit signed integer data and packed or scalar floating-point data elements. Operations are performed between registers only.

The six conversion operations are:

<code>cvtps2pi</code>	Converts packed floating point to rounded integer
<code>cvtss2si</code>	Converts scalar floating point to rounded integer
<code>cvttps2pi</code>	Converts packed floating point to truncated integer
<code>cvttss2si</code>	Converts scalar floating point to truncated integer
<code>cvtpi2ps</code>	Converts packed integer to floating point
<code>cvt si2ss</code>	Converts scalar integer to floating point

SIMD Integer Instructions

The SIMD extensions are not limited to floating-point operations, but also include a series of integer instructions. Just as the SIMD floating-point operations speed floating-point calculations, these integer instructions also manipulate four simultaneous integer operations in parallel.

To support integer operations, the SIMD extension defines four integer types as *packed byte*, *packed word*, *packed doubleword*, and *quadword* integers. Each type is a 64-bit value—the size of the MMX integer registers—as shown in Figure 15.2.

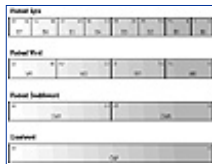


FIGURE 15.2 Four SIMD integer types

The SIMD extensions use eight MMX integer registers—identified as MM0 through MM7—that are aliased on the floating-point registers (XMM0-XMM7).

The SIMD integer instruction mnemonics use a prefix and suffixes to indicate the operation and data types. The prefix *p* indicates a packed operation on multiple integer data elements. These suffixes indicate the data types:

b	Byte
w	Word
d	Doubleword
q	Quadword

The SIMD Integer Instructions

The SIMD integer instructions include:

- Extract
- Insert
- Min/max
- Move byte mask to integer

- Multiply high unsigned
- Shuffle

Extract

```
pextrw eax, mm1, n
```

The integer extract instruction (`pextrw`) copies a single 16-bit word valued from a packed word data type—from an MMX register—to the lower word element in a 32-bit integer register, while the upper 16 bits of the destination register are cleared (set to zero). The third argument (`n`) uses the two least significant bits to determine which of the four packed word values to extract.

Insert

```
pinsrw mm1, eax, n
```

The integer insert instruction (`pinsrw`) copies the lower 16-bit word from a 32-bit integer register or copies from a 16-bit memory operand to an MMX register. The third argument (`n`) uses the two least significant bits to determine where the word is inserted in the MMX register.

Min/Max

```
pminsw mm1, mm2
```

The integer minimum and maximum comparison instructions compare packed data elements as signed word values or as unsigned byte values in the source and destination operands, writing the minimum or maximum values to the destination register. The four comparison operations are:

<code>pminsw</code>	Minimum comparison, four signed word values
<code>pmaxsw</code>	Maximum comparison, four signed word values
<code>pminub</code>	Minimum comparison, eight unsigned byte values
<code>pmaxub</code>	Maximum comparison, eight unsigned byte values

Move Byte Mask to Integer

```
pmovmskb eax, mm1
```

The move byte mask to integer instruction (`pmovmskb`) copies the eight most significant bits from the eight packed data byte values in an MMX register to form an 8-bit mask in a 32-bit integer register. The upper 24 bits in the register are cleared (set to zero).

Multiply High Unsigned

```
pmulhuw source, destination
```

The integer multiply high unsigned instruction (`pmulhuw`) multiplies four unsigned 16-bit word elements—either from a 64-bit operand in memory or from an MMX register—with the four values in the MMX destination register. The high 16 bits of the 32-bit intermediate products are stored in the destination register, while the low 16 bits of each are discarded.

Shuffle

```
pshufw source, destination, pattern
```

The integer shuffle instruction (`pshufw`) selects four word elements from the source operand, either from a 64-bit operand in memory or from an MMX register, writing these to the destination register in the order specified by the pattern argument. The pattern argument is treated as four 2-bit fields where each 2-bit argument (0..3) identifies one of the four word elements in the source.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Like the floating-point shuffle (shufps) instruction, the integer shuffle instruction supports three rearrangements titled *broadcast*, *rotate*, and *swap*.

A *broadcast* operation copies any one data element from the source to all four positions in the destination.

- A value of 00h (00 00 00 00) broadcasts the least significant data element.
- A pattern value of 55h (01 01 01 01) broadcasts the second data element.
- A pattern value of AAh (10 10 10 10) broadcasts the third data element.
- A pattern value of FFh (11 11 11 11) broadcasts the most significant data element.

A *rotate* operation can perform either a right or a left rotation of the data elements. The pattern 93h (10 01 00 11) rotates the four elements to the left, moving the most significant data element to the least significant position. The pattern 39h (00 11 10 01) rotates the elements to the right, moving the least significant data element to the most significant position.

A *swap* operation uses the pattern 1Bh (00 01 10 11) to create a mirror image of the source operand by swapping the upper and lower values and the second and third values.

SIMD Cacheability Control

The SIMD cacheability control instructions:

- Govern how data is cached to minimize cache pollution and reduce the number of write accesses to memory.
- Prefetch data to exploit concurrency between memory and execution pipelines and to hide memory latency effects.

The cacheability control instructions include:

- Streaming store
- Move masked bytes to memory
- Prefetch
- Store fence

Streaming Store

The streaming store instructions are used to write *nontemporal* data directly to memory without using the cache. Bypassing the cache minimizes both cache pollution and the displacement of *temporal* or *spatially cached* data.

Non-temporal data is data that is accessed irregularly at long intervals. An example would be a set of vertex data elements that are read and used only once for each image frame (each refresh).

Temporal data is data that is used repeatedly, commonly from the same memory location. A typical example would be data used in a program loop.

Spatially cached data is data (or instructions) that are found in memory locations close to other recently accessed data. An example of spatially cached data would be the elements in an array or another data set where elements are accessed sequentially and stored in contiguous locations.

The `movntps` instruction moves four data elements (128 bits) from an XMM register to memory, while the `movntq` instruction moves 64 bits from a register to memory. In both cases, neither the L1 nor L2 caches are used.

Move Masked Byte(s) to Memory

`maskmovq source, mask_source`

The move masked byte(s) to memory instruction—`maskmovq`—uses a mask to write selected bytes from an MMX register directly to memory. The mask is generated from the most significant bits of each byte element in the MMX register source, with ones writing byte data to memory and zeros preventing the corresponding bytes from being written. The `edi` register identifies the location in memory where the bytes are written.

Prefetch

`prefetcht0 [esi]`

The prefetch instructions are used to retrieve data from memory to the cache prior to actual need so that the latency of memory access is reduced. Four prefetch instructions are identified by suffixes:

<code>prefetcht0</code>	Fetch temporal data into all levels of cache.
<code>prefetcht1</code>	Fetch temporal data into the first level (L1) cache.
<code>prefetcht2</code>	Fetch temporal data into the second level (L2) cache.
<code>prefetchnta</code>	Fetch nontemporal data into a nontemporal cache structure (implementation specific, uses a portion of the L1 cache).

Store Fence

`sfence`

The store fence instruction (`sfence`) is used to control the order in which data is written to memory. The `sfence` instruction ensures that all data in any store instruction prior to the `sfence` instruction has been written to memory before subsequent storage instructions can be executed. The `sfence` instruction does not affect execution of any other instructions except storage.

SIMD State Management

Because the Pentium III has incorporated new data and controllable registers that did not exist in the previous Intel architecture, new processor state functions have been added to support both the new and existing floating-point states as well as the MMX technology and XMM registers. The State Management operations include:

- Control status register
- Load and store MXCSR register
- State save and restore

SIMD Control Status Register

The Pentium III control status register (MXCSR) is used to:

Set numeric exception status flags The status flags use bits 0–5 in the MXCSR register, while the mask bits are found in bits 7–12.

Mask or unmask numeric exceptions Masked exceptions are handled within the processor, while unmasked exceptions invoke a software exception handler and must be checked for OS support.

Set the rounding mode Four rounding modes can be selected by setting bits 13 and 14 of the MXCSR register control.

- 00B Round to nearest (default rounding mode)
- 01B Round down (toward –4)
- 10B Round up (toward + 4)
- 11B Round toward zero (truncate)

Set the flush-to-zero mode The flush-to-zero mode is controlled by bit 15 of the MXCSR register. If the flag is set (one), when an underflow operation occurs, the processor returns zero with the sign of the true result and sets the precision flag (inexact result), the underflow exception flags, and mask bits in the MXCSR register. When underflow exceptions are common in applications, the flush-to-zero mode offers faster execution. Because of the loss of precision, however, this mode is not IEEE 754-compliant.

Load and Store MXCSR Register

The load and store MXCSR register instructions are used to load or store the contents of the MXCSR register to and from memory. The MXCSR register can be loaded from memory by calling `ldmxcsr`, while the `stmxcsr` instruction stores the contents of the MXCSR register in memory.

State Save and Restore

The state save (`fxsave`) and restore (`fxrstor`) instructions are used to save (store) the processor state information to memory or to restore the state information from memory when a context switch is performed. The processor state information consists of the MXCSR register, the eight XMM registers, and the eight FP/MMX registers, and requires a total of 512 bytes in memory.

Resources for the SIMD Extensions

For those interested in exploring the SIMD extensions further, a variety of resources are available providing additional information, examples, and tutorials. One excellent source for information is the *Welcome to Streaming SIMD Extensions* Web site, found on the Internet at <http://developer.intel.com/vtune/cbts/simd.htm>.

From this site, you may download the *Introduction to the Streaming SIMD Extensions Tutorial*. This tutorial offers examples of the operations of the SIMD instructions, illustrations showing SIMD data organization, and image overlay processes, as well as code examples for such operations as 3-D transformations using matrix/vector operations, alpha saturation for image blending and fade-in/fade-out, or conditional branch elimination.

A variety of application notes and code examples can be found at the *Intel® Streaming SIMD Extensions Application Notes* Web site found on the Internet at <http://developer.intel.com/vtune/cbts/strmsimd/appnotes.htm>. Further program tips, specific to the Streaming SIMD Extensions, are provided in the current edition of the *Intel Architecture Optimization Manual* and the *Intel Architecture Software Developer's Manuals* (both published by Intel). Finally, all of these resources are included in the *VTune Performance Enhancement Environment* CD (available from Intel).

WARNING:

Caution: the links referenced in this chapter are subject to change.

Summary

Potentially, the SIMD Extensions offer the capacity to take full advantage of the 64-bit instruction handling provided by the Pentium III CPUs. The disadvantage, of course, is that it is not practical to simply assume that everyone will have a PIII processor. Also, when taking advantage of the SIMD instructions, whether for graphics or simply for other numerical operations, provisions are still needed to support less capable processors.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART IV **Database Programming**

- Chapter 16: Universal Data Access Architecture
- Chapter 17: ODBC and SQL Database Access
- Chapter 18: ADO Database Access

CHAPTER 16 **Universal Data Access Architecture**

- COM-based Universal Data Access (UDA)
- OLE DB, ActiveX Data Objects (ADO), and their advantages
- Internet advantages with OLE DB and ADO
- Cross-platform support with UDA
- Data access in the Internet age

Microsoft Data Access Components (MDAC) is a set of re-distributable technologies that, collectively, implement Microsoft's *Universal Data Access (UDA)*.

MDAC is not a product in itself, but is rather a set of tools. The purpose of MDAC (as an umbrella label) is to provide a single, synchronized release for all of the key data access technologies used across Microsoft tools, applications, and platform products. Since new versions of MDAC and the release schedules for new versions are driven by the development of associated Microsoft products, 'version' releases of the MDAC tools can be expected to appear concurrently with new releases of Visual Studio, Microsoft Office, or other 'consumer' products.

Collectively, MDAC consists of the current versions of ActiveX Data Objects (ADO), Object Linking and Embedding Database (OLE DB) components, and Open Database Connectivity (ODC), all of which have

been released as an integrated tool set. Using MDAC, developers creating client/server and Web-based data-driven solutions can select whatever tools are needed from the MDAC components and can then invoke these components (using whatever applications, languages, tools, or Web browsers they choose) to create complete database solutions. This premise is the basic idea behind UDA.

The objective of UDA is that high-performance access to a variety of data and information sources, located on multiple platforms, can be provided through a convenient programming interface compatible with virtually all tools and languages. By employing such a universal approach, legacy software/hardware investments are preserved and existing technical skills are extended, allowing organizations to create easy-to-maintain solutions and use their choice of best-of-breed tools, applications, and data sources on the client, middle-tier, or server.

As a further benefit, UDA does not require the expensive, time-consuming, and, of course, error prone process of transferring corporate data from multiple existing data stores to a single, centralized repository. Likewise, using UDA/MDAC does not impose a commitment to a single vendor's products.

Instead, UDA is based on open, industry-standard specifications with broad industry support, and UDA works with all major established database products. The result becomes an extension and advancement of the contemporary standard interfaces—including ODBC, RDO and DAO—while extending the functionality of these tried and tested technologies.

Data Access SDK

The Microsoft Data Access SDK (Software Developer's Kit) is a set of tools and samples designed to help developers create solutions using MDAC. The SDK offers a single convenient source containing tools for data access component development, tools for testing and distributing components, and of course, documentation. The SDK is activity-based with content designed both for consumer and provider writers and for developers working with various languages and deployment environments.

To locate the most recent SDK version, consult the Microsoft Web site at www.microsoft.com/msdownload/defaults.asp and select *Data Access Components SDK* and the appropriate operating system.

COM-Based Universal Data Access

NOTE:

COM is discussed further in Chapters 21 through 23 (Part VI of this book).

The basis for Microsoft's Universal Data Access is found in the Component Object Model (COM), offering a common set of modern object-oriented interfaces. As the most widely implemented object technology in the industry, COM has become the *de facto* standard and provides a variety of advantages including:

- A rich set of integrated services encompassing such elements as message queuing, security, transactions, and data access supporting a broad range of application scenarios
- Tool choices from a wide variety of vendors employing a wide variety of development languages
- An extensive customer base for customizable applications and reusable components
- Established interoperability with existing infrastructures for both developers and end users
- Exposed COM-based interfaces optimized both for low- and high-level application development, including OLE DB and ADO

NOTE:

Both OLE DB and ADO are now system components that are shipped with Internet Explorer (v4.0 and later), Windows NT (service pack 4), Windows 98 and, of course, Windows 2000.

The fact that COM offers established consistency and standards for interoperability allows the Universal Data Access architecture to provide open access and to work with virtually any tool or programming language. This also helps to ensure a consistent data model across all tiers of modern application architecture.

OLE DB

NOTE:

OLE DB is discussed further in Chapter 20, “Network Support,” as an element used with ADO.

OLE DB is an open specification design for Microsoft’s strategic system-level programming interface, offering access to data across organizational networks. While ODBC (Open Database Connectivity) was designed for access to relational databases, OLE DB is the successor to ODBC with access for both relational and non-relational data sources. This access includes custom business objects, documents, email and news, file system stores, geographical data, graphics, Internet sources, mainframe ISAM/VSAM and hierarchical databases, reference materials, and more.

Functionally, OLE DB defines an assortment of COM interfaces that encapsulate management services for a variety of databases. Each COM interface enables the creation of software interfaces that implement such services. OLE DB components can appear as three basic types:

- *Data consumers* use data retrieved from data providers.
- *Data providers* contain and supply data for use by data consumers.
- *Service components* process and transport data. (Examples include query processor engines and cursor engines.)

OLE DB also includes a bridge to ODBC, so that there is continued support for (and compatibility with) the large base of ODBC relational database drivers that are presently in use. Figure 16.1 illustrates how a variety of OLE DB interfaces can provide access to different types of data sources.

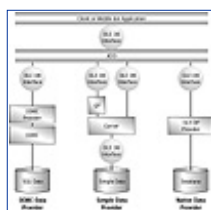


FIGURE 16.1 OLE DB components

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

OLD BD Advantage

While ODBC has been an important and successful data access standard, principally by simplifying and unifying access across databases with differing structures and syntaxes, OLE DB has built on and extended ODBC with an improved architecture yielding significant performance advantages.

ODBC providers were required to implement SQL-relational engines with cursoring and query-processing services to expose data—and to implement these for each ODBC driver. The end result is increased overhead for the ODBC driver developer as well as the end users.

In contrast, OLE DB relies on reusable service components to handle processing chores across a variety of data providers. This simplifies writing data providers and, at the same time, reduces the number of components required and decreases overhead.

ActiveX Data Objects

NOTE:

ADO is covered at greater length in Chapter 20 along with examples of database operations using ADO/OLE DB.

ADO provides an application-level programming interface to data and information (see Figure 16.1). In effect, the purpose behind ADO is to offer consistent, high-performance access to data while supporting a wide variety of development requirements including creating both front-end database clients and middle-tier business objects as well as using applications, tools, languages and Internet browsers. In short, ADO is designed to offer a single data interface satisfying single- to multi-tier client/server applications as well as Web-based, data-driven solutions.

Functionally, ADO offers a simple application-level interface to OLE DB with OLE DB offering the underlying access to the data proper. ADO offers a number of advantages, including being implemented with a small footprint, minimal network traffic requirements, and a minimal number of layers between the front end (or user application) and the actual data source. The end product becomes a lightweight, high-performance interface.

Equally, ADO is a convenient development tool because it uses the COM automation interface that is widely supported by contemporary database and rapid application development (RAD) tools as well as development languages.

NOTE:

Eventually, ADO is expected to replace both RDO and DAO by combining the best features of each and using similar conventions together with simplified semantics.

ADO Advantages

Like OLE DB, ADO is designed to maximize performance by reducing the development code required for a solution, accomplished in part by flattening the coding model.

Both DAO and RDO are hierarchical models. Because of their structures, code used to return results from a data source is required to begin at the top of the object model and to traverse each layer until the targeted recordset is reached.

In contrast, the ADO model is non-hierarchical, allowing a developer to begin by creating a recordset in code and to then retrieve results by setting properties before executing a single method to execute the query and populate the recordset with returned results. In effect, the ADO approach drastically reduces both the amount and complexity of code that must be created by the programmer. This results in less code running on the client or middle-tier business object, and produces higher performance.

Internet Advantages with OLE DB and ADO

Usually, when a client submits a data request to server, the client and server open a “channel” for communications with a series of “messages” cycling between the two. Where a ready pipeline for information exists—such as in the case of a LAN or even a WAN connection—this type of overhead is not a major penalty. For Internet transactions, however, where the effective bandwidths are more limited, the traffic required for a constant pipeline is unacceptable as well as (sometimes) being effectively unreliable.

Both OLE DB and ADO are optimized for Internet operations by implementing a “stateless” model where the client and server systems can be disconnected between data access operations.

In like fashion, MDAC includes a Remote Data Service component that provides effective marshaling of data between a middle-tier or server and the client side operation. The RDS component includes support for batch updates as well as an efficient client-side cursor engine for local data processing without a constant stream of server requests.

Cross-Platform Support with UDA

UDA is designed to provide support and access to data on all major computing platforms. Toward this end, Microsoft is actively engaged in supporting third-party development projects—using OLE DB providers—for non-Windows-based data. Figure 16.2 illustrates how ADO and OLE DB components offer connections across the enterprise.



FIGURE 16.2 Enterprise-wide connectivity

At each level, OLE DB components take the place of DLLs or other system components to provide the data connections. Because the OLE DB specifications define interfaces rather than implementations, components can be created for any environment.

While OLE DB is based on the Windows COM architecture, this does not limit OLE DB components to functioning on Windows/NT based PCs. Instead, two separate approaches provide portability to non-Windows DBMS platforms: a full port of COM (libraries and utilities are available from a number of vendors), or implementations of COM interfaces on non-Windows platforms.

Data Access in the Internet Age

In the past, DBMS systems commonly accessed structured data, often from relational databases but generally from sources with a defined format with clear rules for access. With the Internet becoming an increasingly central element in data transactions, in addition to the issues traditionally addressed by DBMS systems, today's data access also encompasses a host of new data types, new client systems and, not least, new access methods.

The Internet is only the visible tip of a large iceberg. Today's organizations are faced with the need to manage a proliferation not only of DBMS data but also non-DBMS stores that include textual and graphical data, desktop applications, mail systems, and workgroup and workflow systems, as well as others. Further, while this proliferation of non-structured data presents challenges for internal intranet access, leveraging the data available in both mainframe and minicomputer flat files while extending access to Internet-based users simply carries the problem one additional stage.

UDA is aimed at offering transactional control over varied and diverse data sources and components using the Microsoft Transaction Server (MTS). This is accomplished by employing OLE DB data sources to implement the functionality of a resource manager—handling transactions that are local to the data source—thus enabling each data source to participate in distributed transactions. For reliable operations across multiple data sources, MTS employs a distributed transaction coordinator using a two-phase commit protocol. MTS also enables applications to scale, in response to increases in user loads, with minimal additional effort.

Summary

UDA is not a product or even a group of products, but rather is a collective term applied to a strategy and a group of products that together seek to extend access beyond the realm of conventional database sources and beyond the reach of LAN/WAN systems.

To understand where data access is going, however, it's helpful to know where it is and where it has been. Therefore, in the next several chapters, we will begin with contemporary database systems accessed using ODBC and SQL and the MFC CRecordset classes before moving on to ADO and OLE DB and, into the developing realms of COM/COM+.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 17

ODBC and SQL Database Access

- Open Database Connectivity (ODBC)
- The CRecordset class
- Structured Query Language (SQL)
- Database transactions
- Updating records
- Appending records
- Modifying the field exchange operations
- Using custom SQL statements
- Debugging SQL operations

Any way you look at it, computers are basically data engines, and the biggest question is simply what form your data appears in. If you're using a word processor, the data appears as a document file or, for a spreadsheet, in a specialized (custom) database. Likewise, other applications have their own data formats. Of course, for such specialized formats, the applications necessarily know what structure is used and how to retrieve information. But there are also generalized data formats that are shared between applications, and these are commonly referred to as databases.

Although there are standards, databases (and database formats) are hardly standardized, even though the major ones in use today are relational databases. However, even within this broad classification, there remains a wide variety of formats and, consequently, a corresponding need for a means to provide applications with ready access to data across different databases without needing to know all of the details of the internal structure.

To satisfy the requirement of ready access across different formats, the original tool that evolved was Structured Query Language (SQL). Recently, SQL has been supplemented by Open Database Connectivity (ODBC). Because SQL is used with ODBC, both of these topics will be discussed in this chapter.

First, however, we need to look at what databases are and how they are organized.

Basic Database Organization

This is not the place for a detailed discussion of database structures, but a minimal understanding of database organization (specifically, relational database organization) is essential.

In a flat-file database, all information relative to an entry would be included in the entry. The result, even for a simple database, is that data is duplicated unnecessarily and the database quickly becomes bloated. For example, consider a database containing payroll information and assume that the records describe 100 employees who are paid every two weeks, with records for one year. Now if this were a flat-file database, each of the 26 pay records for each employee would also need to include their personal information (such as their address) as well as their pay rate, employment information, and various other details.

However, for a relational database, the information is organized by tables. One table might contain a list of employees with their addresses and personal data, while a separate table would contain pay records at biweekly intervals. And, of course, links between the employee and payroll tables would provide connections such that a single employee record would be linked to 26 payroll records over a one-year period—a rather obvious savings in data size.

Database Tables

Within a database, a table is similar to a spreadsheet and consists of records (rows of information) that are divided into fields (such as the individual cells in a spreadsheet). Thus, in a database table, each field contains a specific data element, commonly in a defined format, while each record (row) consists of a series of fields of related information.

Unlike a spreadsheet, each table will usually have a minimum of one field (called a *key*) that can be used to uniquely identify each record. If the key for a table is a single field, this is called a *primary key*. Alternately, if two or more fields are used to identify each record, these are known as *multi-value keys*.

Keys form the basis for relational databases, and links between tables are created by including key values from one table as fields in a second table. These link entries are known as *foreign key values*.

For example, in Figure 17.1, in the Titles table, the Title_ID field is the primary key. In the Order Details table, the Order_ID field is the primary key. The relationship between the Titles table and the Order Details table, however, is defined by the link between the [Titles].[Title_ID] field (the primary key) and the [Order Details].[Title_ID] field (the foreign key).



FIGURE 17.1 Four tables and their relationships (MSAccess)

Additional links are defined between [Order Details] and [Order] tables through the Order_ID field, which is the primary key in both tables and then between the [Orders] and [Dealers] tables through the Dealer_ID field. You should realize, however, that these links, as shown, exist only for Microsoft Access. Even though the same database and tables will be used, specific programming will be required (as discussed later) to implement the same links through an application.

Also, the database illustrated here (Publisher.mdb) is only a partial database used for a programming example. An actual database would probably contain several additional tables, relationships, and considerable additional information such as discounts, sales terms, billing records, and so on, as well as links to Supplier and Category tables that are not provided here. Still, even a simple database is sufficient to illustrate access methods and procedures.

Structured Query Language (SQL)

SQL was originally developed by IBM for use in a mainframe environment. The purpose of SQL, however, was to provide a simple, structured language for composing query statements to provide access to relational databases.

Today, of course, SQL no longer exists—not, at least, as a standalone product or package. Instead, SQL is hosted by virtually all relational databases and development languages. In Visual Studio, for example, the

Microsoft Foundation Class (MFC) uses SQL to specify queries and to perform other operations across a variety of database formats, supporting SQL operations through a variety of specialized classes such as the CRecordset and CDAORecordset classes.

As a result, when SQL is used in MFC-based programs, instead of writing complete SQL statements, queries are written as segments with MFC handling the task of assembling these partial conditions into complete SQL statement format.

TIP: The CRecordset class includes an HSTMT member that can be used to enter an SQL statement directly.

SQL Statements and MFC

While SQL proper supports a wide variety of statements and statement structures, MFC queries restrict SQL formulas to a less diverse range. Using MFC, the SQL query begins with a SELECT statement and commonly takes the form SELECT *** FROM *** WHERE *** with the SELECT statement specifying the record fields to be retrieved, the FROM statement identifying the table (or tables), and the WHERE statement providing the conditions for selecting records.

In conventional SQL, we might construct a statement taking this form:

```
SELECT DISTINCTROW "Titles"."Title", "Titles"."Author",  
"Titles"."Notes", "Titles"."Price", "Dealers"."Dealer",  
"Dealers"."Terms" FROM "Dealers" INNER JOIN "Titles" ON  
"Dealers"."Dealer_ID" = "Titles"."Dealer_ID";
```

In MFC, however, there are elements in this format that are not supported and a similar query must take a different form; for example:

```
SELECT [Titles].[Title], [Titles].[Author], [Titles].[Notes],  
[Titles].[Price], [Dealers].[Dealer], [Dealers].[Terms] FROM [Deal-  
ers],[Titles] WHERE [Dealers].[Dealer_ID] = [Titles].[Dealer_ID];
```

As you can see, the two forms are very similar but the second query does not use the terms DISTINCTROW or INNER JOIN.

TIP: Where a conventional SQL statement uses quotation marks to delimit table and field names, in VC++, the entire SQL statement is enclosed in quotes, and brackets are used to replace quotation marks within the statement. Where table or field names include spaces, either quotation or bracket delimiters are required; where spaces are absent, delimiters are optional.

Simpler statements are also possible. For example, SELECT * FROM Titles; would select all fields (and therefore all records) from the Titles table, while SELECT Titles.Title FROM Titles; would return only the Title fields from all of the records in the Titles table.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

Restricting Record Selections

Returning all records from a table, even with the response restricted to certain fields, will usually provide an excess of information. Consequently, our usual SQL queries are intended to return only some subset of the available information, with the selection being restricted by a filter statement.

For example, suppose that we want to return only records from the Titles table for books written by Manning Coles. For this purpose, an SQL statement could be written as:

```
SELECT Titles.Title FROM Titles WHERE Author = [Manning Coles];
```

NOTE: Notice that an SQL statement requires only a single equals sign (=) for evaluation, not the double equals sign (==) required by C/C++.

Alternately, we might want to write a query where two (or more) conditions must be satisfied, such as a query to select only titles written by Manning Coles where the price is less than \$10.00. For this purpose, our statement might read:

```
SELECT Titles.Title FROM Titles WHERE Author = [Manning Coles]  
AND Price < 10;
```

In other cases, queries may need to select fields from more than one table (such as `SELECT * FROM Titles, Dealers`) or, as shown in previous statements, multiple fields may be selected by separating the fields with a comma.

Also, when selections are being made from two (or more) tables, but where duplicate field names appear in separate tables, the field names must be qualified as `[Table Name].[Field]`. For example, in the Books_DB database, both the Titles and Dealers tables include fields named Dealer_ID, and a query on both tables would need to qualify the field names as `[Titles].[Dealer_ID]` and `[Dealers].[Dealer_ID]`.

TIP: Even if the duplicated field is queried from only one of the tables, the statement reference will still need to be fully qualified.

Sorting Records

While filter statements can restrict the amount of information returned to some manageable subset of the whole, you'll probably also wish to have the information returned in a particular order—and databases are rarely, if ever, in any particular order. For this purpose, the `ORDER BY` instruction is used to control the order of the returned records.

TIP: An ORDER BY instruction can be used to sort records on any field, although how entries are sorted can vary depending on the database used.

As an example, to sort titles according to price, our SQL statement might be written as:

```
SELECT Title FROM Titles ORDER BY Price;
```

An ORDER BY statement can also be modified to sort records in descending order (ascending is the default) by adding the DESC modifier, such as:

```
SELECT Title FROM Titles ORDER BY Price DESC;
```

Finally, just as an AND statement can be used to combine selection conditions, multiple ORDER BY conditions can also be specified. For example:

```
SELECT Title FROM Titles ORDER BY Date AND Price DESC;
```

As mentioned earlier, discussing the possible modifiers in SQL statements and the correct forms of statements is not our primary topic. This discussion has been provided principally as an introduction to the real subject—programming SQL with MFC and ODBC—of which the discussion follows.

NOTE: There are a variety of books published on the subject of SQL and, as any SQL programmer will be quick to inform you, this introduction to SQL has only touched briefly on the subject.

Registering a Database with ODBC

Before you can proceed with programming access to a database, you must set up a database. A couple of sample databases, `Publisher.mdb` and `Books_DB.mdb`, are provided on the CD accompanying this book. To use these sample databases:

1. Begin by copying the Publisher.mdb database to your local (or network) hard drive.
2. After copying, register the database with the ODBC facilities in Windows 2000. You should also change the file's read-only status.

NOTE: Windows 2000 includes ODBC drivers to support a variety of database formats including Access, Btrieve, dBase, FoxPro, Oracle, Paradox, and others. Since ODBC makes access essentially independent of the database used, for convenience, Microsoft Access database examples are used here.

3. Next, open the Control Panel and then select Administrative Tools.
4. From Administrative Tools, select Data Sources (ODBC). The ODBC Data Source Administrator dialog box (Figure 17.2) will appear.



FIGURE 17.2 The ODBC Data Source Administrator dialog box

5. Click the Add button. The Create New Data Source dialog box (Figure 17.3) will appear.
6. Select the appropriate driver for the database—in this case, the Microsoft Access Driver—and then click Finish. The ODBC Microsoft Access Setup dialog box will appear (Figure 17.4).
7. Click the Select button to display a file selection dialog box, then select the Publisher.mdb database. The Setup dialog box closes, and the Database field of the ODBC Data Source Administrator dialog box now contains the filename and path of the Publisher.mdb database.

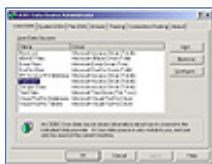


FIGURE 17.3 The Create New Data Source dialog box

8. Type a Data Source Name—“Publisher” for this example—and then click OK to complete registration.

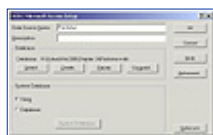


FIGURE 17.4 The ODBC Microsoft Access Setup dialog box

At this point, the Publisher.mdb database has been registered with the ODBC Administrator (see Figure 17.2) and can be accessed by applications by simply using a name reference (the Data Source Name you specified—“Publisher”) rather than the database’s fully qualified drive/path/file identifier.

Programming a Database Application

In order to demonstrate access to a database, the simplest method is to utilize a form to display the contents retrieved from the dataset. Admittedly, this is not necessarily representative of database applications, since many do not depend on form displays at all. Often database applications simply read or write data, compile reports, or perform other processing without displaying any results. Still, a form display provides a convenient illustration while the processes used remain applicable to other application types and purposes.

To design the *Publisher* demo:

1. Begin by using the MFC AppWizard to create an executable program using an SDI interface (see Figure 17.5). In step 2 (of 6), select the “Database view without file support” button.

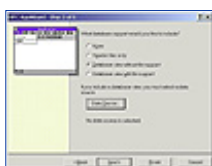


FIGURE 17.5 Adding database support to an application

2. Click the Data Source button to select a database. The Database Options dialog box appears (see Figure 17.6).

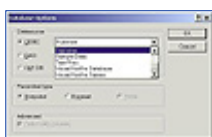


FIGURE 17.6 Selecting a Data Source

3. Since this will be an ODBC-based application, simply click the ODBC button and then select the “Publisher” data source from the pull-down list.
4. Click OK. The Select Database Tables dialog box appears, showing the tables included in the database (see Figure 17.7) where we’ll choose the Titles table for our initial data source. In a moment, other database tables will also be included in the application, but only one is required initially.

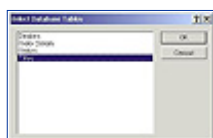


FIGURE 17.7 Selecting a database table

After the table selection has been made, AppWizard returns to Step 2 (see Figure 17.5) but now shows the selection as the table [Titles] in the data source “Publisher”.

At this point, you can proceed with AppWizard until you reach Step 6. In Step 6, AppWizard shows five

Renaming the classes and source files is optional, but changing the default names to a more representative nomenclature offers the convenience of easy identification later. Also, in a moment, four new classes (and source file sets) will be added where we will again choose representative names.

Once we've finished with AppWizard, Visual C++ will open with a blank dialog box where we need to lay out our first form. Again, the use of a form is simply for our demonstration application and may not be relevant to all applications. In this case, the form (see Figure 17.9) will display a series of values from the [Titles] table. We don't need all of the fields from [Titles] so we can pick and choose which we display.



Previous	Table of Contents	Next



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Also notice that the dialog box format has already been selected as a Child dialog box and that it has no border, title bar, system menu, or any other additions—all very plain vanilla, since the application frame will provide the controls and outline. We have, however, included one button, Show Orders, which will be used to display a subsequent form. However, before worrying about other forms and other recordsets, the first task is to finish setting up the current form and classes.

Once the form dialog box has been designed, call ClassWizard and select the Member Variables tab (see Figure 17.10) where the control IDs for the various edit fields are displayed.

Usually, when assigning member variables, you would type in a name for the variable. In this case, however, we want to link the fields to recordset members, and clicking the Add Variable button will display the Add Member Variable dialog (Figure 17.11) where the pull-down list already contains a list of the m_pSet members.

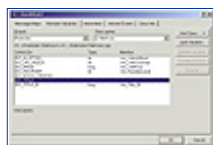


FIGURE 17.10 Assigning member variables

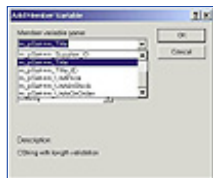


FIGURE 17.11 A list of predefined member variables

Although the m_pSet recordset member hasn't been introduced yet, it was created by AppWizard when the Titles table was selected from the sample Publisher database—as a CTitleSet member belonging to the CTitleForm class—and it contains members corresponding to each field in the table.

Here all that is necessary—immediately, at least—is to assign these members to the appropriate dialog box elements. Once this is done, when a recordset has been retrieved from the database, the corresponding values

will automatically appear in the form.

Also, once these assignments have been made, we have a working application (see Figure 17.12) that is fully functional and ready to step through the recordset entries in the [Titles] table.



FIGURE 17.12 The initial form in the *Publisher* application

Although functional at this point, the application is still rather limited and has only the capability of stepping through one table in the database. But, before extending the *Publisher* application to add new capabilities, it is also worthwhile to look at a few of the internal provisions already present.

Accessing Data: The CTitleSet Class

The CTitleSet class is derived from the CRecordset class and provides the basic functionality to access recordsets from an ODBC database table—that is, to read a record from a database (or, of course, to write a new or updated record as well).

To begin, here is the TitleSet.h header file, with the field declarations appearing in bold.

```
// TitleSet.h : interface of the CTitleSet class
//
/////////////////////////////////////////////////////////////////

class CTitleSet : public CRecordset
{
public:
    CTitleSet(CDatabase* pDatabase = NULL);
    DECLARE_DYNAMIC(CTitleSet)

// Field/Param Data
//{{AFX_FIELD(CTitleSet, CRecordset)
    long      m_Title_ID;
    CString   m_Title;
    CString   m_UnitPrice;
    int       m_UnitsInStock;
    long      m_Category_ID;
    int       m_UnitsOnOrder;
    long      m_Supplier_ID;
    int       m_QuantityPerUnit;
    int       m_ReorderLevel;
    BOOL      m_Discontinued;
//}}AFX_FIELD
```

Notice that declarations are made to match all fields in the selected table despite the fact that we may or may not need all of them. This is necessary since the CRecordset operations will read data as entire rows, rather than as selective fields.

Also, the CTitleSet class declares three public functions:

```
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTitleSet)
public:
    virtual CString GetDefaultConnect();
        // Default connection string
    virtual CString GetDefaultSQL();
        // default SQL for Recordset
    virtual void DoFieldExchange(CFieldExchange* pFX);
        // RFX support
//}}AFX_VIRTUAL
```

The third function, DoFieldExchange, simply provides the transfer mechanism between the retrieved database record and the member variables. AppWizard has already created the implementation for this function.

```
// Implementation
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
};
```

The CTitleSet.cpp source is almost equally brief, and begins with a constructor where the member variables are initialized:

```
// TitleSet.cpp : implementation of the CTitleSet class
//

#include "stdafx.h"
#include "Publisher.h"
#include "TitleSet.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CTitleSet implementation

IMPLEMENT_DYNAMIC(CTitleSet, CRecordset)

CTitleSet::CTitleSet(CDatabase* pdb)
    : CRecordset(pdb)
{
   //{{AFX_FIELD_INIT(CTitleSet)
    m_Title_ID = 0;
    m_Title = _T("");
    m_UnitPrice = _T("");
    m_UnitsInStock = 0;
    m_Category_ID = 0;
    m_UnitsOnOrder = 0;
    m_Supplier_ID = 0;
    m_QuantityPerUnit = 0;
    m_ReorderLevel = 0;
    m_Discontinued = FALSE;
    m_nFields = 10;
   //}}AFX_FIELD_INIT
    m_nDefaultType = snapshot;
}
```

Next, the GetDefaultConnect function simply reports that the data source is an ODBC database with the name (DNS) "Publisher".

```
CString CTitleSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=Publisher");
}
```

The GetDefaultConnect function can be written to supply additional information, such as a username and a password, if these are required for access to the specified data source. To hard-code these, for example, the

statement might read:

```
CString CTitleSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=Publisher;UID=[your name];"
              "PWD=[top secret]");
}
```

Here the UID designation identifies the username while PWD identifies the password entry.

Of course, a more reasonable approach, when access is restricted, would be to use a dialog box to request a username and password or to use the existing logon information. In any case, irrespective of the source of the information, the GetDefaultConnect function should assemble this data into the string format shown and then return the formatted string as its response.

Even more briefly, the GetDefaultSQL function returns the name of the table selected as:

```
CString CTitleSet::GetDefaultSQL()
{
    return _T("[Titles]");
}
```

The string returned by GetDefaultSQL is expanded automatically as the FROM portion of an SQL statement. In this example, the resulting SQL statement would become:

```
SELECT * FROM [Titles];
```

Like the GetDefaultConnect function discussed earlier, the GetDefaultSQL function can also be amplified to impose additional conditions. For the moment, however, the present statement is adequate and other examples will appear later in this chapter.

Also, more immediately, you will see how WHERE and ORDER BY statements are added to the SQL statement.

Transferring Data from the Database to the Recordset

The DoDataExchange member function of the CTitleSet class is responsible for transferring data from the database to the recordset and vice versa. The implementation provided by AppWizard appears as:

```
void CTitleSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CTitleSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
```

The first instruction—pFX->SetFieldType—sets the direction for the information exchange with the mode specification—outputColumn—indicating that the transfer will occur from the database recordset to the named member variables. The specific field and variable exchanges are:

```
        RFX_Long(pFX, _T("[Title_ID]"),      m_Title_ID);
        RFX_Text(pFX, _T("[Title]"),         m_Title);
        RFX_Text(pFX, _T("[UnitPrice]"),     m_UnitPrice);
        RFX_Int( pFX, _T("[UnitsInStock]"),   m_UnitsInStock);
        RFX_Long(pFX, _T("[Category_ID]"),    m_Category_ID);
        RFX_Int( pFX, _T("[UnitsOnOrder]"),    m_UnitsOnOrder);
        RFX_Long(pFX, _T("[Supplier_ID]"),    m_Supplier_ID);
        RFX_Int( pFX, _T("[QuantityPerUnit]"), m_QuantityPerUnit);
        RFX_Int( pFX, _T("[ReorderLevel]"),   m_ReorderLevel);
        RFX_Bool(pFX, _T("[Discontinued]"),   m_Discontinued);
    //}}AFX_FIELD_MAP
}
```

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Notice that several different RFX_XXXX functions are used here—RFX_Long, RFX_Text, RFX_Int and RFX_Bool. Each of these transfer functions is used to exchange different types of data elements, and the list shown here does not by any means exhaust the range of RFX_XXXX functions available.

In general, however, AppWizard is perfectly capable of selecting the appropriate transfer mechanism based on the field types found in the selected database. Of course, since AppWizard also defines the member variables that receive the corresponding data, there appears to be little chance for a mismatch.

However, if you find reason to change the database fields after creating an application, you would then have ample reason for modifying the variable declarations and the exchange operation types. And, if none of the preceding exchange operations are appropriate, remember that there are a variety of other choices described in your online documentation.

Displaying Data: The CTitleForm Class

While the CTitleSet class handles data exchange with the database, the CTitleForm class handles the task of displaying the retrieved data, but also handles such tasks as supplying additional parameters to the SQL statement. These additions can include the provisional WHERE clause and the ORDER BY statement, governing how the data is sorted.

Beginning with the TitleForm.h header, there are a few elements worth noticing.

```
// TitleForm.h : interface of the CTitleForm class
//
////////////////////////////////////
class CTitleSet;
```

First, the class CTitleSet is declared in the CTitleForm header, making the recordset class available to the form.

```
class CTitleForm : public CRecordView
{
```

The view class for our recordset must be derived from the CRecordView class so that the derived class can be customized to display the specific fields and format desired.

Notice also that the constructor for the class has been declared as protected:—a normal enough default provision, since class constructors are usually never called directly.

```
protected: // create from serialization only
    CTitleForm();
    DECLARE_DYNCREATE(CTitleForm)
```

Presently, however, we will change the declaration to make the constructor public: so that we can create our own views.

Next, an explicit instance of the CTitleSet class is declared as the member variable m_pSet, mentioned previously when elements of m_pSet were assigned to dialog box controls (or, more specifically, to edit boxes in the form dialog box).

```
public:
   //{{AFX_DATA(CTitleForm)
    enum { IDD = IDD_PUBLISHER_FORM };
    CTitleSet* m_pSet;
    //}}AFX_DATA
```

A link to the application's document class is also provided, although this is not needed immediately.

```
// Attributes
public:
    CPublisherDoc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CTitleForm)
public:

    virtual CRecordset* OnGetRecordset();
```

The OnGetRecordset function is used to query CTitleSet, returning an instance of the CRecordset class containing data retrieved from the data source.

```
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support
    virtual void OnInitialUpdate();
        // called first time after construct
```

And, last, the OnInitialUpdate function is called after the CTitleForm class is initialized to perform an initial update of the form view but also to set the WHERE and ORDER BY conditions passed to the SQL statement. Further, the OnInitialUpdate function (discussed in the following section) is used to initialize the m_pSet member that the constructor has set as NULL.

The OnInitialUpdate Function

The OnInitialUpdate function begins rather briefly with provisions to initialize the m_pSet member (an instance of the CTitleSet class) before calling the parent (CRecordView) OnInitialUpdate function.

```
void CTitleForm::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_titleSet;
    CRecordView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
```

```

        ResizeParentToFit();
    }

```

The final two provisions simply resize the parent frame to fit the supplied form dialog box. In a few minutes, however, the `OnInitUpdate` function will be modified to add additional instructions.

The OnGetRecordset Function

The `OnGetRecordset` function (supplied by AppWizard) is brief and simply returns the `m_pSet` member, which was initialized in the `OnInitUpdate` function.

```

CRecordset* CTitleForm::OnGetRecordset()
{
    return m_pSet;
}

```

Setting the Sort Order

Now that we've briefly examined the default code—supplied for the most part by AppWizard—we can look at adding additional provisions to the application, beginning by adding an `ORDER BY` instruction to sort the results reported.

```

void CTitleForm::OnInitUpdate()
{
    m_pSet = &amp;GetDocument()->m_titleSet;
    m_pSet->m_strSort = "[Title]";
    CRecordView::OnInitUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
}

```

NOTE: Because a major part of the source code is generated by AppWizard or ClassWizard, changes and additions to the code here and following appear in **boldface type** for ease of identification.

The `m_strSort` member (from the `CRecordSet` class) is used to specify an `ORDER BY` clause, which governs how records are sorted during an `Open` or `Requery` call. The `m_strSort` member must be set after the recordset object is constructed but before calling the `Open` member function.

The ODBC SQL syntax for an `ORDER BY` clause is:

```
ORDER BY "field name" [, "field name2"]...
```

For an `m_strSort` specification, the `ORDER BY` specification is supplied by default and not included in the statement. Thus the corresponding `m_strSort` format would be:

```
m_strSort = [field name], [field name2], ...
```

A recordset can be sorted by one or more fields (columns) by separating field specifications with commas while the sort order occurs in the order the fields are specified. Also, individual fields can be set to sort in ascending (default) or descending order by appending `ASC` or `DESC` to each field listed.

TIP: The number of columns that can be listed depends on the data source used. For additional information, refer to Microsoft's *ODBC SDK Programmer's Reference*.

Setting the Window Caption

Before continuing by adding additional tables (and forms to display the table contents), a small provision to modify the window caption and show the current table name would be in order. And again, the `OnInitUpdate` function is the appropriate location, thus:

```
void CTitleForm::OnInitUpdate()
```

```

{
    m_pSet = &amp;GetDocument()->m_titleSet;
    m_pSet->m_strSort = "[Title]";
    CRecordView::OnInitialUpdate();
    // set the document title
    if( m_pSet->IsOpen() )
    {
        CString csTitle = _T( "Table Name: " );
        CString csTable = m_pSet->GetTableName();

```

The GetTableName function in the CRecordset class simply returns the name of the current table, assuming that the recordset is based on a single table. Alternately, if the recordset is based on a join of multiple tables or on a predefined query (such as a stored procedure), GetTableName returns an empty string.

```

        if( ! csTable.IsEmpty() )
            csTitle += csTable;
        GetDocument()->SetTitle( csTitle );
    }
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
}

```

Now, with these provisions, the *Publisher* demo is again ready to compile and execute. The differences are simple: Now the records will be returned in sorted order, and the window caption will reflect the table name.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Adding Additional Table Access

While the existing application shows records from a single table in the database, the form used also includes a button labeled 'Show Orders'. In this procedure, we will add two new classes and a new form to display the contents of the [Orders] and [Order Details] tables, selecting—naturally—only orders for the title selected from the original [Titles] table.

1. To add a new class, call the ClassWizard and click the Add Class button, and then click New. In response, the New Class dialog box (Figure 17.13) appears.

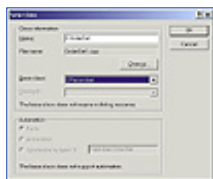


FIGURE 17.13 Adding a new class

2. Type the class name (COrderSet) and then select the base class (CRecordset) from the pull-down list. Click OK. The Database Options dialog box (shown earlier in Figure 17.6) appears.
3. Again, select the Publisher data source from the list and click OK. The Select Database Tables dialog box (Figure 17.14) appears.

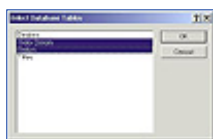


FIGURE 17.14 Selecting tables for a join

4. When you created the first table recordset class (CTitleSet) only one table was selected, the [Titles] table. For the COrderSet, however, you select two tables as a join—[Order Details] and [Orders]. After selecting the tables, the Member Variables tab in the MFC ClassWizard (Figure 17.15) shows how the joined table fields are mapped to member variables.

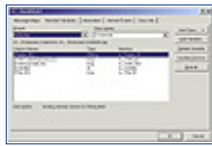


FIGURE 17.15 Member variables from joined tables

Because both the [Orders] and [Order Details] tables have fields named [Order_ID], the entries in the source file (and in the ClassWizard dialog) are fully qualified by combining the table name and field name as shown.

TIP: After creating a CRecordset-derived class—but before creating a corresponding CRecordView class—you should first design the form dialog.

Providing Record Views

Next, just as AppWizard provided a CRecordView class (CTitleForm) for our original table, we need to create a second class to view the [Orders] and [Order Details] recordset. The process is essentially the same as that followed for COrderSet except that we'll call this class COrderForm, it will be derived from the CRecordView class, and it will use the dialog identified as IDD_ORDERS_FORM (see Figure 17.16).

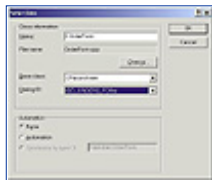


FIGURE 17.16 Adding the COrderForm class

Once the class has been defined, ClassWizard also asks you to identify the appropriate recordset class to link to the form, as shown in Figure 17.17.

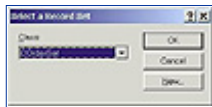


FIGURE 17.17 Linking the COrderSet Class

After these classes have been created, you still need to assign links between the COrderSet recordset members and the dialog elements in the COrderForm dialog. Once these links have been created, there is still one more table to display—the [Dealers] table.

One More Table, Two More Classes

For the [Dealers] table, we need a CRecordset-derived class titled CDealerSet and a CRecordView class titled CDealerForm that uses the dialog IDD_DEALER_FORM and is linked to CDealerSet. The procedure is essentially the same as the procedure for the [Orders] and [Order Details] tables except, of course, that only one table is used in this instance.

Modifying Classes after Creation

If you make a mistake while creating a class or if you find a need to modify a class after creation, the Class Info tab in ClassWizard has facilities to allow some elements of the class information to be modified. For an example, see Figure 17.18.

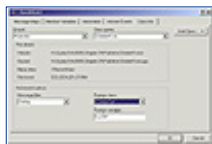


FIGURE 17.18 Examining class information

At the top of the dialog, pull-down lists allow selection of the project (since a workspace may contain more than one project) and the class name. In the center, the file details pertaining to a class are shown but cannot be modified here. At the bottom, however, you may choose the type of message filter used by the class, select an associated foreign class (if any and if relevant), and, optionally, rename the foreign variable used to create an instance of a foreign class.

Modifying the New Classes

Now that the initial files have been created for our new classes, we are going to make a number of modifications.

First, however, for both the COrderForm and CDealerForm classes, the default destructors in each class require modification. As created by ClassWizard, the default destructor for COrderForm appears as:

```
COrderForm::~COrderForm()
{
    if (m_pSet)
        delete m_pSet;
}
```

While this is a perfectly normal destructor format, it is inappropriate in this application and both the if and delete statements should be removed, leaving an empty function.

Since the COrderSet or CDealerSet members (identified as the m_pSet variable) are created and deleted by the framework, these provisions for deletion are unnecessary. And, if they are not removed, the program will fault on an assertion failure when closed, when an attempt will be made to delete each of these objects a second time.

NOTE: If you prefer not to delete the entries, simply comment them out.

Customizing COrderSet and COrderForm

In the COrderSet source files, the default SQL SELECT statement appears as:

```
CString COrderSet::GetDefaultSQL()
{
    return _T("[Order Details],[Orders]");
}
```

Notice that two tables are specified. However, as the code stands—as generated by ClassWizard—a query will return all possible combinations of records from the two tables. To limit the recordsets resulting from a query and restrict these to a subset that contains relevant information, we need to add a WHERE clause to the SQL query. And, to ensure that the [Order Details] information reported matches the [Orders] information (and vice versa), we can specify this restriction:

```
WHERE [Orders].[Order_ID] = [Order Details].[Order_ID]
```

With this condition applied, the joined tables will return only records where both tables have the same values in the [Order_ID] field.

Likewise, since we will begin with the [Titles] table, we also want the order information reported to be restricted to the orders relevant to the currently selected title (from the [Titles] table). For this restriction, the WHERE clause would be written as:

```
WHERE [Title_ID] = ?
```

Using this format, the question mark (?) represents a parameter (not a wildcard) and this says that we want only records from the [Order Details] field where the [Title_ID] field matches a supplied parameter value.

This also means that we will need to define a parameter (in the CTitleSet class) that will provide the necessary value to be matched, but, once this is done, the parameter will be substituted in the DoFieldExchange operation in COrderSet.

Combining these values, we have a new WHERE condition that appears as:

```
WHERE [Title_ID] = ? AND
      [Orders].[Order_ID] = [Order Details].[Order_ID]
```

Before we add the WHERE condition, however, we need to first make provisions for the parameter used to match the [Title_ID] field.

Defining the Filter Parameter The first step, in the OrderSet.h header, is to declare m_TitleIDParam as a long variable:

```
class COrderSet : public CRecordset
{
public:
    COrderSet(CDatabase* pDatabase = NULL);
    DECLARE_DYNAMIC(COrderSet)
    long m_TitleIDParam;
```

And the second step, in OrderSet.cpp, is to modify the constructor:

```
COrderSet::COrderSet(CDatabase* pdb)
    : CRecordset(pdb)
{
   //{{AFX_FIELD_INIT(COrderSet)
    m_Order_ID = 0;
    m_Quantity = 0;
    m_Title_ID = 0;
    m_Order_ID2 = 0;
    m_Dealer_ID = _T("");
    m_nFields = 5;
    //}}AFX_FIELD_INIT
    m_TitleIDParam = 0L;           // initialize the parameter
    m_nParams = 1;                // set number of parameters
    m_nDefaultType = snapshot;
}
```

NOTE: The m_nParams variable is inherited from the CRecordset base class.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

The OnFieldExchange member also requires modification:

```
void COrderSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(COrderSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[Order Details].[Order_ID]"),
        m_Order_ID);
    RFX_Int(pFX, _T("[Quantity]"), m_Quantity);
    RFX_Long(pFX, _T("[Title_ID]"), m_Title_ID);
    RFX_Long(pFX, _T("[Orders].[Order_ID]"), m_Order_ID2);
    RFX_Text(pFX, _T("[Dealer_ID]"), m_Dealer_ID);
    //}}AFX_FIELD_MAP
    // set the field type as a parameter
    pFX->SetFieldType( CFieldExchange::param );
    RFX_Long( pFX, _T( "TitleIDParam" ), m_TitleIDParam );
}
```

Two instructions have been added to the ClassWizard-generated code to specify that m_TitleIDParam is a filter parameter. The first instruction sets the mode for the following RFX_() call (or calls), such that the third parameter in any successive RFX_() call is interpreted as a parameter that replaces a question mark (?) in the recordset filter.

TIP: Since the param mode causes the second argument in the RFX_() exchange to be ignored, the second argument may be NULL or, as shown, may appear as text used as a comment.

In this fashion, we could have more than one parameter in the filter, and each question mark (?) in order (from left to right) would be replaced with the next parameter argument. This also means that the RFX_() field exchange statements must appear in the same order as the arguments.

Initializing COrderForm While the default SQL statement appears in the COrderSet class (OrderSet.cpp), the filter statement is supplied in the COrderForm class (OrderForm.cpp) as part of the OnInitialUpdate function.

However, while we have constructed a WHERE clause for an SQL statement, using MFC, the WHERE clause is replaced by the `m_strFilter` member of the `CRecordset` class. As an `m_strFilter` assignment, the filter condition would be written as:

```
m_pSet->m_strFilter =  
    "[Title_ID] = ? AND "  
    "[Orders].[Order_ID] = [Order Details].[Order_ID]";
```

TIP: Remember, the `m_strFilter` argument is reassembled to become the WHERE clause in the SQL statement, but the keyword WHERE itself is not required.

Before modifying the `OnInitialUpdate` function, there are a few other provisions that need to be handled, beginning with the `CPublisherDoc` class where the `OrderSet` header file needs to be added to the `#include` list:

```
// PublisherDoc.cpp : implementation of the CPublisherDoc  
// class
```

```
#include "stdafx.h"  
#include "Publisher.h"  
#include "TitleSet.h"  
#include "OrderSet.h"    // add the OrderSet header  
#include "PublisherDoc.h"
```

Next, in the `PublisherDoc.h` header, a public member variable is declared as an instance of the `COrderSet` class:

```
public:  
    CTitleSet m_titleSet;  
    COrderSet m_orderSet;    //add the COrderSet member
```

Also, while you're in the `PublisherDoc.h` header file, add another `#include` statement:

```
#include "TitleSet.h"  
#include "OrderSet.h"
```

And, in `OrderForm.cpp`, add an `#include` statement for the `PublisherDoc.h` header as:

```
#include "stdafx.h"  
#include "Publisher.h"  
#include "PublisherDoc.h"  
#include "OrderForm.h"
```

With these provisions in place, we're ready to modify the `OnInitialUpdate` function in `OrderForm.cpp`.

```
void COrderForm::OnInitialUpdate()  
{  
    BeginWaitCursor();  
    CPublisherDoc * pDoc = (CPublisherDoc*) GetDocument();
```

This initial addition creates a pointer to the `CPublisherDoc` class through the `GetDocument` function. Although this is optional, using the `pDoc` pointer in subsequent statements is more convenient than repeated calls to the `GetDocument` function.

The next step is to use the `pDoc` pointer to retrieve a pointer to the order recordset:

```
m_pSet = &amppDoc->m_orderSet;
```

In like fashion, the database opened for the Titles recordset is used for the orders dataset as well:

```
m_pSet->m_pDatabase = pDoc->m_titleSet.m_pDatabase;
```

Reusing the recordset from `CTitleSet` offers two advantages: There is no need to close and reopen the database, and in the next instruction, the current title ID from the previous form is still available.

Now we can set the current title ID—from the current CTitleSet instance—as the parameter to be used in the constructed SQL statement:

```
m_pSet->m_TitleIDParam = pDoc->m_titleSet.m_Title_ID;
```

Finally, the filter specification is stated:

```
m_pSet->m_strFilter =  
    "[Title_ID] = ? AND "  
    "[Orders].[Order_ID] = [Order Details].[Order_ID]";
```

The remainder of the OnInitialUpdate function—including the provisions to re-caption the window—are supplied by the ClassWizard:

```
GetRecordset();  
CRecordView::OnInitialUpdate();  
if (m_pSet->IsOpen())  
{  
    CString strTitle =  
        m_pSet->m_pDatabase->GetDatabaseName();  
    CString strTable = m_pSet->GetTableName();  
    if (!strTable.IsEmpty())  
        strTitle += _T(":") + strTable;  
    GetDocument()->SetTitle(strTitle);  
}  
EndWaitCursor();  
}
```

Switching Views

Now that we've added a second form (view) to the application and have provided code to select records from the [Orders] and [Order Details] tables that correspond to the previous selection from the [Titles] table, we still need a means to switch between the two views.

The CMainFrame class controls which form is displayed and handles the switching between forms, but first we need a method of identifying the views and of identifying which view is currently being displayed. As a first provision, in the MainFrame.h header, a member variable m_CurrentFormID is declared as:

```
// Attributes  
public:  
    UINT m_CurrentFormID;
```

Next, in the MainFrame.cpp source, the default constructor is modified to initialize m_CurrentFormID:

```
CMainFrame::CMainFrame()  
{  
    m_CurrentFormID = IDD_PUBLISHER_FORM;  
}
```

The constant IDD_PUBLISHER_FORM identifies the form used to display the [Titles] table. In like fashion, IDD_ORDERS_FORM will be used for the COrderForm display, and IDD_DEALERS_FORM for the CDealersForm.

The next step is to declare a new function in MainFrame.h:

```
protected: // create from serialization only  
    CMainFrame();  
    DECLARE_DYNCREATE(CMainFrame)  
    void SelectForm( UINT FormID );
```

Then, in MainFrame.cpp, the function can be implemented thus:

```
void CMainFrame::SelectForm( UINT FormID )
{
    CView* pOldView = GetActiveView();
    CView* pNewView = (CView*) GetDlgItem( FormID );
```

The first task handled is to retrieve a pointer to the active (current) view and then a pointer to the new view. If the new view (pNewView) is NULL—in other words, if the view has not been created yet—then the next routine will create an instance of either COrderForm or CDealersForm.

```
if( pNewView == NULL )
{
    switch( FormID )
    {
        case IDD_ORDERS_FORM:
            pNewView = (CView*) new COrderForm;
            break;

        case IDD_DEALER_FORM:
            pNewView = (CView*) new CDealerForm;
            break;

        default:
            AfxMessageBox( "Invalid Form ID" );
            return;
    }
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Notice that no provisions are made to create an instance of the CTitleForm class, since this view was created when the application started. Likewise, once an instance of the desired class has been created for either of the remaining forms, the instance and the view will not be deleted and there won't be any need to recreate these later.

After creating a CView-derived class instance, a context is derived from the existing view and the class instance's Create function is called to create the view and to perform the initial update for the class.

```
CCreateContext    context;
context.m_pCurrentDoc = pOldView->GetDocument();
pNewView->Create( NULL, NULL, 0L,
                CFrameWnd::rectDefault,
                this, FormID, &context );
pNewView->OnInitialUpdate();
}
```

Finally, the new view is set as the active view, the old view is inactivated (hidden), the m_CurrentFormID member is reset to the current FormID, and last, RecalcLayout tries to resize the frame to fit the dialog form.

```
SetActiveView( pNewView );
pOldView->ShowWindow( SW_HIDE );
pNewView->ShowWindow( SW_SHOW );
pOldView->SetDlgCtrlID( m_CurrentFormID );
pNewView->SetDlgCtrlID( AFX_IDW_PANE_FIRST );
m_CurrentFormID = FormID;
RecalcLayout();
}
```

Of course, by itself, the CMainFrame::SelectForm function isn't everything needed to change views. In addition, member functions need to be added to the CTitleForm, CorderForm, and CDealerForm classes to respond to the buttons in each form.

Since the necessary member functions will differ only in the details of the declaration and the argument used,

the OnShowOrders function from TitleForm.cpp will serve to illustrate all.

```
void CTitleForm::OnShowOrders()
{
    ( (CMainFrame*)
      GetParentFrame() )->SelectForm( IDD_ORDERS_FORM );
}
```

The function should be self-explanatory, since all that occurs is to call the SelectForm function from the CMainFrame class with an argument to select the desired view.

Optionally, you might also add OnActiveView functions to each of the three form classes and then provide code to update the window title each time the view is changed. The example from the CTitleForm class offers a typical sample:

```
void CTitleForm::OnActiveView( BOOL bActivate,
    CView* pActivateView, CView* pDeactivateView )
{
    if( bActivate )
    {
        // update the window caption
        CString csTitle = _T( "Table Name: " );
        CString csTable = m_pSet->GetTableName();
        if( ! csTable.IsEmpty() )
            csTitle += csTable;
        GetDocument()->SetTitle( csTitle );
    }
    CRecordView::OnActiveView( bActivate, pActivateView,
        pDeactivateView);
}
```

Is It Safe to Switch?

Before switching forms, it can be helpful to know whether there are corresponding records to display when the new form appears. For example, after stepping through the title records and selecting a title, clicking the Show Orders button is intended to change forms to show orders for this title. But what if there aren't any orders?

To find out, before switching views, we add a CheckOrders function to the COrderForm class that returns TRUE if there are any corresponding order records or FALSE if there are none.

The CheckOrders function is fairly simple and begins in a fashion very similar to the OnInitialUpdate function by getting a pointer to the document and then setting the m_TitleIDParam value before querying the database.

```
BOOL COrderForm::CheckOrders()
{
    CPublisherDoc* pDoc = (CPublisherDoc*) GetDocument();
    if( ! m_pSet->IsOpen() )    // dataset isn't open
        return FALSE;         // return failure
    m_pSet->m_TitleIDParam = pDoc->m_titleSet.m_Title_ID;
    m_pSet->Requery();          // query database
```

Following the query, the only thing that needs to be known at this point is whether or not at least one matching record was found.

```
    if( m_pSet->IsEOF() )      // nothing found,
        return FALSE;         // return failure
```

If no matches were found—in other words, the end of the database was reached as indicated by the IsEOF function—then the function simply returns, reporting failure.

On the other hand, if the end of the database has not been reached, a minimum of one match has been found and the function can return reporting success.

```

        else                                // match found,
            return TRUE;                    //      return success
    }

```

We really don't care at this point how many matches were found, only whether or not any were. Once this has been decided, then we know whether to switch views or to stay as we were.

Of course, we also need to include a provision to call the test, and this is done in `MainFrm.cpp`, in the `SelectForm` function, after ensuring that the new view has been created.

```

    CCreateContext    context;
    context.m_pCurrentDoc = pOldView->GetDocument();
    pNewView->Create( NULL, NULL, 0L,
        CFrameWnd::rectDefault, this, FormID, &context );
    pNewView->OnInitialUpdate();
}
if( FormID == IDD_ORDERS_FORM &&
    ! ((COrderForm*)pNewView)->CheckOrders() )
{
    AfxMessageBox( "No orders found for this title" );
    return;
}

```

The test is performed only when we are going to the Orders form. If we're already in the Orders form and choose to display the Dealers form, we don't worry about whether there are any corresponding entries—it would be pretty odd to have an order without a matching recipient. Or, more accurately, the absence of a corresponding dealer entry would be an error in the data, not in the application.

Likewise, if we're returning to the Titles form, we aren't looking for a specific title to correspond to a dealer or an order, we're simply going back to the entire Titles list. Therefore, no check is required.

Updating and Adding Records

Thus far, we've looked at how to register an ODBC database and we've talked about how to use ClassWizard to create `CRecordset`-derived classes to read records from a database. We've also discussed how to sort records and how to select specific records by defining joins between tables.

Simply reading a database, however, is only half the story. In addition to retrieving records, applications also need to be able to edit existing records and to append new records. For this purpose, we'll begin by registering a second database with ODBC—`Books_db.mdb`, included on the CD—under the name "Book List". The process, of course, is the same as illustrated earlier so there is no need to repeat the details here.

Likewise, the basic operations for creating the *BookList* demo program are also the same, initially, so we'll cover only the differences here. However, this is not intended to imply that there are no differences—there are differences, and they are not minor.

The `Books_db.mdb` database was chosen for this example to demonstrate a number of aspects of database access that the `Publisher.mdb` database (used earlier) did not illustrate. The `Publisher.mdb` database, for example, contained only "standard" data fields—that is, data fields that were brief text entries (with emphasis on "brief"), Boolean entries, or numerical values. However, as you will see presently, not all databases are quite so conveniently behaved. Failure to recognize potential conflicts—and failure to modify your program code accordingly—can be more than a little frustrating.

First, however, before going into the details of programming, a little more background on database (`CRecordset`/`SQL`) operations is in order.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

CRecordset Update Transactions

Earlier, the RFX_() function calls—in the DoFieldExchange function of our CRecordset-derived classes—were used to read data from the database. These same RFX_() functions, however, are also used to write updated or added records into the database. In addition, the RFX_() functions used to write to the database are precisely the same as those used to read from the database.

For example, the following DoFieldExchange function from BookListSet.cpp was generated by AppWizard and shouldn't look very different from the corresponding functions shown previously.

```
void CBookListSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CBookListSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[Title_ID]"), m_Title_ID);
    RFX_Text(pFX, _T("[Title]"), m_Title);
    RFX_Text(pFX, _T("[Author]"), m_Author);
    RFX_Text(pFX, _T("[Notes]"), m_Notes );
    RFX_Text(pFX, _T("[Price]"), m_Price);
    RFX_Text(pFX, _T("[Reference]"), m_Reference);
    RFX_Text(pFX, _T("[Dealer]"), m_Dealer);
    //}}AFX_FIELD_MAP
}
```

The only real difference between reading records from the database and writing or appending to records does not appear in the RFX_() functions. Instead, like the DDX_() operations used to copy data to and from dialog controls, the RFX_() functions are bi-directional with the direction of transfer controlled by other operations.

In this case, the relevant operations are found in five member functions provided by the CRecordset class that are used to support update operations.

AddNew The AddNew function initiates the process of appending a new record to a database. If a new record cannot be appended, for whatever reason, a CDBException event occurs.

Edit The Edit function initiates the process of updating an existing record. If the record can not be updated, for whatever reason, a CDBException event occurs.

Delete The Delete function is used to delete the current record by creating and executing an SQL Delete operation. If the record cannot be deleted—for example, if the database is read-only—a CDBException event occurs. Following a Delete operation, all data members (fields) in the recordset are set to NULL values (the database equivalent of an empty record), and a new record must be selected before any other operation can be performed on the recordset.

Update The Update function is used to conclude an AddNew or Edit operation. If no records are updated or appended or if any other error occurs, a CDBException event occurs.

CancelUpdate The CancelUpdate function is used to cancel either a pending AddNew or Delete operation. No exceptions are generated by the CancelUpdate function.

None of these five functions are called with any form of parameters, but, because four of the five generate exceptions when errors occur, a try/catch block (as illustrated in the *BookList* program) should be used to prevent your application from abruptly terminating on errors.

Editing an Existing Record

The sequence for editing (modifying) an existing record is:

1. Select the record to edit.
2. Call Edit to initiate operations. The current values from the recordset field members are saved in a buffer, allowing a CancelUpdate operation to restore the original values.
3. Enter new values for one or more field data members.
4. Call the Update function to complete the operation and ensure that the modified data is written to the database.

NOTE: If Edit is called a second time, after making changes but before calling Update, the recordset's field data members are restored to the values they had prior to the original Edit function call. Likewise, scrolling to a new record before calling Update leaves the original record unchanged.

Appending a New Record

To add (append) a new record to a database:

1. Call the AddNew function to prepare a new empty record using the recordset's field data members. Initially, all fields are set to Null.

TIP: In database terminology, the term “Null” means simply “having no value” and may indicate a reserved flag value rather than the fields actually being 0 or empty strings. Thus, the term “Null” applied to a database does not necessarily correspond to the NULL used in C++.

2. Enter new values for all field data members.
3. Call the Update function to complete the operation and ensure that the modified data is written to the database.

TIP: The SetFieldNull function can be called to set an empty field in either an Edit or AddNew operation.

Checking the Legality of Operations

No, the database police will not come knocking on your door if you attempt illegal database operations. (For one thing, they never wait to knock...) It is worthwhile, however, to perform a few brief checks before attempting certain operations to determine if those operations are valid and legal. While the CRecordset class offers a number of checks for querying supported attributes, three of these are particularly useful:

CanUpdate The CanUpdate function returns TRUE if a database can be updated or FALSE if the database is not accessible for writing (if, for instance, the database file was flagged “read only” or if the database was opened using the CRecordset::readOnly parameter.)

CanAppend The CanAppend function reports TRUE if a database can accept new records or FALSE if the database is not accessible for writing.

CanRestart The CanRestart function reports TRUE if the database can be refreshed using the Requery function.

In addition to questioning whether or not functions are permitted, there are several other useful functions employed to check the status of a database. These include:

IsOpen The IsOpen function is used to query whether the database is opened or not. Examples of the IsOpen function can be found in the *Publisher* and *BookList* example.

IsBOF The IsBOF function reports TRUE if the current position is before the first record in the database (and, consequently, there is no current record).

IsEOF The IsEOF function reports TRUE if the current position is after the last record in the database (and, again, there is no current record).

IsDeleted The IsDeleted function reports TRUE if the current position is on a deleted record.

Other query functions include CanScroll, CanBookmark, and CanTransact. (Refer to the CRecordset documentation for details on these and other functions.)

Record Locking

Record locking is used to prevent other users from accessing a table row while the locked record is being updated. Two locking modes are supported: optimistic and pessimistic.

CRecordset::optimistic Optimistic record locking (default mode) locks the record only while the Update function is operating and minimizes the period during which the record is inaccessible.

CRecordset::pessimistic Pessimistic record locking locks the record as soon as the Edit function is called and maintains the lock until the Update function concludes or until the update is aborted.

To change locking modes, call the SetLockingMode function with the appropriate argument.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Transactions

In the context of a database, transactions are used to allow operations to be undone if necessary. For example, if we wished to perform several related operations on different tables, we might initiate a transaction before beginning the updates, and then, if any problems were encountered, the transaction could be rolled back (abandoned) without needing to undo the individual operations.

Using transactions, the database system manages processing the data operations and records recovery information during the process such that any changes or additions to the data can be undone in the event that a problem occurs before the transaction is concluded.

Transactions are supported by the CDatabase class (not the CRecordset or CDAORecordset classes), which provides the database connection. Transaction processing requires three member functions in CDatabase:

BeginTrans The BeginTrans function initiates a database transaction. Subsequent to calling BeginTrans, all recordset operations become part of the transaction until the transaction is concluded by calling CommitTrans or is abandoned by calling Rollback.

CommitTrans The CommitTrans function commits (concludes) a transaction. When CommitTrans is called, all operations on the recordset that were part of the transaction are committed and become part of the database.

WARNING: If CommitTrans returns FALSE, an error has occurred and the state of the data source is undefined.

Rollback Rolls back all of the data operations that have been executed since the BeginTrans function was called. This restores the database to the point before the transaction was initiated.

The sequence of operations for a transaction is very simple:

1. BeginTrans initiates the transaction.
2. AddNew, Delete, Edit, and Update operations are called to perform operations on the database.
3. CommitTrans is called to conclude the transaction (or Rollback is called to abort the operation).

Transactions can have a couple of minor complications, since a CommitTrans or Rollback call can cause the position of the current record to be lost. After a CommitTrans operation, the GetCursorCommitBehavior function

should be called, and following a Rollback, GetCursorRollbackBehavior would be called. In each case, a returned int value offers the appropriate instruction and can be one of three values:

SQL_CB_PRESERVE The recordset's connection to the data source has not been affected by the Commit or Rollback operation and no action is required.

SQL_CB_CLOSE The recordset's connection to the data source is no longer valid and the Requery function should be called to restore the current position in the recordset.

SQL_CB_DELETE The recordset's connection to the data source is no longer valid and the recordset must be closed (call the CRecordset::Close function) and reopened before any further operations.

Refer to the Visual Studio online documentation for further details.

The *BookList* Example

In the previous database example (*Publisher*), when the CRecordset-derived class was created, a snapshot recordset type was selected. At the time, we were concerned only with reading the database and we knew that the database would not be accessed by anyone else while we were using it. For the *BookList* demo program, to create the CBookListSet class, we'll use the Book List ODBC database, but this time we're going to select the Dynaset recordset type and, from the database tables, we'll selected the [Titles] table.

NOTE: Since the MFC database classes support only read-only access to joins on multiple tables, in order to be able to edit or add records, we can select only one table for our recordset class.

Next, while creating the CBookListView class, we'll use the dialog box form illustrated in Figure 17.19.

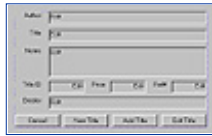


FIGURE 17.19 The *BookList* data form

Notice in the dialog box that all of the edit fields are grayed (indicating that these are read-only) and that there are four buttons shown across the bottom: Cancel, New Title, Add Title, and Edit Title. In operation, however, you'll see only two of these buttons at any given time, and during Add Title and Edit Title operations, most (but not all) of the edit fields will become enabled. In effect, a single form is being pressed into service to become several different forms to suit different functional requirements.

The operations that will be performed here are:

Viewing Records Allows the user to step through the [Titles] table (the default operation). All fields are read-only.

Editing Records The user is able to edit a record permitting changes to the [Author], [Title], [Notes], [Price], and [Reference] fields but not to the [Title ID] or [Dealer] fields.

Adding Records The user is able to enter a new record. Entry fields include the [Author], [Title], [Notes], [Price], and [Reference] fields while the [Title ID] is supplied as a uniquely generated ID. Last, the [Dealer] field is selected from a pull-down list box.

During Viewing and Editing operations, all of the record information comes from the [Titles] table (via the CBookListSet recordset). When a new record is being added, however, a second recordset class—CDealerListSet—is used to read the Dealers table and build a list of dealer entries.

Initially, the CBookListApp, CBookListDoc, CBookListSet, CBookListView, and CMainFrame classes are generated by AppWizard when the application is created. The only programming required is to create the dialog form (Figure 17.19) and, using the ClassWizard, to assign data members to the dialog controls.

The CDealerListSet can also be generated by the ClassWizard (as discussed previously), but, once created, some modifications are required to the source code (DealerListSet.cpp and DealerListSet.h). But, while we're at it, the CBookListSet source file (BookListSet.cpp) also needs a small change.

At least one of these changes are required because of the nature of the RFX_() functions.

More on RFX_() Function Calls

Earlier, the *Publisher* demo program worked with very simple data—brief strings, numerical data, and in one

case, a Boolean field. As suggested previously, however, data records are not always quite so conveniently simple, and because more complex data may be used, there are pitfalls in relying blindly on the DoFieldExchange functions created by ClassWizard.

For example, in BookListSet.cpp, the DoFieldExchange member appears (in the default version) as:

```
void CBookListSet::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CBookListSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[Title_ID]"), m_Title_ID);
    RFX_Text(pFX, _T("[Title]"), m_Title);
    RFX_Text(pFX, _T("[Author]"), m_Author);
    RFX_Text(pFX, _T("[Notes]"), m_Notes );
    RFX_Text(pFX, _T("[Price]"), m_Price);
    RFX_Text(pFX, _T("[Reference]"), m_Reference);
    RFX_Text(pFX, _T("[Dealer]"), m_Dealer);
    //}}}AFX_FIELD_MAP
}
```

Here, the first record—[Title_ID]—is a long value, while the remaining fields are all text ... or are they?

Well, as far as ClassWizard is concerned, they're text, but if we look at the actual database (see Figure 17.20), the [Notes] field is not a simple text record.



FIGURE 17.20 The Book_db [Titles] table (design view)

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

 [GO!](#)

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

 [GO!](#)

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

 [GO!](#)

[Previous](#) [Table of Contents](#) [Next](#)

As you can see in the design view, the [Notes] field is identified as a Memo type, not as Text. In Microsoft Access, a Text field has a size limit of 255 characters, and in Visual C++, the RFX_Text function has a corresponding default setting for a maximum length of 255 characters. However, the RFX_Text function is defined as:

```
void RFX_Text( CFieldExchange* pFX,           // required
               const char* szName,           // required
               CString& value,               // required
               int nMaxLength = 255,         // optional
               int nColumnType = SQL_VARCHAR, // optional
               short nScale = 0 );           // optional
```

Since the last three arguments have default values defined, it is very easy to be unaware of the default length, particularly since ClassWizard does not supply an argument or value beyond the required first three. Also, more importantly, ClassWizard does not distinguish between Text and Memo fields in any fashion.

Still, as long as none of the records accessed have a [Notes] field that is longer than 255 characters, everything is fine. But depending on this is a little like depending on winning the California Lottery—that is, you could get lucky...but do you really want to count on it?

NOTE: The results of a mismatch with a too long entry are very simple; at run time, you will be tersely informed "Data Truncated," but without any details and without any chance of recovery.

Now, while the default size parameter is 255, the valid range is much wider, with a minimum size of 1 and a maximum size of INT_MAX (2,147,483,647 = 2GB). Since 2GB seems a little extreme, we'll use a 2KB limit and modify the CBookListSet::DoFieldExchange function:

```
void CBookListSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CBookListSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Long(pFX, _T("[Title_ID]"), m_Title_ID);
    RFX_Text(pFX, _T("[Title]"), m_Title);
    //}}
```

```

RFX_Text(pFX, _T("[Author]"), m_Author);
RFX_Text(pFX, _T("[Notes]"), m_Notes, 2048 );
RFX_Text(pFX, _T("[Price]"), m_Price);
RFX_Text(pFX, _T("[Reference]"), m_Reference);
RFX_Text(pFX, _T("[Dealer]"), m_Dealer);
//}}AFX_FIELD_MAP
}

```

In like fashion, if you examine the [Dealers] table, you should find that both the [Address] and [Terms] fields are Memo types and, more importantly, several of the [Terms] entries are longer than 255 characters.

However, rather than modifying the DoFieldExchange members for the CDealerListSet class, we have a simpler solution in this instance. Since the only reason the *BookList* application has for accessing the [Dealers] table is to retrieve a list of Dealer names (to match the [Dealer] field in the [Titles] table), we can simply modify the variable declarations in DealerListSet.h:

```

// Field/Param Data
//{{AFX_FIELD(CDealerListSet, CRecordset)
CString    m_Dealer;
// CString    m_Address;    // while supplied by
// CString    m_EMail;      // ClassWizard, these
// CString    m_Phone;      // fields will not
// CString    m_Fax;        // be required
// CString    m_Terms;      //
//}}AFX_FIELD

```

We've left only one field parameter defined—m_Dealer—and we can make corresponding changes in DealerListSet.cpp:

```

CDealerListSet::CDealerListSet(CDatabase* pdb)
: CRecordset(pdb)
{
    //{{AFX_FIELD_INIT(CDealerListSet)
    m_Dealer = _T("");
    // m_Address = _T(""); // these fields were
    // m_EMail = _T("");   // declared by
    // m_Phone = _T("");   // ClassWizard but
    // m_Fax = _T("");     // are not needed
    // m_Terms = _T("");   //
    // m_nFields = 6;      // remove this entry and
    // m_nFields = 1;      // replace with a new field count
    //}}AFX_FIELD_INIT
    m_nDefaultType = snapshot;
}

```

Here, after commenting out the unneeded initializations, we also need to be careful to change the m_nFields member to reflect the correct field number.

Lastly, the CDealerListSet::DoFieldExchange function is also modified to match the other revisions:

```

void CDealerListSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CDealerListSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[Dealer]"), m_Dealer);
    // RFX_Text(pFX, _T("[Address]"), m_Address); // comment
    // RFX_Text(pFX, _T("[EMail]"), m_EMail);     // out these
    // RFX_Text(pFX, _T("[Phone]"), m_Phone);     // as well
    // RFX_Text(pFX, _T("[Fax]"), m_Fax);         //
    // RFX_Text(pFX, _T("[Terms]"), m_Terms );    //
    //}}AFX_FIELD_MAP
}

```

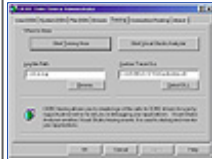
We could, of course, simply delete all of the unneeded entries instead of commenting them out. But for our present purposes, it was more to the point to show the changes from what ClassWizard supplied to what was actually needed.

NOTE: While ClassWizard relies mainly on the RFX_Text, RFX_Int, RFX_Long, and RFX_Bool transfer operations, a much wider choice of RFX_() operations are available. Refer to the Visual Studio online documentation for details.

Now that the immediate provisions have been handled, we'll return to the CDealerListSet and the AddEntry operations presently...after taking a look at how we're going to handle the simpler Edit operations.

Tracing SQL Operations

One utility that you may not be aware of is the trace facility provided by the ODBC Data Source Administrator:



You're offered two choices for tracing SQL calls: either as a log file, or using the Visual Studio Analyzer. Ideally, you would probably prefer to use the Analyzer; however, there is a small problem—the Analyzer requires SQL Server version 6.5, while the log file option is less restricted.

If you simply select the log file, a file is created reporting all SQL transactions (by any source), and afterwards, you can look through the log to find out what's happening.

The following fragment is a sample from a log created by the *BookList* example program (the entire log following a minimal transaction runs 31 pages).

```
BookList      fff47cfb:fff452eb  EXIT
               SQLPrepare  with return code 0 (SQL_SUCCESS)
HSTMT         0x00a91280
UCHAR *       0x02500600 [      48]
               "SELECT 'Dealer' FROM 'Dealers' ORDER BY 'Dealer'"
SDWORD        48

BookList      fff47cfb:fff452eb  EXIT
               SQLPrepare  with return code 0 (SQL_SUCCESS)
HSTMT         0x00a91280
UCHAR *       0x007836bc [      -3]
               "SELECT 'Dealer' FROM 'Dealers' ORDER BY 'Dealer'\ 0"
SDWORD        -3

BookList      fff47cfb:fff452eb  ENTER SQLExecute
HSTMT         0x00a91280

BookList      fff47cfb:fff452eb  ENTER SQLExecute
HSTMT         0x00a91280

BookList      fff47cfb:fff452eb  EXIT
               SQLExecute  with return code 0 (SQL_SUCCESS)
HSTMT         0x00a91280
```

Perhaps the resulting log is not the clearest source, but it does provide essential information that can be used to identify problems with SQL transactions.

Of course, you can also cut down on the size of the report by using breakpoints in your application to pause execution, turn on tracing, resume execution until the next breakpoint is reached (or until an error occurs), and then halt tracing. This leaves you with a more selective (or at least less verbose) report.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright](#) © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

A Sequence of Operations

In the *BookList* program, we've established three types of operations: scrolling through the titles, editing a title, and adding a new title to the records. (We could add a fourth—deleting an entry—but this is left as an exercise for the reader.)

The first operation, scrolling through entries, is our default functionality and has been pretty well supplied for us through AppWizard when we created the basic application. The only changes to the default provisions have been to create the form and to connect record fields to the edit fields.

Of course, while the initial form was shown earlier in Figure 17.19, there are two details that do not appear in the image. (Or, more accurately, there are two details that do appear in the image but that will not appear in the application—not immediately, anyway.)

Specifically, the Cancel and Add Title buttons shown in Figure 17.19 are defined as not visible by default, and the application as it initially appears is shown in Figure 17.21.

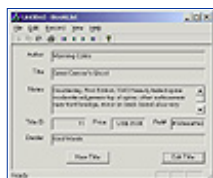


FIGURE 17.21 Using *BookList* to scroll through the database

TIP: In Figure 17.21, because the Notes edit field is read-only, a permanent vertical scrollbar (versus an auto-scrollbar setting) is used to ensure that long notes can be scrolled conveniently.

In this initial view, we can conveniently scroll through the records using the scroll buttons on the toolbar (or using the Record menu).

Adding Edit Operations

In order to move from scrolling through the records to performing an Edit operation, we need to make a few

changes to our record form. First, to control operations, two Boolean variables are declared in the BookListView.h header—m_bUpdateMode and m_bAppendMode—and of course both are initialized as FALSE.

Modifying the Display Form

When the Edit Title button is selected, the OnEditTitle function is called and begins with a test to determine if the database is opened for updating:

```
void CBookListView::OnEditTitle()
{
    if( m_pSet->CanUpdate() )
    {
        try
        {
```

Assuming that the database can be updated, we enter a try/catch block to handle exceptions and, since m_bUpdateMode is initially FALSE, we prepare the view form for editing:

```
        if( ! m_bUpdateMode )
        {
            // prepare for editing
            m_EditBtn.SetWindowText( "Update Title" );
            // change button text
            m_CancelBtn.ShowWindow( SW_SHOW );
            // show Cancel button
            m_NewTitleBtn.ShowWindow( SW_HIDE );
            // hide New Title button
            m_pSet->Edit();          // initiate update
        }
    }
```

To prepare for editing, we begin by changing the Edit Title button caption to “Update Title”, making the Cancel button visible, and hiding the New Title button. Then, last but not most important, we call the CRecordset::Edit function to initiate the editing process.

With this done, we call the ResetControls function (discussed in the following section, “Resetting Edit Fields”) and then flip the m_bUpdateMode flag to prepare to close the Edit operation.

```
        ResetControls();
        m_bUpdateMode = ! m_bUpdateMode;  // flip flag
    }
    catch( CException* pE )
    {
        pE->ReportError();
    }
}
else
    AfxMessageBox( "Recordset does not permit updates" );
}
```

Thus, once an Edit operation has been initiated, the form appears as shown in Figure 17.22.

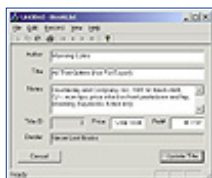


FIGURE 17.22 Editing an existing record

Here all fields, except for the Title ID and Dealer entries, are ready for editing, revision, and so on. The Title ID field cannot be changed because it is a unique record identifier.

The Dealer entry has also been left read-only, since it is linked to a second [Dealers] table where the entry must correspond. In the Add Title operation, however, we’ll show how an entry from the second table can be

selected and written to the [Titles] table. In any case, at the present, the record is ready for editing.

The second time the Edit Title button is selected (remember, it is now labeled “Update Title”), the call will be made to complete the Edit operation and the alternative (else) block will be executed.

```
else
{
    // finished editing
    m_EditBtn.SetWindowText( "Edit Title" );
    // change button text
    m_CancelBtn.ShowWindow( SW_HIDE );
    // hide Cancel button
    m_NewTitleBtn.ShowWindow( SW_SHOW );
    // show New Title button
    m_pSet->Update();           // complete update
    m_pSet->Requery();
}
```

This time, to close things down, the button title is reset to the original text, the Cancel button is hidden, and the New Title button becomes visible again before calling `m_pSet->Update` to write the changed information to the database. Finally, after finishing the update, the `Requery` function is called to refresh access to the database.

Resetting Edit Fields

Since we’ve initially set the edit box controls as “Read only,” the `ResetControls` function is used to change the edit box status according to the values of the `m_bUpdateMode` and `m_bAppendMode` flags.

```
void CBookListView::ResetControls()
{
    m_editTitle.SetReadOnly( m_bUpdateMode ||
                             m_bAppendMode );
    m_editReference.SetReadOnly( m_bUpdateMode ||
                                 m_bAppendMode );
    m_editPrice.SetReadOnly( m_bUpdateMode ||
                             m_bAppendMode );
    m_editNotes.SetReadOnly( m_bUpdateMode ||
                             m_bAppendMode );
    m_editAuthor.SetReadOnly( m_bUpdateMode ||
                              m_bAppendMode );
}
```

Here, if either flag is TRUE, the OR’d combination is also TRUE and the `SetReadOnly` function changes the corresponding control to read/write status for editing and entry. Or, if both are FALSE, the control is set to read-only.

Resetting Controls

Along with changing the edit fields from read-only to read/write, we also need (during Edit or Add operations) to disable the scroll buttons on the toolbar and the corresponding options on the Record menu.

This is not simply a matter of esthetics, but is a purely practical requirement, because if scrolling is enabled during an Edit or Add operation and if a scrolling operation is selected, the result invalidates the operation being attempted by moving the current record pointer.

Of course, we could add member functions to intercept and take control of the scroll operations—and this is possible—but there’s also an easier method. We begin by using `ClassWizard` to create `UPDATE_COMMAND_UI` functions for the `ID_RECORD_FIRST`, `ID_RECORD_LAST`, `ID_RECORD_NEXT`, and `ID_RECORD_PREV` Object IDs and then filling in Enable functions for each.

```
void CBookListView::OnUpdateRecordFirst(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( ! m_bUpdateMode && ! m_bAppendMode &&
                   ! IsOnFirstRecord() );
}
```

```

}
void CBookListView::OnUpdateRecordPrev(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( ! m_bUpdateMode && ! m_bAppendMode &&
                    ! IsOnFirstRecord() );
}

```

Here, three conditions are used. The first two—the `m_bUpdateMode` and `m_bAppendMode` flags (with a NOT condition applied)—apply to all cases, while a third test —`! IsOnFirstRecord`—is applied only to the two forward scroll buttons.

For the last pair of scroll buttons, the reverse scroll buttons, the two mode flags are combined with `! IsOnLastRecord`:

```

void CBookListView::OnUpdateRecordLast(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( ! m_bUpdateMode && ! m_bAppendMode &&
                    ! IsOnLastRecord() );
}

void CBookListView::OnUpdateRecordNext(CCmdUI* pCmdUI)
{
    pCmdUI->Enable( ! m_bUpdateMode && ! m_bAppendMode &&
                    ! IsOnLastRecord() );
}

```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The reasons for the `IsOnFirstRecord` and `IsOnLastRecord` tests are simple. If the current position is already at the first or the last record, there isn't any point in scrolling further ... and the controls should, ideally, offer a visual clue to the position even when the controls are not disabled for other reasons. Further, since these are `OnUpdate_` functions, the handling is automatic and they require no further attention.

In the Edit operation, most of the functionality has been provided by default through the `CRecordset` and `CRecordView` classes. The only real customization here has been to arrange the appearance of the form and controls and, during editing, to disable the scroll buttons.

To add a new record, the process is almost as simple, although a bit more customization and special coding is needed.

Adding a New Record

When we add a new record to the database, we will generally have a few requirements that were not present while editing an existing record. In this example, there are two additional requirements: to generate a unique record ID (for the `[Title_ID]` field in the `[Titles]` table) for each entry, and to derive a Dealer entry from the `[Dealers]` table.

Of course, we still need to set up the form for the entries, change a few buttons, and disable the scroll buttons, but first let's look at the question of generating a unique ID.

Generating a Unique Record ID

The simplest method of generating a unique record ID is to create a subroutine that reads through the existing data records and then returns an index value that is one more than the highest record ID found. In theory, this is a wasteful practice since, as records are deleted, older IDs could be reused. But we're less concerned with wasting numbers than we are with simply getting a usable number as quickly as possible.

The `NewTitleID` function is written to return a long value and begins by initializing a local variable (as zero) and then checking to ensure that the database does contain records.

```
long CBookListView::NewTitleID()  
{
```

```

long    newTitleID = 0;

if( ! ( m_pSet->IsBOF() && m_pSet->IsEOF() ) )
{

```

If the current record pointer is either before the first record (IsBOF is TRUE) or after the final record (IsEOF is TRUE), then we can be pretty sure that the database is empty. If so, then we can simply start numbering our records with 1, which will be created when the newTitleID variable is incremented before returning.

On the other hand, assuming that we do have records, we simply move to the first record and then begin stepping through the table until we reach the end. On each step, the value in newTitleID is compared to the current record's m_Title_ID value, with newTitleID taking the new value if a higher value is found.

```

        m_pSet->MoveFirst();
        while( ! m_pSet->IsEOF() )
        {
            if( newTitleID < m_pSet->m_Title_ID )
                newTitleID = m_pSet->m_Title_ID;
            m_pSet->MoveNext();
        }
        return ++newTitleID;
}

```

Finally, after running through the records, the resulting value is incremented by one before returning the result as a new and unique ID value.

So, now that we have a unique ID for the new record, what about the list of Dealers from the [Dealers] table?

Populating Data from a Second Table

When we create a new Title record—since the [Titles].[Dealer] field for each has to match an entry in the [Dealers].[Dealer] records—we don't want to simply depend on the user entering a name (and perhaps misspelling or otherwise producing an error). Instead, we are going to read the Dealer entries from the [Dealers] table, load these into a list (using a CComboBox control) and then let the user select an entry from these.

In actual fact, we've already loaded the list of Dealer names in advance, since this was done at the same time that we initialized the main record view, in the OnInitialUpdate function.

In OnInitialUpdate (in the CBookListView implementation) the process is pretty much the same as was demonstrated in the *Publisher* demo. The only real difference is the LoadDealerList function that is called at the end of the process.

```

void CBookListView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_bookListSet;
    m_pSet->m_strSort = "[Title]";
    CRecordView::OnInitialUpdate();
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
    LoadDealerList();
}

```

The LoadDealerList function has been called last for a simple reason. Because LoadDealerList is going to load information into a combo box, if the dialog (form) containing the combo box member hasn't been created, the operations to store the retrieved information will fail if this function is called too soon.

Direct SQL Statements

The LoadDealerList function offers a couple of points of interest including an opportunity to show how a direct SQL statement can be used with a CRecordset operation. In LoadDealerList, we begin by defining our SQL statement as a CString variable.

```
void CBookListView::LoadDealerList()
{
    CString    csSQL =
        "SELECT [Dealer] FROM [Dealers] ORDER BY [Dealer]";
```

As alternatives, the SQL statement could have been assembled in response to other program operations—for example, to include a WHERE statement to select a subset of the entries. But for present purposes, a simple direct statement is sufficient.

After creating the SQL statement, we retrieve a pointer to the `m_dealerListSet` member (as `m_pDealer`) from the document class before entering a try/catch exception-handling block.

```
m_pDealer = &GetDocument()->m_dealerListSet;
try
{
```

Within the try/catch block, we check to make sure that the `m_pDealer` recordset isn't opened yet (which it shouldn't be at this point) and then open the recordset using the defined SQL statement.

```
    if( ! m_pDealer->IsOpen() )
        m_pDealer->Open( CRecordset::snapshot, csSQL );
```

After opening the recordset, we perform the usual check to ensure that there are records in the table, move to the first record of the table, and then begin stepping through the table.

```
    if( ! ( m_pDealer->IsBOF() && m_pDealer->IsEOF() ) )
    {
        m_pDealer->MoveFirst();
        while( ! m_pDealer->IsEOF() )
        {
```

As we read each record (remember that the `CDealerListSet` has been rewritten to only retrieve the [Dealer] field), we simply copy the retrieved dealer name into the `m_cbDealerList` combo box and then move on to the next record.

```
            m_cbDealerList.AddString(
                m_pDealer->m_Dealer );
            m_pDealer->MoveNext();
        }
    }
}
catch( CException* pE )
{
    pE->ReportError();
}
}
```

Since the `m_cbDealerList` combo box can provide all of the sorting needed, we haven't bothered to specify an ORDER BY argument in the SQL statement. Instead, we've simply read all of the records, loaded them into the list box portion of the combo box and left it at that—a very simple and expedient process.

In another case, however, we might wish to wait to load our list until some other criteria had been supplied, perhaps criteria that would be used to select an appropriate subset of data.

And of course, we might actually want some form of information other than strings, but that would be no problem either. All that would be necessary would be to have some type of identifying string that could be loaded in the combo box. The real information could be stored as an associated value or as an external structure with an identifier associated with the combo box entry.

Not that these variations exhaust the possibilities; depending on circumstances and needs, we might even want to search other tables using information entered by the user to find appropriate links to be included in the new record. The details, however, aren't important since the same form of handling (including a generated

SQL statement passed in the CRecordset::Open function) could be used to retrieve virtually anything.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Customizing the Add Entry Form

The OnNewTitle function is similar to the OnEditTitle function, beginning by determining whether the database is opened for updating, entering a try/catch exception handling block, and finally, checking the m_bAppendMode flag to determine whether a new entry is being initiated or concluded.

```
void CBookListView::OnNewTitle()
{
    if( m_pSet->CanUpdate() )
    {
        try
        {
            if( ! m_bAppendMode )
```

To prepare to add a new title, we begin by re-captioning the New Title button, showing the Cancel button, and hiding the Edit button—much like the OnEditTitle procedure.

```
        {
            // prepare to add new title
            m_NewTitleBtn.SetWindowText( "Add Title" );
            m_CancelBtn.ShowWindow( SW_SHOW );
            m_EditBtn.ShowWindow( SW_HIDE );
```

We have two other controls to switch for this operation, however. Another item that is not readily visible in the dialog layout is the fact that we have two controls—an edit box and a combo box—that are the same size and that occupy the same window position.

```
            m_editDealer.ShowWindow( SW_HIDE );
            m_cbDealerList.ShowWindow( SW_SHOW );
```

Originally, the Dealer edit box is visible by default, but now the edit box is hidden and the combo box revealed. Remember, the combo box has already been loaded with a list of dealers.

And now we also call the NewTitleID function to get a unique record ID.

```
m_pSet->m_Title_ID = NewTitleID();
```

We have a few other details to attend to before proceeding. First, we need to reset several of the fields in `m_pSet` (but not, of course, the `m_Title_ID` field where we just entered the new value).

```
m_pSet->m_Author = _T("");
m_pSet->m_Notes = _T("");
m_pSet->m_Price = _T("");
m_pSet->m_Reference = _T("");
m_pSet->m_Title = _T("");
UpdateData( FALSE );
```

Also, after clearing the fields, we call `UpdateData` to refresh the display, before finally initiating the new entry.

```
m_pSet->AddNew();
}
```

At this point, we're ready for the user to enter new information and to select a dealer from the pull-down list, as shown in Figure 17.23.

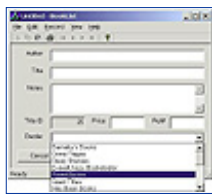


FIGURE 17.23 A new record waiting for information

After the user has created an entry (and clicked the Add Title button), there are again some special provisions to ensure that the data provided is complete and, equally important, to make sure that the proper information is added to the database.

```
else
{
    UpdateData( TRUE );
```

The first step is to call `UpdateData` to copy the information from the display into the member variables themselves. And, don't forget, these member variables are also recordset members.

But there's one piece of information that isn't in the recordset even after the `UpdateData` operation.

No, there hasn't been any oversight; it's just that one critical bit of information—the Dealer selection—needs to be copied from the combo box (by retrieving the current selection) into the recordset member. Also, after doing so, you must call `UpdateData` again, but this time copy the information from the recordset back to the edit boxes—that is, update the Dealer edit box that has not been corrected previously.

```
m_cbDealerList.GetLBText(
    m_cbDealerList.GetCurSel(),
    m_pSet->m_Dealer );
UpdateData( FALSE );
```

If this sounds like an elaborate and circuitous routine, what you need to remember is that the `m_pSet->Update` function, which will be called in a moment, will read the contents of the linked controls (the edit boxes) and write this information to the database. Ergo, the important consideration is to make sure that the displayed information is fully and correctly updated.

Once this is done, we also need to perform a test to ensure that all fields have been filled in correctly. In this case, the only test applied is to ensure that all entries have something in them.

```
if( m_pSet->m_Dealer.IsEmpty() ||
    m_pSet->m_Author.IsEmpty() ||
    m_pSet->m_Notes.IsEmpty() ||
```

```

        m_pSet->m_Price.IsEmpty()      ||
        m_pSet->m_Reference.IsEmpty()  ||
        m_pSet->m_Title.IsEmpty() )
    {
        ResetControls();
        return;          // cancel operation
    }

```

But if any fields are empty, we can also abort without concluding the Add Entry operation, leaving the user to correct the deficiency. On the other hand, if the entries are satisfactory, we again reset the display:

```

m_NewTitleBtn.SetWindowText( "New Title" );
m_NewTitleBtn.ShowWindow( SW_SHOW );
m_EditBtn.ShowWindow( SW_SHOW );
m_CancelBtn.ShowWindow( SW_HIDE );
m_editDealer.ShowWindow( SW_SHOW );
m_cbDealerList.ShowWindow( SW_HIDE );

```

Then, once the display is reset, we call the Update function to write the information to the database and, finally, call MoveLast to position the current record on the entry just created.

```

        m_pSet->Update();
        m_pSet->MoveLast();
    }

```

The remaining operations are the same as the OnEditEntry operation.

```

        ResetControls();
        m_bAppendMode = ! m_bAppendMode;
    }
    catch( CException* pE )
    {
        pE->ReportError();
    }
    else
        AfxMessageBox( "Recordset does not permit updates" );
}

```

As you should be able to see, neither the Edit nor Add operations are particularly difficult. Each requires some degree of care—and, depending on needs, you might want to exercise additional validation before accepting changes: but the majority of the operations are handled by the CRecordset and CRecordView classes.

We do, however, have one remaining operation to discuss—what happens when we need to cancel an edit or add operation.

Canceling Edit or Add Record Operations

The OnCancel function is called when either an Edit or an Add Title operation is aborted. The response is essentially the same in either case: Reset everything as if nothing has been changed.

The first step is to call the CancelUpdate function, which is used to terminate either a CRecordset::Edit or a CRecordset::Add operation:

```

void CBookListView::OnCancel()
{
    m_pSet->CancelUpdate();
    m_pSet->MoveFirst();           // reposition everything
    UpdateData(FALSE);           // and redisplay data
}

```

Second, MoveFirst is called to reposition the current record and then UpdateData refreshes the display to correspond. Alternately, if a database supports bookmarks, a bookmark could be assigned before an Add or Edit operation and then used to reset the position if an operation was cancelled.

Last, after refreshing the display, the next block of provisions simply ensures that the original display configuration is recreated—without worrying about which elements may have been changed in the interim.

```
m_EditBtn.SetWindowText( "Edit Title" );
m_EditBtn.ShowWindow( SW_SHOW );
m_NewTitleBtn.SetWindowText( "New Title" );
m_NewTitleBtn.ShowWindow( SW_SHOW );
m_CancelBtn.ShowWindow( SW_HIDE );
m_editDealer.ShowWindow( SW_SHOW );
m_cbDealerList.ShowWindow( SW_HIDE );
```

Finally, both operation flags are reset to FALSE and the ResetControls function is called.

```
m_bAppendMode = FALSE;
m_bUpdateMode = FALSE;
ResetControls();
}
```

And that's about it! The process of canceling an operation is simple, involving a minimum of cleanup and a minimum of effort.

Summary

We started this chapter by discussing how to create database access applications using the CRecordset class while using the CRecordView class to show the results. We then extended operations to edit existing entries and to add (update) new entries, again using the CRecordView class to provide an interactive display and entry form.

You should keep in mind, of course, that applications do not necessarily need to display database records (or use the CRecordView classes) in order to access and manipulate database information.

Also, Visual C++ offers several examples of ODBC/SQL applications, including the Enroll example, and you are invited to take a look at these as well as to play with the code for the *Publisher* and *BookList* demos.

Next, in Chapter 18, "ADO Database Access," we'll look at another method of database access using data access objects.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 18

ADO Database Access

- ActiveX Data Objects
- OLE DB
- The ADO Object Model
- Data operations using ADO

In the two preceding chapters, we've looked at database access using ODBC. Now we're going to go a step further and look at ActiveX Data Objects (ADO) and ADO's underlying Object Linking and Embedding Database (OLE DB) technology, because both ADO and OLE DB form the focus of major commitments by Microsoft, providing the keystone of their Universal Data Access (UDA) strategy.

NOTE:

Due to the scope and extent of ADO functionality and operations, this chapter can provide only an introduction to the topic and some simpler examples of database access using ADO and OLE DB. For a more comprehensive treatment of this lengthy topic, there are a number of volumes available that are devoted solely to programming using ADO and to documenting the complete feature set.

OLE DB

OLE DB is designed to be the logical successor to the ODBC mechanisms used in the previous chapters. However, considering that ODBC—a widely accepted and reasonably mature technology—works, why replace it?

The reasons are found in contemporary trends, in the form of information. While previous strategies have focused on relational databases, since these formed the vast majority of the information sources used, the current trends in information have branched from dependence on relational databases alone to encompass two non-database formats: Internet Web pages and electronic mail.

At the same time, other non-relational data sources are appearing, including documents and spreadsheets in a variety of formats but also encompassing image and audio data. Adaptations to these emerging formats require mechanisms beyond those provided by ODBC.

Further, since information today is often accessed through the Internet rather than through a LAN or WAN system, the old standards of assuming that the client and server sides were in constant communication must be abandoned. The nature of the Internet simply does not support a steady-connection paradigm, and modern data access mechanisms are required to recognize and accommodate the existence of transient accessibility.

Also—and far from least—ADO is central to Microsoft’s interests in creating a COM world where all objects and all access to information is handled through Microsoft’s Component Object Model (COM).

While none of these considerations say that ODBC is obsolete—and new versions and enhancements to ODBC are continuing—OLE DB is designed to solve many of these problems and is intended to supplant and replace the earlier standard.

ADO

ADO is intended to be a high-level programming model providing access to and handling for the COM-based ODBC data access interfaces. As an ActiveX component, ADO can be used from any language supporting COM—including Visual Basic, VBA, Visual C++, Visual J++, and scripting languages.

Advantages in using ADO include speed and ease of development provided by an object model allowing data to be retrieved with as little as a single line of code, while portability is provided in the fact that lines of code can be switched between languages with minimal revisions.

Support for a Transient Environment

In Chapter 17, “ODBC and SQL Database Access,” in examples using ODBC, transactions expected to have a cursor indicating a location within a database. That is, a transaction to retrieve data would expect the transaction to end with a reusable cursor pointing to the recordset just accessed, while subsequent transactions might assume a cursor as a basis for their operations.

In contrast, ADO/OLE DB deals with disassociated recordsets. A recordset can be retrieved and then disassociated with the server while still remaining a valid recordset for the client. Later, the recordset can be re-associated with the server database (or another source). In the interim, recordsets can be saved locally (on the client side), modified by the client, and only later updated as part of the central records.

The advantage in client-side data manipulation is that records can be retrieved, sorted, manipulated, and altered without requiring multiple trips to the server. Also, while this disconnected data access is primarily suited to the Internet, the practice can be used equally well for LAN applications, thus reducing the level of network traffic.

Data Providers versus Data Consumers

Although we traditionally speak of client and server systems, OLE DB introduces a new distinction with the terms *data provider* and *data consumer*.

A data consumer is anything that uses or consumes data. Thus, since ADO uses data provided by OLE DB, ADO is a data consumer.

Likewise, a data provider is simply anything providing data. However, a data provider is not necessarily the source of the data, but instead is the OLE DB mechanism that supplies data from the data source.

Data providers may interact directly with data sources, may work through other mechanisms (such as ODBC) to retrieve data, or may use either (or both) mechanisms while introducing added value during the transaction. They work by combining data from different sources or manipulating the data in one fashion or another (see, for example, the MSDataShape provider).

Data Sources

Initially, with ADO 2.0, OLE DB interfaces are provided for a variety of data sources, including:

Data Shape Provides access to hierarchical recordsets and allows the creation of master/detail type recordsets, to permit drilling down into detailed data.

Directory Services Provides access to resource data stores such as the Windows 2000 Active Directory services.

Index Server Provides access to the Microsoft Index Server used to provide indexed data for Web sites.

Jet 3.51 Provides access to Microsoft Access databases (illustrated in this chapter), allowing use of all standard Access databases, including linked tables.

ODBC Drivers Provides access to data sources that do not support OLE DB interfaces.

Oracle Provides access to Oracle databases.

Persisted Recordset Provides access to locally saved recordsets. Technically, Persisted Recordset interface is not a data provider, but it functions in the same fashion for recordsets that have been saved locally.

Site Server Search Provides access to the Microsoft Site Server used to maintain large or complex Web sites and offer indexed data for sites.

SQL Server Provides access to Microsoft SQL Server databases.

NOTE:

A newer version of ADO—version 2.5—is now in beta and may be shipped with Windows 2000.

The preceding are only the standard providers supplied by Microsoft. At the same time, other vendors are engaged in creating OLE DB interfaces for their own data stores or providing extended OLE DB interfaces that provide functionality beyond those offered by Microsoft.

In addition to access to data sources, OLE DB also offers services such as a query processor and a cursor engine for use by client-side applications. By making these services available to the client, the server-side application is freed from the responsibility of providing such services. Equally important, by making cursor handling a local responsibility, disconnected recordsets are supported while Remote Data Services can also rely on locally maintained data cursors.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The ADO Object Model

ADO is constructed around an object model defining the objects comprising the model and their hierarchy, as shown in Figure 18.1.



FIGURE 18.1 The ADO object model

Because ADO creates objects as required without demanding the user (the programmer) to be aware of the details, the structure and particulars of the object model are relatively unimportant. Still, it's worth being aware of the existence of the objects.

While ADO relies principally on three main objects—Connection, Command, and Recordset—each of the several ADO objects are discussed briefly in the following sections.

The Connection Object

The Connection object forms the basis for ADO, providing the methods used to connect to a data store. While the Connection object appears at the top of the hierarchy, there are no actual requirements to create a Connection object prior to creating a Recordset or a Command. Instead, a Recordset object's Open method can be called with a data source name such that the Connection object is created indirectly.

By preference, however, and also for clarity, we will begin by creating reusable Connection objects and then creating Recordsets using the established connections.

NOTE:

The Connection object also serves to contain any errors that occur during ADO operations.

The Command Object

The Command object is designed principally to execute single commands, particularly commands requiring parameters. A Command object is useful for executing stored procedures or queries, and provides a means for improving speed of operations as well as segmenting an application, but it may also be used to return recordsets.

Further, a Command object—depending on the data provider—can be used to store command details for execution at a later time and can, optionally, be executed several times either on the same data source or on different data sources by changing the Connection object.

The Recordset Object

The Recordset object is the ADO object used in most operations. As a consequence, the Recordset object has more properties and methods defined than the other objects, including methods for retrieving, updating, adding, and deleting recordsets.

The principal use of a Recordset object is to examine or manipulate data from the data source, and methods are provided to allow movement through records, to locate records or subsets of records, to sort records, and, in a multi-tier system, to work with disconnected recordsets.

The Fields Collection

The Fields collection identifies each field in a recordset and may be used in two ways.

The first form uses a Fields collection with an existing recordset to allow individual fields to be examined by type, name, and other details. In this respect, Fields are useful when you are unsure precisely what a recordset contains or if you wish to show tables dynamically. A Fields collection could be used, for example, to build an HTML table by supplying the table header and then using recordsets to fill in the table details.

A second form uses a Fields collection to create a recordset—using the Fields.Append function to build a new recordset or to create a single recordset based on other retrieved recordsets.

The Errors Collection

The Errors collection contains all errors generated by a data provider or any warnings issued in response to failures of operations. An Errors collection is used because a failure can generate multiple errors. The Errors collection could be used to build a routine for centralized error handling.

The Parameters Collection

The Parameters collection belongs to the Command object and is used to pass parameters to stored procedures and queries. Parameters may contain several items of information, including:

Direction Specifies whether the parameter is used for input, output, or both, or as a return value.

Name The name identifying the parameter in the stored procedure or query.

Size The size of the data stored.

Type The type of data stored in the parameter.

Value The actual parameter value.

Parameters can be used in two fashions. A data source can be queried to fetch parameter information from the data source, filling in the Parameters collection automatically. Alternatively, Parameter collections can be constructed manually.

The Properties Collection

The Properties collection contains provider-specific information about an object. For a Connection, for example, the Properties object reports on the facilities supported by the object, such as the maximum number of columns that can be returned in a SELECT statement, or what type of join capabilities are supported.

The principal use of the Properties collection—aside from possibly querying an object during development—would be in developing an application to support multiple data providers, where the application might query the provider to find out which methods or members are supported.

Shortcomings in ADO

While ADO is the newest data handler in Microsoft's arsenal and is supplied as a part of Visual Studio 6.0 (but is also available by download from www.microsoft.com/data), the documentation supporting ADO appears to be somewhat lacking. That is, while the online documentation supplied with Visual Studio 6.0 does include extensive material on ADO, it does not provide full documentation for the functions (particularly those used in Visual C++). As a second source, the Windows 2000 Platform SDK also includes some coverage for ADO.

While it seems reasonable to assume (and expect) that this shortcoming will be addressed in future releases of the Visual Studio package, you should be aware that the current implementation is not complete in this respect. In order to program using ADO, you will probably find a severe need for some third-party source of documentation.

Programming with ADO

Before a program can be written using ADO and OLE DB, a first requirement is to ensure that the necessary Visual C++ database components are installed—specifically, the ADO components and the OLE DB data providers. If you have not installed these—or if you are unsure and need to check—run the Visual Studio 6 setup program and then, in the Visual Studio 6.0 Enterprise Setup dialog box, click the Add/Remove button. This will bring you to the Maintenance dialog box shown in Figure 18.2.

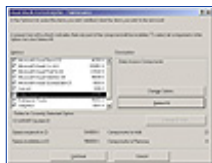


FIGURE 18.2 The Visual Studio 6.0 Enterprise Maintenance dialog box

In the Maintenance dialog box, select the Data Access option from the list box. If the checkbox next to the Data Access option is both clear and checked, then all of the Data Access components are already installed. If there is no check mark, no components are installed, or, if the box is gray with a check mark, some components are installed but some are not.

After highlighting the Data Access option, click the Change Option button to display the Data Access dialog box shown in Figure 18.3.

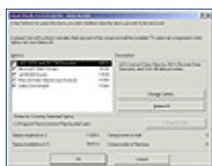


FIGURE 18.3 The Data Access dialog box

From the Options list, the ADO, RDS, and OLE DB Providers option should be selected. Also, if you have been working with any of the examples used in Chapter 17, it seems reasonable to assume that the Microsoft ODBC Drivers are already installed.

TIP:

While you are at it, you might as well—unless you have space limitations to consider—install the Jet IISAM Drivers, the Remote Data Objects and Controls, and the Data Environment components.

After selecting options, simply click the OK buttons in each dialog box to complete the selection process and return to the original dialog box.

At this point, if you have no components selected to install, the dialog box will show 0 components selected and you can then choose Cancel to exit the setup program. Otherwise, click Continue to complete installation.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Creating an ADO Database Program

Since we will be using the same database chosen for the illustration program in Chapter 17, presumably the Books_db.mdb database has already been copied to your local (or network) hard drive. In the following example, however, instead of using a registered ODBC driver, you will need to know the drive/path information to modify the source code accordingly.

TIP:

In other circumstances, it would be appropriate to use a registry entry to specify the database file location, or perhaps to use the file/directory common dialog to locate the database file. Using a hard-coded drive/file specification is appropriate only for the purpose of demonstration and should not be employed in actual applications.

For our demo program, we begin with AppWizard and create a simple, dialog-based application and, in the initial dialog box, we create a single list box with the “Use tabstops” option selected as shown in Figure 18.4. Simple static text entries are used to label the columns.

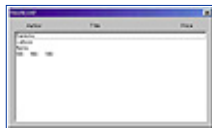


FIGURE 18.4 The opening dialog box for the *BookList2* demo program

The entries shown in the list box were supplied by the Dialog Editor during testing. Note that three tab entries—using default tab positions—are also shown. While tabs will be used, new tab positions will be assigned with more appropriate spacing.

Once AppWizard has created our basic application, a few immediate modifications are necessary and we begin with the BookList2.cpp and BookList2.h files.

First, in the BookList2.h header, we need to declare a member variable as shown in bold in the following listing:

```

class CBookList2App : public CWinApp
{
public:
    CBookList2App();
    CString m_csDataSource;
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CBookList2App)
public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL
~BL
// Implementation

//{{AFX_MSG(CBookList2App)
// NOTE - the ClassWizard will add and remove member
// functions here.
//      DO NOT EDIT what you see in these blocks of
//      generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

```

```

extern CBookList2App theApp;

```

```

////////////////////////////////////

```

We also want to make a second change (above) to declare theApp as externally accessible. Note that theApp is not defined in the header but does appear in the .CPP file. The extern declaration here simply makes it possible, from other application classes, to refer to members of CBookList2App as the App.m_membername—in effect, allowing members of CBookList2App to be treated as globally available member variables.

Continuing with the CBookList2.cpp source file, you'll find that theApp has already been declared as an instance of CBookList2App and appears thus:

```

////////////////////////////////////
// The one and only CBookList2App object

```

```

CBookList2App theApp;

```

Immediately following this declaration, in the InitInstance procedure, two additions (shown in bold) are needed:

```

////////////////////////////////////
// CBookList2App initialization

```

```

BOOL CBookList2App::InitInstance()
{
    AfxOleInit();
    m_csDataSource = "Provider=Microsoft.Jet.OLEDB.3.51;"
    "Data Source=G:\\Sybex\\Win2000\\Books_db.mdb;";

    AfxEnableControlContainer();
}

```

The AfxOleInit instruction is called to initialize the OLE DLLs, a necessity before we can use the OLE DB operations.

The second entry—for m_csDataSource—is used to set up a string that identifies the data provider—in this case, Microsoft's Jet OLE DB engine, version 3.51—and the data source. This second entry you will need to modify to match your system and to identify the location where the Books_db.mdb database can be found. (Be sure that you use double backslashes (\\) in the string—otherwise, you will get errors.)

The reason for placing the data source specification here in the app source file is so that it can be easily

accessed from more than one location in other application classes (from other source files). Since the data source is going to be used several times, maintaining multiple instances of the data source specification would be counterproductive and a single shared specification saves time, trouble, and errors.

We have one more preparatory change to make, this time in the StdAfx.h header. Again, the additions to the default provisions are shown in bold.

```
// stdafx.h : include file for standard system include
//      files, or project specific include files that are
//      used frequently, but are changed infrequently
//
...
#ifdef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>      // MFC support for Windows Common
                        // Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
```

#include <comdef.h>

The first provision is to include the ComDef.h header to provide access to a number of special COM support classes in Visual C++ that make it easier to work with the OLE Automation data types.

The second provision (following) is an import directive to import the ADO library class declarations.

```
#import "C:\Program Files\Common Files\System\ADO\µ
MSADO15.dll" \
no_namespace \
rename( "EOF", "adoEOF" )
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.
```

The first line (which has been split here due to page width limitations) instructs Visual C++ where the MSADO15 dynamic link library is found. If you are unsure of the location, then use the Find Files or Folders option from the Start menu to locate the file and change this entry accordingly. (Or, if the file cannot be found, repeat the installation procedures discussed previously.)

Since this directive is used only while compiling the application, this setting information does not need to be provided when the application is executing on other systems.

The `no_namespace` directive specifies that no namespace is to be used for the ADO objects. Some applications require namespace to avoid naming collisions between application objects and ADO objects, and a namespace can be specified (replacing the `no_namespace` directive) as:

```
rename_namespace( "AdoNS" ) /
```

The final provision is to rename the EOF (end of file) in ADO as `adoEOF`. This is to avoid a potential conflict with other libraries that define their own EOF. (See Microsoft Knowledge Base article Q169496 for further details.)

TIP:

When you compile your project, the Build directory (the Debug subdirectory) should include two files named `msado15.tli` and `msado15.tlh`. These headers are created from the compiler using the type library in `msado15.dll` and can be examined to see the ADO classes that can be called from your application.

With these preliminaries accomplished, the next item is to start adding application-specific functionality to the `BookList2Dlg` source files.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Opening a Connection

The first thing needed in an ADO application is to open a connection to the data source. In this example, the data source location has been specified in the BookList2.cpp file as the m_csDataSource member but we'll also need two members in the CBookList2Dlg class, which are defined in the header as:

```
class CBookList2Dlg : public CDialog
{
// Construction
public:
    CBookList2Dlg(CWnd* pParent = NULL);
    bool    m_IsConnectionOpen;
    _ConnectionPtr m_pConnection;
// Dialog Data
```

The Boolean member is simply used as a flag to record whether we have a connection opened. The _ConnectionPtr member is a pointer to the ADO Connection object that is used when a connection is created, and that can be shared and reused after initialization.

To initialize our members and open a connection, we begin in the OnInitDialog function (in the CBookList2Dlg class).

```
BOOL CBookList2Dlg::OnInitDialog()
{
    HRESULT    hRes;
    m_IsConnectionOpen = FALSE;
    CDialog::OnInitDialog();
    // Add "About..." menu item to system menu.
    ...
```

AppWizard has supplied the bulk of the initialization of the dialog so the bulk of our additions appear at the end of the function.

```
// TODO: Add extra initialization here
```

```
try  
{
```

We're using try/catch exception handling with ADO so failure events can be reported as Trace statements in the Debug window (in Visual Studio). During actual execution, of course, the Trace information will not be available.

Within the try/catch block, the first step is to call the CreateInstance function to create an instance of an ADO Connection object:

```
hRes = m_pConnection.CreateInstance(  
    __uuidof( Connection ) );  
if( SUCCEEDED( hRes ) )  
{
```

If successful, the next step is to call the Open function:

```
hRes = m_pConnection->Open(  
    _bstr_t( (LPCTSTR) theApp.m_csDataSource ),  
    _bstr_t(L""),  
    _bstr_t(L""),  
    adModeUnknown );
```

The first argument is the data provider and data source specification that has been defined in the CBookList2App class (in BookList2.cpp). Note the use of the theApp reference.

The second and third arguments create temporary instances of a _bstr_t object that are passed as parameters to the Open function. The L macro preceding the empty quotes indicates that the string is passed as a wide character string.

The fourth argument—adModeUnknown—simply indicates that the permissions for the connection have not been set or cannot be determined at the present time.

TIP:

The _bstr_t class is one of the COM support classes provided by Visual C++ and is defined in ComDef.h. The _bstr_t class encapsulates the BSTR data type used by ADO and COM to pass strings in function calls and makes it easier, in Visual C++, to pass BSTR arguments with a minimum of typecasting.

Finally, if the Open function has returned a success result, the m_IsConnectionOpen member is set as TRUE and the ADO data source connection is ready for use.

```
if( SUCCEEDED( hRes ) )  
    m_IsConnectionOpen = TRUE;  
}  
}
```

Alternately, if the try/catch block catches an error, the catch section is used to query _com_error and to report the error condition, error code, and other details.

```
catch( _com_error &ampe )  
{  
    // get info from _com_error  
    _bstr_t bstrSrc( e.Source() );  
    _bstr_t bstrDes( e.Description() );  
    TRACE( "Exception thrown for #import generated µ  
        class\n" );  
    TRACE( "\tCode = %081x\n", e.Error() );  
    TRACE( "\tMeaning = %s\n", e.ErrorMessage() );  
    TRACE( "\tSource = %s\n", (LPCTSTR) bstrSrc );  
    TRACE( "\tDescription = %s\n", (LPCTSTR) bstrDes );  
}
```

A second catch block is included to provide for non-COM errors, but offers only a laconic message such as:

```
catch(...)
{
    TRACE( "An unhandled exception has occurred" );
}
```

Since the same catch block is used in all of the ADO transactions in this example, you'll find this block of code repeated regularly. When an error does occur, the Debug window in Visual Studio will display something looking a lot like this:

```
First-chance exception in BookList2.exe (MSVCRTD.DLL):µ
0xE06D7363: Microsoft C++ Exception.
Exception thrown for #import generated class
Code = 0000000000000000000000000000000000000000000000000000000000000000µ
0000000000000000000000000000000000000000000000000000000000000000µ
Meaning = Unknown error 0x800A0xxx
Source = ADO.DB.Recordset
Description = ADO could not yada yada yada yada
```

These Trace messages are worth your attention, since this is essentially the only debugging mechanism usable within a `try/catch` block.

WARNING:

Attempting to trace execution within a `try/catch` block is fine—until an exception occurs. Then, since the debugging trace also causes exceptions, the usual result is to hang your system. In other words, if you attempt to step through the `try` block of an exception handling section, you’re likely to require a reboot to proceed.

Last, assuming that a connection has been opened successfully, the `FetchRecords` function is invoked to load the dialog list box.

```
FetchRecords();
return TRUE;
}
```

Cleaning Up

Along with opening a connection, we also need to include a provision to close the connection when the application exits. This is relatively brief—no exception handling is required here—and is provided in the `OnClose` function as:

```
void CBookList2Dlg::OnClose()
{
    if( m_IsConnectionOpen )
        m_pConnection->Close();
    m_IsConnectionOpen = FALSE;
    CDialog::OnClose();
}
```

Building an Item List

The `FetchRecords` function is used to retrieve a series of records from the data source and to load these entries into the dialog list box. The `FetchRecords` function begins with declarations for a series of variables and then performs two tasks to prepare the list box.

```
void CBookList2Dlg::FetchRecords()
{
    _RecordsetPtr pRecordset;
    _bstr_t        bstrQuery( "SELECT [Title_ID], [Author], p
```

```
[Title], [Price] FROM [Titles]" );
```

The bstrQuery variable provides a query string—using conventional SQL nomenclature—to retrieve the records desired.

Unlike the previous database examples where data was simply transferred (almost automatically) from the recordset to the dialog box, form, or other member variables, here we need a set of _variant_t variables to receive the data pending further disposal.

```
_variant_t  vRecsAffected, vAuthor, vTitle,  
            vPrice, vTitle_ID;  
long        lItemID;  
int         nIndex, nTabs[2] = { 60, 200 };  
CString     csFormat;  
  
m_lbResults.ResetContent();  
m_lbResults.SetTabStops( 2, nTabs );
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The tab positions are used so that we can display three items of information on each line: the author, title, and price in a columnar format. While there are other methods, the values for the tab positions were arrived at largely by trial and error.

Beginning the try/catch exception handling, the first task is to execute this query string:

```
try
{
    pRecordset = m_pConnection->Execute(
        bstrQuery,
        &pvRecsAffected,
        adOptionUnspecified );
}
```

Note that when we execute the query, we call using the Connection object that was opened in the OnInitDialog function, but we receive a pointer to a recordset—pRecordset—if the query executes correctly.

Next, if the initial result is not an end-of-file report—meaning that the data source does contain data—we begin a while loop, which terminates if an end-of-file condition is reached.

```
if( ! pRecordset->GetadoEOF() )
{
    while( ! pRecordset->GetadoEOF() )
    {
```

Within the while loop, we make individual GetCollect calls for each record field we want to retrieve from the recordset. The recordset contains more fields than we actually want to use but we aren't required to employ all of them and, for the present use, we'll restrict ourselves to four.

```
vAuthor = pRecordset->GetCollect(L"Author");
vTitle  = pRecordset->GetCollect(L"Title");
vPrice  = pRecordset->GetCollect(L"Price");
vTitle_ID =
```



```
pRecordset->GetCollect(L"Title_ID");
```

Once the four fields have been collected, we use the LPCTSTR and _bstr_t typecasts to make it possible to format three of the fields as a CString object, with the fourth field converted to a long variable.

```
csFormat.Format( "%s\t%s\t%s",
    (LPCTSTR)(_bstr_t) vAuthor,
    (LPCTSTR)(_bstr_t) vTitle,
    (LPCTSTR)(_bstr_t) vPrice );
nIndex = m_lbResults.AddString( csFormat );
lItemID = atol( (LPCTSTR)(_bstr_t) vTitle_ID );
m_lbResults.SetItemData( nIndex, lItemID );
```

As each CString is added to the list box, the item ID value is set as a data element associated with the list box entry. This will allow us later to retrieve the Title_ID value when a list box selection is made.

Last, we call the MoveNext function to step to the next recordset in the data source that matches our query statement.

```
        pRecordset->MoveNext();
    }
}
pRecordset->Close();
```

Once we're done with the recordset object, because we've finished with the data source for now, the recordset Close function is called.

Continuing, the catch block is the same as we discussed a moment earlier (and it is omitted here). Thus, all that remains—assuming, of course, that execution has not been interrupted by an exception—is to call the UpdateData function to update the list box.

```
        UpdateData( FALSE );
    }
}
```

At this point, the *BookList2* application is ready to open our data source and display a list of author/title/price information as shown in Figure 18.5.

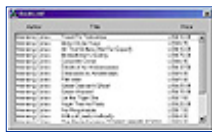


FIGURE 18.5 Displaying a list of entries

A list is a good method of displaying multiple records but, along with the list, we need to provide a means of selection so that a record can be examined in detail, manipulated further, or updated with new information. And this facility is provided very easily by creating an OnDblclk member for the list box.

Selecting a List Box Entry

The OnDblclkResultList function is very simple and is used to call another dialog after retrieving the Title_ID value associated with the list entry.

```
void CBookList2Dlg::OnDblclkResultList()
{
    long lTitle_ID =
        m_lbResults.GetItemData( m_lbResults.GetCurSel() );
    CDetailsDlg *pDetailsDlg;

    pDetailsDlg = new CDetailsDlg;
    pDetailsDlg->m_lTitleID = lTitle_ID;
    pDetailsDlg->DoModal();
}
```

As you can see, the value associated with the selected entry is used to initialize the CDetailsDlg's m_lTitleID member after the dialog is created but before the dialog is called and displayed. Once this is done, the rest of the operation is found in the CDetailsDlg class implementation.

Presenting the Details

If the dialog format used to present the details from a Books_db recordset looks suspiciously similar to the forms used in the *BookList* application in Chapter 17, the reason might be simply that they are identical. But even if they are identical in appearance, the operations used to populate the data fields are quite different.

In the CDetailsDlg.h header, we begin by declaring a series of Boolean variables that will be used later to determine which fields, if any, have been changed—but we also declare a _ConnectionPtr member just as we did in the CBookList2Dlg class.

```
class CDetailsDlg : public CDialog
{
// Construction
public:
    CDetailsDlg(CWnd* pParent = NULL);

    bool m_IsConnectionOpen, m_bUpdateMode, m_bAppendMode,
        m_bAuthorChange, m_bNotesChange, m_bPriceChange,
        m_bReferenceChange, m_bTitleChange;
    _ConnectionPtr m_pConnection;
```

The real work begins in the OnInitDialog function which, except for initializing a couple of member variables, is almost identical to the same operation in the CBookList2Dlg class.

```
BOOL CDetailsDlg::OnInitDialog()
{
    HRESULT    hRes;

    CDialog::OnInitDialog();
    m_IsConnectionOpen = FALSE;
    m_bUpdateMode = FALSE;
    m_bAppendMode = FALSE;
    try
    {
        hRes = m_pConnection.CreateInstance(
            __uuidof( Connection ) );
        if( SUCCEEDED( hRes ) )
        {
            hRes = m_pConnection->Open(
                _bstr_t( (LPCTSTR) theApp.m_csDataSource ),
                _bstr_t(L""),
                _bstr_t(L""),
                adModeUnknown );
            if( SUCCEEDED( hRes ) )
                m_IsConnectionOpen = TRUE;
        }
    }
}
```

Notice that the same data provider and data source specification—defined in theApp.m_csDataSource—is used, the same arguments, and the same result test. And, following this, the same catch block is used...again.

Following the catch block, however, instead of retrieving multiple records, we call a new function:

```
    FetchSingleTitle();
    return TRUE;
}
```

The FetchSingleTitle function really isn't that different overall from the FetchRecords function discussed a

moment earlier.

```
void CDetailsDlg::FetchSingleTitle()
{
    _RecordsetPtr pRecordset;
    _bstr_t      bstrQuery(
        "SELECT * FROM [Titles] WHERE [Title_ID] = " );
    char         buf[10];
    _variant_t   vRecsAffected, vAuthor, vTitle, vPrice,
                vDealer, vReference, vNotes;

    ltoa( m_lTitleID, buf, 10 );
    bstrQuery += buf;
}
```

The most important difference is in the query string. In the FetchRecords function, we used a predefined query string to retrieve all records, and here we are constructing a specific query string to retrieve a single record matching a selected Title_ID. These differences aside, both functions use perfectly conventional SQL statements.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#) [Table of Contents](#) [Next](#)

Likewise, the try/catch blocks are functionally similar, with this version retrieving all fields from the record set instead of only selected fields.

```
try
{
    pRecordset = m_pConnection->Execute(
        bstrQuery,
        &vRecsAffected,
        adOptionUnspecified );

    if( ! pRecordset->GetadoEOF() )
    {
        vAuthor = pRecordset->GetCollect(L"Author");
        vTitle = pRecordset->GetCollect(L"Title");
        vPrice = pRecordset->GetCollect(L"Price");
        vNotes = pRecordset->GetCollect(L"Notes");
        vDealer = pRecordset->GetCollect(L"Dealer");
        vReference = pRecordset->GetCollect(L"Reference");
        // copy results to dialog member variables
        m_csAuthor = (LPCTSTR)(_bstr_t) vAuthor;
        m_csDealer = (LPCTSTR)(_bstr_t) vDealer;
        m_csNotes = (LPCTSTR)(_bstr_t) vNotes;
        m_csPrice = (LPCTSTR)(_bstr_t) vPrice;
        m_csReference = (LPCTSTR)(_bstr_t) vReference;
        m_csTitle = (LPCTSTR)(_bstr_t) vTitle;
    }
    pRecordset->Close();
}
```

Notice again that we've used LPCTSTR and _bstr_t to typecast the _variant_t wide-character results returned to fit the CString members in the dialog.

We could, of course, rewrite these transactions in a briefer format as:

```

m_csAuthor = (LPCTSTR)(_bstr_t)
pRecordset->GetCollect(L"Author");

```

This format would save a few local variables, but the difference is minor.

Of course, since we need only one record and the selection is specified in the query statement, there's no need for a loop in this function. However, like the `FetchRecords` function, the `FetchSingleTitle` function concludes with exactly the same catch block as before.

```

catch( _com_error &ampe )
{
    ...
}
catch(...)
{
    TRACE( "An unhandled exception has occurred" );
}

```

Or, if everything is successful, we conclude by calling `UpdateData` to display the results in the dialog box.

```

UpdateData( FALSE );
}

```

The resulting display appears in Figure 18.6.

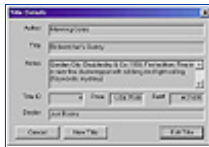


FIGURE 18.6 Displaying the Details

Similar to the previous example, buttons are provided for Cancel (return to the list), New Title (enter a new record), and Edit Title (update the selected record).

Again, these options are similar to those in the *BookList* application but the mechanisms for each are different.

Updating a Record

The process of preparing the dialog box for an edit operation is basically the same as we discussed in the *BookList* demo program. There are a few differences, but none that are exceptionally important.

```

void CDetailsDlg::OnEditTitle()
{
    if( ! m_bUpdateMode )
    {
        // prepare for editing
        m_EditBtn.SetWindowText( "Update Title" );
        m_NewTitleBtn.ShowWindow( SW_HIDE );
        m_bAuthorChange = FALSE;
        m_bNotesChange = FALSE;
        m_bPriceChange = FALSE;
        m_bReferenceChange = FALSE;
        m_bTitleChange = FALSE;
    }
}

```

We do set a series of Boolean flags while a series of `OnChange_` functions (one for each of the enabled edit fields) resets these flags if the contents of the field are changed. These flags are tested in the `UpdateEntry` function to decide which fields need to be updated in the data source.

After editing and after retrieving the contents of the fields, the fields are checked for empty strings (which we've decided are unacceptable) before resetting the controls and calling the `UpdateEntry` function.

```

else
{
    // finished editing
    UpdateData( TRUE );           // get data from fields
    if( m_csAuthor.IsEmpty() || m_csDealer.IsEmpty() ||
        m_csNotes.IsEmpty() || m_csPrice.IsEmpty() ||
        m_csReference.IsEmpty() || m_csTitle.IsEmpty() )
        return;                  // unacceptable entry?
    // clean up
    m_EditBtn.SetWindowText( "Edit Title" );
    m_NewTitleBtn.ShowWindow( SW_SHOW );
    UpdateEntry();
}
EnableEdits( TRUE );
m_bUpdateMode = ! m_bUpdateMode;           // flip flag
}

```

The UpdateEntry function begins by asking if any of the edit fields have been changed.

```

void CDetailsDlg::UpdateEntry()
{
    if( ! m_bAuthorChange && ! m_bNotesChange &&
        ! m_bPriceChange && ! m_bReferenceChange &&
        ! m_bTitleChange ) return;
}

```

If there are no changes, there's no point in continuing and we'll simply return. Otherwise, if anything has been changed, we need to open a new recordset, which means, of course, recreating the same query string used to display the original record.

```

_RecordsetPtr pRecordset;
_bstr_t      bstrQuery(
    "SELECT * FROM [Titles] WHERE [Title_ID] = " );
_variant_t vNull, vRecsAffected;
char        buf[10];
bool        bAdd = FALSE;
HRESULT      hRes;
CString      csEntry;

vNull.vt = VT_ERROR;
vNull.scode = DISP_E_PARAMNOTFOUND;

ltoa( m_lTitleID, buf, 10 );
bstrQuery += buf;           // and the query string

```

Remember that even though we've kept our connection open to the data source, we have not maintained a cursor to the selected record. So before we can update the record, we need to create a new recordset, reestablish the connection, and reload the recordset...

```

try
{
    hRes = pRecordset.CreateInstance(
        _uuidof(Recordset) );
    if( SUCCEEDED( hRes ) )
    {

```

TIP:

The SUCCEEDED macro is used instead of a simple Boolean evaluation because a successful operation returns 0—which is commonly taken as FALSE—while a failure returns a non-zero error code.

Simply creating an instance of the recordset is only half the preparation, because we need to connect the recordset to the open (active) connection—our link to the data source—and then call the Open member of the

recordset using our query string to connect to a specific record.

```
pRecordset->PutRefActiveConnection(  
                                m_pConnection );  
hRes = pRecordset->Open(  
    _variant_t( bstrQuery ),  
    vNull,  
    adOpenForwardOnly,  
    adLockOptimistic,  
    adCmdText );  
if( SUCCEEDED( hRes ) )  
{
```

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Note that, this time, the recordset is opened using the `adOpenForwardOnly` mode to give us a forward-only type cursor and using record locking (`adLockOptimistic`) so that no one else can alter the record while we're updating it.

Then, once we've reconnected to the data source and to a specific record, before we attempt to update the selected record, it is also wise to inquire whether an update operation is supported—which we do in the next step:

```
if( pRecordset->Supports( adUpdate )  
{
```

If the `Supports` function returns failure—indicating that the current OLE DB driver does not support updates for the selected data source—then there isn't much else we can do with the recordset.

However, assuming success—that is, a `TRUE` result—we can proceed with a second, relatively minor test to confirm that we have found the requested recordset.

```
if( pRecordset->GetadoEOF() ) return;
```

Naturally, if we can't find a matching record, we certainly can't update the record. And if so, we abandon everything and simply return.

In the next set of instructions, we check the Boolean variables for each edit field to decide which of these have actually been changed. We have no need to update fields with old information and, if any are unchanged, we simply skip them.

```
if( m_bAuthorChange )  
    pRecordset->PutCollect( L"Author",  
                           _variant_t( m_csAuthor ) );  
if( m_bNotesChange )  
    pRecordset->PutCollect( L"Notes",  
                           _variant_t( m_csNotes ) );  
if( m_bPriceChange )
```

```

        pRecordset->PutCollect( L"Price",
                                _variant_t( m_csPrice ));
    if( m_bReferenceChange )
        pRecordset->PutCollect( L"Reference",
                                _variant_t( m_csReference ));
    if( m_bTitleChange )
        pRecordset->PutCollect( L"Title",
                                _variant_t( m_csTitle ));

```

The PutCollect function (**Put** in **Collection**) requires two arguments: the name of the field to be updated (within the recordset) and the value to be written to the field. Notice that just as we cast retrieved BSTR data into a format acceptable for a CString object, we also have to cast CString arguments as _variant_t for the PutCollect function.

And, since these operations update only the recordset, we also need to call the recordset's Update function to write the information to the data source.

```

        pRecordset->Update( vNull, vNull );
        pRecordset->Close();

```

Once the data source has been updated, we close the recordset. Or, if any of our previous tests have failed, we report the problem in a brief message box.

```

    }
    else
        AfxMessageBox( "Updates are not supported" );
    }
    else
        AfxMessageBox( "Failed to open recordset" );
    }
    else
        AfxMessageBox( "Recordset creation failed" );
}

```

Of course, if the Supports test has reported failure, we've simply skipped the update operations and have offered a message to this effect. The error messages supplied here are intended for development and debugging only. In actual applications, a more informative error message would be appropriate.

Lastly, we still have the catch block for exception handling—but, once more, this is the same as before. (Monotonous, isn't it?)

```

        catch( _com_error &ampe )
        {
            // get info from _com_error
            ...
        }
        catch(...)
        {
            TRACE( "An unhandled exception has occurred" );
        }
    }
}

```

And this concludes the process of editing (or updating) a record. Not quite as convenient, perhaps, as in the previous examples, but not terribly difficult either.

Appending a New Record

To add a new entry to our data source, we begin with the OnNewTitle function, which is essentially the same as the process illustrated in the *BookList* demo in Chapter 17. The differences appear once we've finished entering the new data when the AddNewEntry function is called to create a new recordset and then to write the recordset to the data source.

The AddNewEntry function begins in a fashion very similar to the UpdateEntry function (discussed earlier in this chapter). One important difference, however, is found in the bstrQuery statement:

```

void CDetailsDlg::AddNewEntry()
{
    _RecordsetPtr pRecordset;
    _bstr_t        bstrQuery( "SELECT * FROM [Titles] WHERE "
                               "[Title_ID] IS NULL" );

```

Note that this query statement is asking for the selection to be made where the Title_ID field is null—in other words, an empty record. While this may sound slightly odd, the query is phrased to exclude records—that is, all records where a title ID already exists—because we don't want to overwrite an existing record.

WARNING:

Potentially, without this provision, you could overwrite all records in the database with your new entry.

```

_variant_t    vNull, vRecsAffected;
bool          bAdd = FALSE;
HRESULT       hRes;
CString       csEntry;
long          lIndex[1];
COleSafeArray vaFieldList, vaValueList;

```

Continuing with the declarations, two instances of the COleSafeArray class are declared as variables along with a single (long) index value. These will be used presently to build arrays of field names and field values to populate a new recordset.

```

vNull.vt = VT_ERROR;
vNull.scode = DISP_E_PARAMNOTFOUND;
try
{
    hRes = pRecordset.CreateInstance(
        _uuidof(Recordset) );
    if( SUCCEEDED( hRes ) )
    {
        pRecordset->PutRefActiveConnection(
            m_pConnection );
        hRes = pRecordset->Open( _variant_t( bstrQuery ),
                                vNull,
                                adOpenDynamic,
                                adLockOptimistic,
                                adCmdText );

        if( SUCCEEDED( hRes ) )
        {

```

As with previous examples, we create an instance of the recordset, reestablish a connection to the data source, and then open the recordset. (Remember, at this point, our query string specifies an empty recordset.)

Next, before proceeding too far, we want to query the recordset to ensure that the ADO AddNew method is supported.

```

        if( pRecordset->Supports( adAddNew ) )
        {

```

Usually, we'd expect any database to support entry of new material. There are, however, a variety of circumstances that might prevent us from writing new entries even when we are otherwise able to access the data source. For example, we might lack the appropriate permissions, or another process might have locked the database table, or...well, the possibilities are varied and it is appropriate to check.

But, assuming that we do have rights and permissions, the first task is to create and populate an array containing the names of the fields in the table. Granted, the construction is a bit cumbersome, but since we are using a COleSafeArray class, we do have some freedom and, rather than declaring an array of fixed size, we

begin by dimensioning the array.

```
vaFieldList.CreateOneDim( VT_VARIANT, 7 );
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Then, having established the array size, we're free to proceed by filling in the names of the table columns:

```
lIndex[0] = 0;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Title_ID" ) ) );
lIndex[0] = 1;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Title" ) ) );
lIndex[0] = 2;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Author" ) ) );
lIndex[0] = 3;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Notes" ) ) );
lIndex[0] = 4;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Price" ) ) );
lIndex[0] = 5;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Reference" ) ) );
lIndex[0] = 6;
vaFieldList.PutElement( lIndex,
                        &(_variant_t( L"Dealer" ) ) );
```

Essentially the same process is followed for the value list. Do observe, however, that the order in both arrays is consistent.

```
vaValueList.CreateOneDim( VT_VARIANT, 7 );
lIndex[0] = 0;
vaValueList.PutElement( lIndex,
                        &(_variant_t( m_lTitleID ) ) );
lIndex[0] = 1;
```

```

        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csTitle ) ) );
        lIndex[0] = 2;
        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csAuthor ) ) );
        lIndex[0] = 3;
        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csNotes ) ) );
        lIndex[0] = 4;
        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csPrice ) ) );
        lIndex[0] = 5;
        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csReference ) ) );
        lIndex[0] = 6;
        vaValueList.PutElement( lIndex,
                                &(_variant_t( m_csDealer ) ) );

```

Once the two arrays are populated, these are passed as arguments in the AddNew function, thus:

```

pRecordset->AddNew( vaFieldList,
                   vaValueList );

```

We do not need to call the Update method; the data source is updated automatically. Therefore, after adding the new recordset to the data source, all that remains is to close the recordset.

```

        pRecordset->Close();
    }

```

As an alternative, the AddNew method can be called without arguments, thus placing the current record pointer on the new record. After this, the details for the new record are established and the Update method is called to complete the operation.

Of course, as with the previous example, the AddNewEntry function concludes with an else statement and a series of message boxes to report if adding records isn't supported or if other errors have occurred.

```

        else
            AfxMessageBox( "New records not supported" );
    }
    else
        AfxMessageBox( "Failed to open recordset" );
    }
    else
        AfxMessageBox( "Recordset creation failed" );
    }

```

Lastly, we have the usual catch block for exception handling.

```

        catch( _com_error &ampe )
        {
            ...
        }
        catch(...)
        {
            TRACE( "An unhandled exception has occurred" );
        }
    }

```

Deleting a Record

In all of the previous examples, we've left the question of deleting records to the reader as an exercise instead of making deletion provisions within the demo applications. The essentials of the process for ADO, however,

are shown in the following code fragment:

```
try
{
    hRes = pRecordset.CreateInstance(
                                   _uuidof(Recordset) );
    if( SUCCEEDED( hRes ) )
    {
        pRecordset->PutRefActiveConnection(
                                           m_pConnection );
        hRes = pRecordset->Open(
                               _variant_t( bstrQuery ),
                                           vNull,
                                           adOpenForwardOnly,
                                           adLockOptimistic,
                                           adCmdText );
        if( ! pRecordset->GetadoEOF() )
        {
            pRecordset->Delete( adAffectedCurrent );
            pRecordset->Close();
        }
    }
}
catch ...
```

The process is very much like the Update and AddNew operations illustrated previously. However, considering the failure discussed earlier with the AddNew operation, it might be advisable to include a test such as:

```
if( pRecordset->Supports( adDelete ) ) ...
```

More on Supports Testing

Since you know which data sources you are programming for—and, presumably, you know whether the various operations are supported or not—why bother with these Supports tests?

In fact, there are several reasons:

- One of the advantages of ADO is that applications can be written to select different data sources that may support different capabilities, and which methods are valid can only be known at run time.
- Operations may be valid in general for a data source but may, at run time, be unavailable because:
 - Records have been locked by another process or application.
- The data source may be unavailable or may be flagged “read-only” at run time.
- Permissions may be granted for read-only access but not to write, update, or delete entries.

In any case, since methods may be available but not permitted under some circumstances, using the Supports test is a simple provision that avoids throwing exceptions unnecessarily.

Summary

Hopefully, this introduction to ADO and OLE DB has served to whet your appetite if only because ADO offers opportunities for flexible database access. Probably equally important is the fact that ADO can be used with a variety of development platforms, and that code can be moved from one platform to another with minimal changes.

The misfortune of ADO—as of Visual Studio 6—is that the available documentation fails to provide adequate information on the function (methods) available and on the numerous parameters and flags used with the ADO methods. There is, however, one title available from Wrox Press, *ADO 2.0 Programmer’s Reference*, which offers comprehensive coverage for the classes and methods comprising ADO.

Beyond this, in this chapter you have seen some of the possibilities provided by ADO even though, due to space limitations, we have covered only the basic operation types and these only in one of several possible forms.

In closing, if you are heavily involved in database operations, ADO is certainly worth your further exploration. Alternately, if your data source requirements are less extensive, the chances are that we have discussed the basic tools you might need already...but you still might find additional possibilities in investigating ADO further.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART V

Internet and Network Programming

- CHAPTER 19: Internet Support
- CHAPTER 20: Network Programming

CHAPTER 19

Internet Support

- An overview of Microsoft's Internet support
- The Winsock API functions
- The Internet (WinInet) API functions
- The ActiveX Web controls
- Active Channels and Active Server Pages

Developing Internet-aware software that runs on Windows poses many challenges for any software developer. These challenges involve keeping informed, making sure your applications remain compatible, and simply leveraging all the technology that is currently available and applicable. On the other hand, the opportunities are richer, and the tools and SDKs currently available are both more powerful and more usable than ever before. They enable the development of many ingenious new solutions for business, commerce, and home applications.

In this chapter, we'll begin with a survey of the Internet-related technologies Microsoft has been promoting over the past several years. Next, we'll take a look at some Internet-related concepts of concern to those developing Internet software. Then we will focus on the three chief toolkits or APIs for writing Windows Internet software: Winsock 2.0 (see the CSocket class), the Internet (WinInet) API, and the ActiveX Web controls. You can think of these as the low-level, medium-level, and high-level Windows Internet toolkits.

We'll also look at what Active Channels are and discuss how to convert a normal Web site to use an Active

Channel. Finally, we'll finish up with a brief discussion of Active Server Pages.

Four Years' Worth of Progress at Microsoft

At the time Windows 95 was introduced, Internet support in Microsoft operating systems was limited to dial-up and Ethernet TCP/IP support, plus a handful of Unix-style command-line utilities, such as FTP, ping, tracer, and the like. Actually, this was not a bad start, since by using the built-in TCP/IP stack and the FTP client, a user could download a Web browser and other Internet software and be set to access the Internet.

In the course of four years, Microsoft has done an amazing turnabout and embraced Internet technologies on a scale unprecedented even by Microsoft's own previous efforts in other areas. If this sounds even the least bit overdramatic to you, take a look at the software tools and applications listed in Tables 19.1 through 19.6. Microsoft has either adopted or introduced these since its "embrace the Internet" announcement made in January 1996.

TABLE 19.1: Improved Internet Connectivity in Operating Systems

Feature	Description
Multi-channel modem binding	Allows two or more modems (or other devices) to be treated as a single network connection—doubling or tripling (or more!) the throughput available to all Internet and/or other network-aware software. Improved Windows Internet Name Service (WINS) and Dynamic Host Configuration Protocol (DHCP) Allows WINS and DHCP to be administered more easily and to work seamlessly with other clients (such as Macintosh clients).
Point-to-Point Tunneling Protocol (PPTP)	Provides a secure way to transparently interconnect wide-area networks across insecure networks such as the Internet by encapsulating network packets inside a secure packet "envelope."
Windows ISDN	Brings integrated support for internal and external ISDN modems to Windows 98 and NT/2000. Includes support for multiple-channel binding.

TABLE 19.2: Tools for Web Page Design and Management

Tool	Description
Microsoft FrontPage	Supports HTML page authoring and Web site management.
Microsoft GIF Animator	Supports creation of GIF and animated GIF files for use with Web pages.
Microsoft Image Composer	Simplifies creating and manipulating images for placement on Web pages.
Microsoft Office 2000	Integrated HTML authoring (with Microsoft Word) and Web page management allowing documents to be saved as Web pages or frames.

TABLE 19.3: Internet Servers

Server	Description
Internet Information Server (IIS)	Provides a high-performance Web server with support for WSAPI .DLL extensions, secure Web pages, performance monitoring, and flexible FrontPage Server extensions.
Peer Web Servers	Provides built-in peer Web and FTP servers.
Microsoft (Catapult) Proxy Server	Allows multiple PCs to connect to the Internet via a single server connection, even using other protocols (instead of TCP/IP). Also fetches and caches most-often used Web pages for all users.
Internet News Server	Gives companies (or individuals) their own self-managed newsgroups.
Chat Server	Keeps a running round-table dialog, displayed via Web pages.
White Pages Indexing/Locator Server	Automatically indexes an entire Web site, providing keyword searches.
Content Replication Server	Assists in porting and/or mirroring part or all of a Web site's content from one server to another.

Server Personalization System	Provides Web content customized for each connected user, as an addition to IIS.
Mail Server	Provides sites with POP and SMTP server support (plus MS Mail support).
Merchant Server	Supports “store-front” shopping via Web pages.

TABLE 19.4: Internet-Related Programmer’s Tools

Tool	Description
Visual J++ 6.0	Provides a Visual Java environment, including Java compiler, debugger, and Windows SDK class libraries.
Jscript	Allows embedding of Java script applets in HTML pages (supported in IE).
Visual Basic 6.0	Allows development of ActiveX Controls, component packaging for Internet downloading, and native code support.
VBScript	Provides a Visual Basic scripting language for embedding in HTML pages (supported in IE).
Visual C++ 6.0	Provides enhanced support for various Internet APIs, DCOM, ActiveX, and so on.
Visual Interdev 6.0	Provides tools for creating dynamic Internet and intranet Web applications.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

NOTE:

Although we do not cover Java, Jscript, or VBScript here, these are significant application-development technologies available. These tools have entire books devoted to them—Java in particular!

TABLE 19.5: Internet-Related SDKs and APIs

SDK or API	Description
Winsock 2.0	Provides expanded support for socket-style Internet programming, including AppleTalk, IPX, and other networking protocols, in addition to TCP/IP. Also supports encrypted socket communications using PCT and SSL.
Internet API	Simplifies adding TCP/IP support to applications, encapsulating and abstracting Winsock-style functionality.
ActiveX Web Control Pack	Simplifies adding Internet support to applications, via drop-in ActiveX objects. Includes Web browser, FTP, e-mail, and news client objects.
Windows SDK for Java	Provides class libraries for Java applets to access Windows-specific functionality.
Game SDK	Includes support for multiplayer gaming over the Internet.
Crypto API	Provides public and private key encryption and decryption, code and message signing and authentication.
Uploading API	Provides a set of functions to simplify the job of uploading or moving Web pages to and from a Web site.
Ratings API	Provides functions to set content restrictions and query content ratings tags from HTML pages.
URL Security Zoning	Divides URL name spaces into zones that can have different levels of trust assigned to them (supported in IE 4 and accessed either via Control Panel or browser settings).

Secure Sockets Layer (SSL) for Winsock	Provides encrypted socket-style communications as an extension to Winsock 2.0.
Active Channels	Allows Web sites to support Web-casting of content to channel-viewing software such as IE 4 or the Active Desktop of Windows 98.
Cascading Style Sheets (CSS)	Provides a (non-Microsoft) standard that adds some 70 new style properties, extending control over the layout and presentation of Web pages (supported in IE 4).
Dynamic HTML	Enables Web authors to dynamically change the rendering and content of a document (without necessarily reloading the page).
Design-time Control SDK	Allows creation of controls that can simplify designing Web page content.
NetMeeting SDK	Allows integration of real-time Internet multimedia conferencing via software or embedded in Web pages.
WebWizard SDK	Allows creation of wizards that can automate the creation of specialized Web pages.
Active Server Pages	Provides pages that contain scripting or ActiveX controls that are executed on the Web server.
Personal Information Exchange (PFX)	Provides protocols for securely transferring personal identity information, private keys, personal certificates, and other miscellaneous “secrets” over the Internet.
Web Content Management	Provides a Microsoft initiative to reduce the complexity of managing larger Web sites (currently, primarily by using features of Visual SourceSafe).
Cabinet (CAB) SDK	Simplifies compression, packaging, and online distribution of Java classes and other software.
Common Internet File System (CIFS)	Provides a standard remote file-system-access protocol for enabling groups of users to work together and share documents across the Internet or within their corporate intranets.
HTMLHelp API	Modeled on the WinHelp function, allows easier porting of Windows help files to the new HTML Help format.
Web Publishing API	Provides functions to simplify the process of uploading Web pages to a Web site.

NOTE:

Although we won’t go into detail about all of the APIs, you should be aware of their suitability for more specialized uses. The Game SDK includes much more than support for multiplayer game communications over the Internet. DirectPlay, DirectSound, DirectVideo, Direct3D, and related technologies provide speedy access to a wide variety of video, sound, and control hardware. The SSL API is an extension to Winsock 2, providing encrypted data flow through the normal socket architecture. The Ratings API provides access to functions useful for setting, checking, and managing access to classes of information that have been assigned particular ratings, such as for violence and profanity. The crypto API, which is already being used by Microsoft’s Internet Explorer, is covered in detail in Chapter 11, “Security and Cryptography.”

TABLE 19.6: Other Internet-Related Technologies

Item	Description
Internet Explorer	A Web browser that now supports Netscape plug-ins, ActiveX controls, VBScript, Jscript, SSL, code signing and certificates, pop-up menu controls, auto-installation of ActiveX controls, and content ratings system.
Internet Assistants	Programs that export data from Access, Excel, PowerPoint, Word, and Schedule+ to formatted HTML pages.
Data Viewers for Word, PowerPoint, and Excel	Viewers that display Word, PowerPoint, and Excel files on stations not equipped with Word, PowerPoint, or Excel.
NetMeeting	A multiuser collaboration tool that supports text and voice chatting, white boarding, and file transfers.

ISAPI and FrontPage Server Extensions	Support for CGI-like server extensions, with less overhead than an interpreter or loading and unloading standard binary executable files.
DCOM (Distributed Component Object Model)	Support for remote-object access and execution using TCP/IP (as well as other protocols).
Authenticode	Support for code-signing and authentication using certificates.
CAB Files	Files initially used for installing Microsoft software, now expanded for simplified downloading of Java source files.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

In addition to offering the tools and applications listed in these tables, Microsoft has been rushing to include Internet hooks and functionality into *all* of its software applications. Furthermore, it has switched Microsoft Network (MSN) to be Internet-based rather than proprietary, and has moved all of its public (and beta) support forums from CompuServe over to the Internet. Each individual Microsoft product has its own Web page, and there are currently several hundred public Internet newsgroups run by Microsoft alone, not including many independent newsgroups, which focus, for example, on various elements of Win32 programming.

This explosive usage of the Internet has benefited both software end users and software developers. On the Internet, you can find more accessible (and timely) documentation, significant peer support, software upgrades, bug fixes, and enhancements.

And this survey covers recent developments at only one (admittedly rather huge) company. We could also review (were enough space available) all the Internet work being done by Netscape, Sun, IBM, Symantec, Apple, and literally thousands of other companies. Much of their work also has some bearing on Windows Internet software development.

Internet-Related Concepts

Those who are interested in designing Internet software need to have some understanding of many Internet-related concepts and terms. Here, we will briefly cover IP addressing, Internet protocols, and sockets.

IP Addresses

Most new Internet users quickly become familiar with the fact that they have an e-mail address and an IP address. Setting up a computer for TCP/IP networking (using a TCP/IP protocol stack, as it is sometimes called), typically involves specifying IP addresses for a Domain Name server and network gateway, and entering POP and SMTP server names. IP addresses are typed in using dotted, or dotted-quad notation, where each byte of the 4-byte address is displayed in decimal and separated from the next byte value with a period. As you probably know, this dotted notation is simply to allow easier reading by human users. Computers and other hardware on the Internet always use the 4-byte (32-bit) values directly.

Because every computer on the Internet has its own IP address (at least while it's connected) and because the number of computers joining the Internet has been and continues to increase rapidly, work is being done to

devise a new address scheme using wider network addresses. However, there are still some issues remaining to be hashed out, so for the near future, we can continue programming using the 32-bit Internet (IP) addresses.

Internet Protocols

IP (Internet Protocol) is the Internet “delivery system” protocol, which underlies the other Internet protocols. IP itself uses connectionless datagrams; thus IP packets are often called IP *datagrams*. TCP/IP is actually a whole suite of related protocols, some higher level than others. At the lower levels, for example, are the two main packet protocols that transport data in either sequenced, two-way “conversations” or connectionless “radio-style” (one-way) transmissions: TCP and UDP.

TCP (Transmission Control Protocol) provides a sequenced, two-way connection-based protocol using byte streams. It is considered “reliable” because it takes responsibility for check-summing each received packet and re-requesting any dropped or damaged packets.

UDP (User Datagram Protocol), on the other hand, is termed “unreliable,” in the sense that it provides connectionless transport of data packets, and therefore cannot support built-in error checking. Instead, it acts analogously to a radio signal—you can be certain that it was transmitted, but you cannot verify that it was received.

These two approaches are complementary. Connectionless transport is faster, because it does not have the overhead associated with verification of accurate reception and transmission. TCP, with its greater overhead, ensures that data correctly reaches its destination, and thereby makes up for a multitude of potential failure points within the underlying networking hardware (cabling, switching mechanisms, and so on).

At a higher level is a set of protocols with more specialized purposes. These higher-level protocols are the various service-related protocols, which are used to transfer e-mail, files, Web documents, and so on. They include the following:

Simple Mail Transfer Protocol (SMTP) Commonly used for sending e-mail to a server.

Post Office Protocol 3 (POP3) Commonly used for retrieving e-mail from a server.

File Transfer Protocol (FTP) Widely used for uploading and downloading files.

Hypertext Transfer Protocol (HTTP) Used as the standard World Wide Web (WWW) protocol.

Remote Terminal Access (Telnet) Used for remote login and interactive access to remote servers.

Whereas the lower-level transport protocols package their routing information and data using rather complex structures and packet arrangements, the higher-level protocols use techniques that human programmers generally find friendlier to implement. These higher-level service protocols have the following common properties:

- They start with recognizable four-character command strings.
- They assume a client and a server conversation, established via a bi-directional socket.
- They use a numerical scheme for response codes.

TIP:

Later in this chapter, we will look at a program that implements SMTP to send an e-mail message to a mail server. You may find it interesting to experiment directly with these protocols by using your Windows 2000 Telnet client to send properly formatted protocol strings to an FTP, SMTP, POP3, or HTTP server, and view the incoming replies generated by the server.

Sockets and Winsock

Most Internet programming has traditionally been done using the *sockets* paradigm. In the early days, when TCP/IP became integrated with the Unix operating system, a set of functions was originated that came to be called the Berkeley sockets standard. These functions let you create a socket, listen for incoming connections (in the case of a server), connect the socket to a given host computer and port number (as a client), and send and receive data via the socket.

A socket is analogous to a file handle and, on some versions of Unix, can be used directly with the file I/O functions. There are two types of sockets, corresponding to the connected and connectionless protocols:

- *TCP sockets* require that a connection be established between a socket and a host computer.
- *UDP sockets* require only that you specify where the data should end up (a specific port number at a

specific IP address).

Several years ago, a consortium of individuals and companies got together and came up with a version of socket functions for Windows. Winsock, as it is called, is somewhat different from Berkeley sockets. Because earlier versions of Windows, unlike Unix, were not a true multitasking operating system, certain provisions had to be made. These provisions generally took two approaches. First, functions that might take a while to receive a response or otherwise complete execution could be “blocking,” or simply not return until they were finished. Although this works under Windows, it understandably isn’t considered good programming practice. Using lots of blocking functions in an application can cause the application to become sluggish at best, or completely unresponsive (or crash!) at worst.

Therefore, Winsock also provides a set of asynchronous functions, which can notify a Windows application when they have finished executing by sending a special message to a specified window. The application then simply needs to call the function, continue other processing, and check for a function return message in the message queue. This worked about as well as could be expected, considering that 16-bit Windows cannot do preemptive multitasking of Windows applications.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Windows 2000, on the other hand, has no problem with multitasking (although to retain backward compatibility, the 32-bit Winsock functions remain almost unchanged). With these versions of Windows, an application can use blocking functions in the Unix style of spawning a separate thread to allow continued execution of the application while the function completes. Certain functions that block for the sake of ease of use in Windows 3.1 don't need to block in Windows 2000. Also, by and large, GUI-based Win32 applications (which have a window to receive notification messages) often use the asynchronous Windows versions of the socket functions, although console applications and some multithreaded applications go with the more "traditional" synchronous socket functions.

The Winsock 2 API Functions

Winsock 2 is important for several reasons:

- It supports various protocols.
- It's the best way to produce fast, close-to-the-hardware, transport code.
- It's the only way to go when writing servers or when speed and efficiency of throughput are essential (more important than development time, for example).
- It provides more cross-platform support.

In its current version, Winsock 2 maintains most functions of previous Winsock releases and adds a few functions that improve multi-protocol support. Although Winsock is probably most often used for TCP/IP, it also supports socket connections over several other protocols, including IPX, SPX, Banyan VINES, and AppleTalk. In this chapter, we will be looking at Winsock in its TCP/IP capacity, but keep in mind that socket applications can often be "ported" to use other protocols with little additional work.

Within the Winsock set of functions, there are both synchronous and asynchronous socket functions. Additionally, there are some data-conversion functions and database-lookup functions.

Socket Functions

Table 19.7 lists the Winsock socket functions. This list is the traditional set of synchronous functions.

TABLE 19.7: Winsock Socket Functions

Function	Description
accept	Accepts a connection on given socket
AcceptEx	Accepts a new connection, returns local and remote addresses, plus the first block of data sent by the client
bind	Associates a local address with a socket
closesocket	Closes a socket
connect	Connects a socket with a given peer
GetAcceptExSockaddrs	Parses data from AcceptEx and passes the local and remote addresses, along with the first block of data received upon connection
ioctlsocket	Gets or sets socket mode parameters
listen	Places a given socket in listen mode
recv	Retrieves data from a given socket
recvfrom	Retrieves a datagram along with a source address
select	Determines the status of one or more sockets
send	Sends data on a given (connected) socket
sendto	Sends data to a given destination
Setsockopt	Sets a socket option
shutdown	Disables sending and/or receiving on a specified socket
socket	Creates a socket bound to a given transport service provider
TransmitFile	Transmits file data over a connected socket handle; uses the operating system's cache manager to retrieve the file data; provides high-performance file data transfer over sockets
WSAEnumProtocols	Retrieves information about available transport protocols

Data-Conversion Functions

Another set of Winsock functions deals with converting data between *network byte order* and *host byte order*. It is recommended that these functions be used even where it is known that the order is the same on both machines, as this makes for more consistent code, which is more easily portable. Table 19.8 lists the data-conversion functions.

TABLE 19.8: Winsock Data-Conversion Functions

Function	Description
htonl	Converts a 32-bit value from host to TCP/IP network byte order
htons	Converts a 16-bit value from host to TCP/IP network byte order
inet_addr	Converts a dotted IP address string to an unsigned long value
inet_ntoa	Converts a network address as unsigned long into a dotted IP address string
ntohl	Converts a 32-bit value from TCP/IP network byte order to host byte order
ntohs	Converts a 16-bit value from TCP/IP network byte order to host byte order

Database-Lookup Functions

Another, smaller set of Winsock functions are called the *database-lookup* functions. These functions, which are listed in Table 19.9, have the duty of looking up correct IP addresses, domain names, service IDs, and protocol port numbers. The database functions behave much like their Unix counterparts. The services information and (usually) a small number of domain names and IP addresses are typically stored in two text files on each host computer: the HOSTS and SERVICES files. The socket functions make use of these as their first source of information for addresses or service numbers during address or service resolution.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 19.9: Winsock Database-Lookup Functions

Function	Description
gethostbyaddr	Gets host information for a given address
gethostbyname	Gets host information for a given host name
gethostname	Gets the standard host name for local machine
getpeername	Gets the address of the peer connected to a given socket
getprotobyname	Gets protocol information for a given protocol (name)
getprotobyname	Gets protocol information
getservbyname	Gets service information for a given service name and protocol
getservbyport	Gets service information for given port number and protocol
getsockname	Gets the local name for a given socket
getsockopt	Gets the current value for a specified socket option

HOSTS and SERVICES Files

Whether on a Windows 2000, Windows 98, or Unix computer, the HOSTS file contains pairs of addresses and host names, and the SERVICES file contains information about “well-known” services, along with their port numbers, the low-level protocol used, and one or more optional alias names. On Windows 2000, the SERVICES and HOSTS file can be found in the \WIN2000\SYSTEM32\DRIVERS\ETC directory. (On Windows 98, these files are located in the \WINDOWS directory.)

Table 19.10 is part of a typical SERVICES file, taken from a Windows 2000 workstation. (Similar files exist on Windows 95, 98, and NT stations.) Note how each entry matches up a service name with a port and protocol type, and one or more optional service name aliases.

TABLE 19.10: Typical SERVICES File

Service	Name Port # / Protocol	Aliases...	Comments

echo	7/tcp		
discard	9/tcp	sink null	
systat	11/tcp	users	#Active users
daytime	13/udp		
qotd	17/udp	quote	#Quote of the day
chargen	19/udp	ttytst source	#Character generator
ftp-data	20/tcp		#FTP, data
ftp	21/tcp		#FTP. control
telnet	23/tcp		
smtp	25/tcp	mail	#Simple Mail Transfer Protocol
time	37/udp	timserver	
rlp	39/udp	resource	#Resource Location Protocol
nameserver	42/udp	name	#Host Name Server
nicname	43/tcp	whois	
domain	53/udp		#Domain Name Server
bootps	67/udp	dhcps	#Bootstrap Protocol Server
bootpc	68/udp	dhcpc	#Bootstrap Protocol Client
tftp	69/udp		Trivial File Transfer
gopher	70/tcp		
finger	79/tcp		
http	80/tcp	www www-http	#World Wide Web
kerberos-sec	88/udp	krb5	#Kerberos
hostname	101/tcp	hostnames	#NIC Host Name Server
iso-tsap	102/tcp		#ISO-TSAP Class 0
rtnet	107/tcp		#Remote Telnet Service
pop2	109/tcp	postoffice	#Post Office Protocol - Version 2
pop3	110/tcp		#Post Office Protocol - Version 3

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The following shows a typical HOSTS file. This file is usually kept quite small, with most names being resolved through a Domain Name Service (DNS) server.

```
# HOSTS file used by Microsoft TCP/IP
102.54.94.97      rhino.acme.com      # source server
38.25.63.10       x.acme.com          # x client host
127.0.0.1         localhost
```

In this file, each IP address is matched with a host name. Note the standard address of 127.0.0.1, which is mapped to the local host. By convention, 127.0.0.1 is always a reference to the local host itself. Pinging 127.0.0.1 is sometimes used as a test of one's own IP address.

NOTE:

In Windows 2000, there is also typically a file similar to the HOSTS file called LMHOSTS. This performs a similar function; however, the host names are NetBIOS names, and certain extensions are allowed (such as #DOM: to handle Windows NT domains).

Most often, an address lookup will not be resolved by finding a corresponding entry in the HOSTS file. Instead, the lookup request is passed along to the DNS server, which either fills the lookup request, or passes along the request to a first-level DNS server. These IP address and name servers contain many, but by no means all, names and corresponding IP addresses, as registered by InterNIC. (InterNIC is responsible, in the United States, for maintaining the DNS entries for all top-level domains, except for the *mil* or military domain.) If a match is not found on a first-level server, the request is passed down to the next DNS server in the hierarchy, and so on. This continues until either a match is found or no entry is found (in which case, you may see an error message).

The *FindAddr* demo, discussed a little later in this chapter, is an example of using host name-to-IP address resolution (as well as IP address-to-host name reverse resolution).

Asynchronous Versions of the Winsock Functions

To do asynchronous Windows socket programming, you use the WSA version of a socket function, which

typically takes a window handle as a parameter and posts a status or completion message to that window's message queue when the function completes or needs to communicate information to the application. Because Windows 2000 supports multithreading, you can also use the synchronous functions by spawning off a separate thread, thus freeing the main application to continue with other processing while waiting for a function to finish executing. Table 19.11 lists the asynchronous functions.

TABLE 19.11: Asynchronous Winsock 2 Functions

Function	Description
WSAAsyncGetHostByAddr	Gets host information for a given address
WSAAsyncGetHostByName	Gets host information for a given host name
WSAAsyncGetProtoByName	Gets protocol information for a given protocol name
WSAAsyncGetProtoByNumber	Gets protocol information for a given protocol number
WSAAsyncGetServByName	Gets service information for given service name and protocol
WSAAsyncGetServByPort	Gets service information for given port number and protocol
WSAAsyncSelect	Requests Windows message-based notification of network events occurring on specified socket
WSACancelAsyncRequest	Cancels an (incomplete) asynchronous socket operation
WSAAccept	Conditionally accepts a connection based on the return value of an (optional) condition function, creating or joining a socket group
WSAAddressToString	Converts all addresses in a SOCKADDR structure into human-readable string representations
WSACleanup	Terminates use of Winsock.DLL
WSACloseEvent	Closes an event object handle
WSAConnect	Connects to a peer, exchanges connect data, and specifies needed quality of service based on supplied flow specification
WSACreateEvent	Creates a new event object
WSADuplicateSocket	Returns a WSAPROTOCOL_INFO structure, which can be used to create a new socket descriptor for a shared socket
WSAEnumNameSpaceProviders	Retrieves information about the available name space providers
WSAEnumNetworkEvents	Retrieves information about which network events have occurred on a given socket
WSAEnumProtocols	Retrieves information about available transport protocols
WSAEventSelect	Specifies an event object to be associated with given FD_XXX network events
WSAGetAddressByName	Resolves an address by name and returns the first available result
WSAGetLastError	Returns the error status for the last failed Winsock operation
WSAGetOverlappedResult	Returns the results of an overlapped operation on the given socket
WSAGetQOSByName	Initializes a QoS (Quality of Service) structure based on a named template
WSAGetServiceClassInfo	Retrieves the class information for a service class from a specified name space provider
WSAGetServiceClassNameByClassId	Returns the generic service name for a given service type
WSAHtonl	Converts an unsigned long from host byte order to network byte order
WSAHtons	Converts an unsigned short from host byte order to network byte order
WSAInstallServiceClass	Registers a service class schema within a name space
WSAIoctl	Sets the mode of a given socket and also configures sockets for secure communications using SSL or PCT
WSAJoinLeaf	Joins a leaf node into a multi-point session, exchanges connection data, and specifies needed QoS based on supplied flow specification
WSALookupServiceBegin	Returns a handle that can be used in further client queries to search for specified service information
WSALookupServiceEnd	Frees a query handle created by WSALookupServiceBegin
WSALookupServiceNext	Retrieves the next set of service information using given query handle
WSANtohl	Converts an unsigned long from network byte order to host byte order

WSAProviderConfigChange	Notifies an application when the provider configuration has changed
WSANtohs	Converts an unsigned short from network byte order to host byte order
WSARecv	Receives data from a given socket
WSARecvDisconnect	Terminates reception on a given socket, retrieving disconnect data if applicable
WSARecvFrom	Retrieves a datagram and the source address
WSARemoveServiceClass	Permanently unregisters a service class schema
WSAResetEvent	Resets the state of a specified event object to be “nonsignaled”
WSASend	Sends data on a connected socket and permits asynchronous, or overlapped, I/O
WSASendDisconnect	Initiates termination of a given socket connection and sends disconnect data
WSASendTo	Sends data to a given destination, using overlapped I/O where applicable
WSASetEvent	Sets the state of the specified event object to be “signaled”
WSASetLastError	Sets the error code which is returned by subsequent calls to WSASetLastError
WSASetService	Registers or unregisters a service instance within one or more name spaces
WSASocket	Creates a socket that is bound to a specific transport service provider, optionally creating or joining a socket group
WSAStartup	Initiates use of Winsock.DLL by a process
WSAStringToAddress	Converts a human-readable address string to a socket address structure
WSAWaitForMultipleEvents	Returns one or all of the specified event objects are in the signaled state, or when the time-out expires

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The *FindAddr* Demo: An IP Address Lookup Example

As our first Internet example, the *FindAddr* demo will perform an IP address lookup using a host name—an identifier using the form: www.hostname.org. If the host name lookup succeeds and an IP (DNS) address is returned, the IP address will be used to look up the host name—a process known as *reverse lookup*.

NOTE:

The *FindAddr* demo is included on the CD that accompanies this book, in the Chapter 19 folder.

For simplicity, we'll create the program using the MFC AppWizard as a dialog-based application and (since we need WinSocket support) in Step 2, the option for including WOSA support (Windows Sockets) is selected as shown in Figure 19.1.

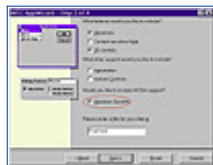


FIGURE 19.1 Including Windows Sockets in a project

With this provision, setup and initialization for Windows Sockets is automatic. In the `StdAfx.h` header file for the project, you will find:

```
#include <afxsock.h>           // MFC socket extensions
```

This include statement provides access to the Windows Sockets functions and, in the `FindAddr.CPP` program file, the actual initialization of the WinSock services occurs in the `InitInstance` function.

```
BOOL CFindAddrApp::InitInstance()  
{  
    if( ! AfxSocketInit() )  
    {
```

```

        AfxMessageBox( IDP_SOCKETS_INIT_FAILED );
        return FALSE;
    }

```

This is everything needed to allow the application full access to the WinSock services and we have no need to mess with version support or to try to find out which version is available.

Since the *FindAddr* demo is dialog-based, all of the subsequent processing occurs in *FindAddrDlg.cpp*—in this case, in the *OnOK* procedure. The application itself is shown in Figure 19.2, where you are prompted to enter a URL in the edit box.

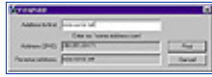


FIGURE 19.2 The *FindAddr* Demo

When the Find button is clicked—assuming that an Internet or network connection is open—the *FindAddr* program will look up and retrieve the DNS address and then use the DNS address to retrieve the server name.

The *OnOK* procedure begins with two variables:

```

void CFindAddrDlg::OnOK()
{
    LPHOSTENT lpHostEntry;
    DWORD      dwIPAddress;

```

The *LPHOSTENT* variable is a long pointer to a *HOSTENT* (Host Entry) structure, which is returned by the *gethostbyname* function. The *HOSTENT* structure is defined as:

```

struct hostent
{
    char FAR *      h_name;
    char FAR * FAR * h_aliases;
    short           h_addrtype;
    short           h_length;
    char FAR * FAR * h_addr_list; };

```

The *HOSTENT* fields are described in Table 19.12.

TABLE 19.12: The *HOSTENT* Structure

Member	Description
<i>h_name</i>	Official name of the host (PC).When using DNS or similar, this will be the Fully Qualified Domain Name (FQDN) or, using a local “hosts” file, the first entry following the IPaddress.
<i>h_aliases</i>	A null-terminated array of alternate names.
<i>h_addrtype</i>	The type of address being returned.
<i>h_length</i>	The length, in bytes, of each address.
<i>h_addr_list</i>	Uses a NULL-terminated list of addresses for the host with addresses returned in network byte order. For compatibility with older software, the macro <i>h_addr</i> is defined to be <i>h_addr_list[0]</i> for compatibility with older software.

This structure is allocated by Windows Sockets, and applications should not attempt to modify the structure or to free any of the structure’s components.

WARNING:

Since only one copy of the *HOSTENT* structure is allocated for each executing thread, processes using this structure should copy any information needed to local variables before issuing any further WinSock API calls.

In the *FindAddr* demo, the first step in response to the Find button (in the *OnOK* function) is to call *UpdateData* to retrieve the entry from the edit box and, of course, to ensure that the user did provide an entry.

```

UpdateData( TRUE );

```

```

if( m_csURL.IsEmpty() )
{
    m_csIPAddress = "Enter a URL address above";
    UpdateData( FALSE );
    Invalidate();
    return;
}

```

Once we're assured that we do have an entry, the `gethostbyname` function is called to retrieve the host's DNS address:

```

lphostentry = gethostbyname( m_csurl );
if( lphostentry == NULL )
{
    m_csipaddress.Format( "Unable to find %s", m_csurl );
    UpdateData( FALSE );
    Invalidate();
    return;
}

```

If the `lphostentry` is returned as a `NULL` value, then a mistake has occurred and all that can be done is to report the error.

Ideally, however, the assumption is that the host name was valid and the returned `lphostentry` structure contains the requested information. So, to extract the information, we use the `inet_ntoa` function:

```

m_csipaddress = inet_ntoa( *(LPIN_ADDR)
                           *lphostentry->h_addr_list );

```

The `inet_ntoa` function converts the `DWORD` value (the first entry in the `h_addr_list` list) to a dot-formatted string (`nnn.nnn.nnn.nnn`). Computers, of course, simply use the `DWORD` value but the dot format is for human convenience.

As the next step, the `inet_addr` function is used to convert the dot-formatted string back to a `DWORD` value.

```

dwipaddress = inet_addr( m_csipaddress );

```

Here we're simply following the caution about relying on local variables for our data rather than assuming that the value in the `HOSTENT` structure has remained untouched.

Then, for the reverse location operation, the `dwipaddress` value is passed to the `gethostbyaddr` function that, again, returns a `HOSTENT` structure.

```

lphostentry = gethostbyaddr( (LPSTR) &dwipaddress,
                             sizeof( dwipaddress ),
                             AF_INET );

```

The `AF_INET` constant is defined in `WinSock.H` and identifies the type of address (address family) to retrieve. In this case, the requested family includes internetwork: UDP, TCP, and so on—that is, the TCP/IP family of network operations. (Refer to `WinSock.h` for a complete list of address families.)

NOTE:

Reverse lookups are used in some TCP/IP server programs, such as e-mail and Web servers, to verify that a given client address is authentic. Web servers usually let you disable this reverse-lookup feature, since performing the lookup will add extra wait time to each incoming client connection.

Again, there are provisions included to report an error if one occurs, but in brief, this is the finish for the *FindAddr* program.

```

if( lphostentry == NULL )
    m_csRevAddress = "Reverse find failed";
else

```



```
        m_csrevaddress = lphostentry->h_name;  
        UpdateData( FALSE );  
        Invalidate();  
        // CDiallog::OnOK();  
    }
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

As a small bonus, since we've used the MFC socket extensions, there are no requirements for cleanup. In this form, we can simply use the WinSock APIs as required and forget them without worrying about startup and shutdown procedures.

And, of course, the supplied `Cdialog::OnOK()` statement has been commented out so that *FindAddr* does not close and exit as soon as a Find operation is completed.

Note also that, in addition to address and domain-name lookup functions, Winsock also has a port lookup function, which returns the port number used for a given TCP/IP protocol or service, which then is used to bind a socket to the desired port. There's an example of this in the *SendMail* demo, discussed next.

The *SendMail* Demo: Sending E-Mail

Our second Winsock example uses not only the TCP stream-oriented protocol, but also the higher-level SMTP to send e-mail. Again, the demo has been written as a dialog-based application that expects four entries: a mail server address, both from- and to-e-mail addresses, and some form of message text. The program connects to the specified mail server, and e-mails the text file to the specified "to" address while showing the results of the several send and recv operations in a list box (at right) as shown in Figure 19.3.



FIGURE 19.3 Using *SendMail* for a message

Of course, if an error occurs, the error will be reported in the results list.

NOTE:

The *SendMail* demo is included on the CD that accompanies this book, in the Chapter 19 folder.

Like the *FindAddr* demo, the *SendMail* demo includes WinSock support, which was selected in Step 2 of the AppWizard setup.

Also, for clarity, CRLF is defined so we can use it for adding carriage return/line feeds to the SMTP strings

and to each line of the outgoing e-mail message.

```
#define CRLF "\r\n" // carriage-return/line-feed pair
```

Now, we start the main program declaration, and declare needed variables.

```
void CSendMailDlg::OnOK()
{
    LPHOSTENT    lphostentry = NULL;
    LPSEVENT     lpServEntry = NULL;
    SOCKET       hSocket;
    SOCKADDR_IN  SockAddr;
    int          nProtocolPort;
    CString      csMsg, csResult;
    char         szBuffer[4096];
}
```

Note that several of the variables are identical to those used in the *FindAddr* demo. A socket address (input) or SOCKADDR_IN is necessary for binding the socket to the mail port. The iProtocolPort variable stores the mail port number, and hServer is the socket we will create to communicate with the mail server.

Initially, the OnOK procedure clears the error (event) list and then calls UpdateData to retrieve entries from the various edit boxes.

```
m_lbErrors.ResetContent();
UpdateData( TRUE );
```

Once the entries have been retrieved, a series of fairly simple checks are made to ensure that none of the fields were blank.

```
if( m_csServerName.IsEmpty() )
{
    ReportEvent( "Server name required" );
    return;
}
if( m_csFromAddress.IsEmpty() )
{
    ReportEvent( "From address required" );
    return;
}
if( m_csToAddress.IsEmpty() )
{
    ReportEvent( "To address required" );
    return;
}
if( m_csMessage.IsEmpty() )
{
    ReportEvent( "No message to send" );
    return;
}
```

NOTE:

At this point in the program, a couple of provisions are omitted from the present discussion—the means for checking and for establishing a network or an Internet connection. These provisions are discussed following as part of the WinInet API coverage—see “Opening A Dial-Up Connection,” later in this chapter.

The gethostbyname function was demonstrated previously in the *FindAddr* demo, but is used here to retrieve an IP address.

```
// look up server's IP address
lphostentry = gethostbyname( m_csservername );
```

```

if( lphostentry == NULL )
{
    csmmsg.Format( "Can not find SMTP server (%s)",
                  m_csservername );
    ReportEvent( csmmsg );
    return;
}

```

If we didn't get an IP address—meaning that no SMTP server was found—the error is reported and the procedure returns to allow the user to correct their entry. However, assuming that the SMTP server is found, our next step is to create a TCP/IP socket.

```

Hsocket = socket( PF_INET, SOCK_STREAM, 0 );

```

The first parameter specifies that this socket will be used in a TCP/IP context (as opposed to some other network protocol). Next, we want to specify a stream-type socket, because we will be using SMTP, which uses TCP, which is a stream protocol. And, since we don't need a specific protocol type at this point to create the socket, we can specify a zero as the third parameter.

```

if( Hsocket == INVALID_SOCKET )
{
    ReportEvent( "Can't open mail server socket" );
    return;
}

```

Again, the results of the operation are checked and failure is reported.

Assuming the socket is valid, the next step is to request a mail server—that is, the port number assigned to the “mail” service, if one exists—or, if no mail server is reported, to use the default SMTP port.

```

// get mail service port
lpserverentry = getservbyname( "mail", 0 );
// use SMTP default port if no other port specified
if( lpserverentry == NULL )
    Nprotocolport = htons( IPPORT_SMTP );
else
    Nprotocolport = lpserverentry->s_port;

```

If there isn't a specific entry for the mail service, we'll use the default port number for SMTP.

WARNING:

In many cases today, partially as an effort to prevent e-mail “spam” (unwanted, unsolicited e-mail), requesting a server in the form “www.domain.com” will appear to work at this point but will fail with a *refusal of service error* when the `connect` function is called. Instead, if you are having trouble, try requesting a server in the form “`mail.domain.com`”.

The next step is to fill in the family, port, and address fields of the input socket address structure, which will be used to connect to the just-created socket. The socket family is type `AF_INET` (for TCP/IP), the port number is the mail service port that was just determined, and the address filled in should be that of the mail server, which was retrieved in the call to `gethostbyname`.

```

// setup a socket address structure
SockAddr.sin_family = AF_INET;
SockAddr.sin_port = Nprotocolport;
SockAddr.sin_addr = *( (LPIN_ADDR)
                       *lphostentry->h_addr_list );

```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Next, the stream socket is connected to the mail server, at the mail service port. Note that connect returns zero if successful. Also, the third parameter must specify the size of the socket address structure, since the connect function can be used to connect other socket types, and must therefore know what socket-address type it's dealing with. If all goes well, when this function returns, a "live" socket connection will be established.

```
// connect the socket - returns 0 on success
if( connect( Hsocket, (PSOCKADDR) &sockAddr,
            sizeof( SockAddr ) ) == SOCKET_ERROR )
{
    csmmsg.Format(
        "Error connecting to Server socket: %d",
        WSAGetLastError() );
    ReportEvent( csmmsg );
    return;
}
```

Typically, a server will generate some initial response to the client upon connecting. This is always the case with the simple mail protocol, so the next step is to read this initial response from the socket connection.

If you want to see what this response looks like, szbuffer could be sent to the results list or you could insert a breakpoint and examine the results directly. Or, as an extreme, you could connect to the mail server using Telnet, and use it to send each string as is done here.

```
Check( recv( Hsocket, szbuffer, sizeof( szbuffer), 0 ),
        "recv: INIT" );
```

Continuing, following the simple mail protocol, the next step is literally to say "hello," which is done by sending HELO followed by a space, and the client's host name. Each line in this protocol is terminated by a carriage-return/line-feed pair.

```
csmmsg.Format( "HELO %s%s", m_csservername, CRLF );
Check( send( Hsocket, csmmsg, csmmsg.GetLength(), 0 ),
        "send: HELO" );
```

```
Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),
        "recv: HELO" );
```

Note that in this example we are assuming the mail server host name and the client's gateway host name are identical. However, you will frequently have a scenario where you'll want to distinguish these and simply specify the client host name.

NOTE:

It's important to be aware that in SMTP, every time you send, you need to receive back an acknowledgment. As you'll see shortly, there is one exception to this rule: When you're sending the body of the message, all lines of the message body are sent before reading back any acknowledgment from the mail server.

Note that we aren't doing anything with the data received from the server. We *could* look at it, and in a robust commercial application, particularly when using protocols that are not as lock step (not as reliant on expected command/reply pairs) as SMTP, we would want to look at and parse the result code from each response. This would allow the application to more gracefully recover from particular error conditions. However, in the case of SMTP, once we are connected, it is very unlikely that we will get an unexpected reply. And in the case of a connection error, the Check function will at least give us an indication that we have a problem.

Having sent the HELO to (and received a response from) the server, the next step is to tell the server we have a mail message and indicate the return e-mail address. This is done by sending MAIL FROM:, followed by a space, and the reply-to address in angle brackets. Again, a CRLF pair always terminates each response in SMTP, and after sending, we must read back the server's reply before sending again.

```
csmsg.Format( "MAIL FROM:<%s>%s", m_csfromaddress,
              CRLF );
Check( send( Hsocket, csmsg, csmsg.GetLength(), 0 ),
        "send: MAIL FROM" );
Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),
        "recv: MAIL FROM" );
```

Next, we send one (or more, if desired) destination e-mail address, using the form RCPT TO:, followed by a space, and the to-address enclosed in angle brackets then terminated by a CR/LF pair.

```
csmsg.Format( "RCPT TO:<%s>%s", m_cstoadress, CRLF );
Check( send( Hsocket, csmsg, csmsg.GetLength(), 0 ),
        "send: RCPT TO" );
Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),
        "recv: RCPT TO" );
```

You may send mail to as many recipients as you like this way, but remember to read the server reply after sending each recipient string. Also, if you're attempting to send to many recipients, it is possible to reach the mail server's maximum recipient count. If this happens, the server would return an error result (which you could detect by checking the result returned each time by recv).

When we're finished sending the message header information, we signal this to the server by sending DATA (again, followed by a CR/LF pair).

```
csmsg.Format( "DATA%s", CRLF );
Check( send( Hsocket, csmsg, csmsg.GetLength(), 0 ),
        "send: DATA" );
Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),
        "recv: DATA" );
```

Note that we could add more optional lines to our header, but the sender and the recipient e-mail addresses are the only parts essential to getting the message delivered.

Now it's time to send the body of the message. Since the *SendMail* demo contains the message as a single Cstring, the message can be sent in a single step:

```
Check( send( Hsocket, m_csmessage,
             m_csmessage.GetLength(), 0 ),
```



```
        "send: message" );  
    csmmsg.Format( "%sThis message sent by SendMail%s",  
        CRLF, CRLF );
```

Later, in the *SendMail2* demo, where the message is read from a text file, the file is read one line at a time and transmitted. Note, however, that in neither case is any attempt made to read back a response from the server until the entire body of the message is sent. Then, after sending the message, a "signature line" has been added:

```
    Check( send( Hsocket, csmmsg, csmmsg.GetLength(), 0 ),  
        "send: signature-line" );
```

To signal the server that body of the message is finished, a single period is sent on a line by itself and the server responds by issuing an acknowledgment reply, which must be read:

```
    csmmsg.Format( "%s.%s", CRLF, CRLF );  
    Check( send( Hsocket, csmmsg, csmmsg.GetLength(), 0 ),  
        "send: end-message" );  
    Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),  
        "recv: end_message" );
```

At this point, *SendMail* could either send another e-mail message or quit. Since there isn't anything else to send, a QUIT string is sent to the server to indicate that transmissions are finished:

```
    csmmsg.Format( "QUIT%s", CRLF);  
    Check( send( Hsocket, csmmsg, csmmsg.GetLength(), 0 ),  
        "send: QUIT" );  
    Check( recv( Hsocket, szbuffer, sizeof(szbuffer), 0 ),  
        "recv: QUIT" );
```

Next, for demo purposes, a report is sent to the results list:

```
    csmmsg.Format( "Sent email message to %s",  
        m_cstoadress );  
    ReportEvent( csmmsg );
```

And, finally, the server socket is closed and the job is finished:

```
    closesocket( Hsocket );  
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The *SendMail2* demo

A second version of the *SendMail* program—*SendMail2*—is also included in the Chapter 19 folder with the principal difference being that *SendMail2* expects command-line parameters instead of being dialog-based. As a command-driven utility, *SendMail2* could be used by batch processes or other utility functions. For example, suppose that you have a workstation or server that runs FTP or Web services and connects to the Internet via a dial-up connection. Most often, this means that the IP address will be dynamically assigned. In this case, you'll have no way of connecting to that station remotely unless you first get someone to go to the station, see what the currently assigned IP address is, and read it to you. Then, you need to repeat this procedure anytime the connection breaks. You can solve this problem by writing a batch file that looks like this:

```
rasdial MyConnection username password
netstat -r > current-ip
sendmail mailserver me@mydomain.com me@mydomain.com current-ip
```

The net effect (pun intended) of this script is to connect to the Internet, output the currently assigned IP address to a file, and e-mail this file to some account that you have access to. You can use a program scheduler to cause this script to be run at a certain time of day (or week), so that either at connect time or at specified time intervals, you can find out the station's current IP address by simply checking your latest e-mail. You could even add multiple sendmail lines if you want to notify other users of the IP address change. (Of course you'll get people's permission first, as over time this could generate a number of e-mail messages that are probably unwanted.)

TIP:

You could write a program similar to *SendMail* that looks for incoming mail from *SendMail* and retrieves it from your mail server. A program that reads e-mail would look similar to *SendMail*, except it would typically use the POP3 protocol to retrieve the e-mail.

Winsock Applications

The demos discussed in the previous sections should give you a good idea of what is required to write a

simple Winsock application. Moreover, they should also give you a basic understanding of how the higher-level TCP/IP protocols communicate information between client and server. POP3, FTP, and HTTP are quite similar in this respect, although in the case of FTP and HTTP, the order in which commands and replies occur is more loosely structured. With SMTP, you can implement sending an e-mail message by means of a very predictable series of send/receive pairs, executing in “see-saw” or lock-step fashion. An HTTP or FTP server, on the other hand, requires a more flexible approach, and would be better implemented using a state-table or an event-driven program structure.

You may find it interesting to experiment directly with these protocols by using your Windows 2000 Telnet client to send properly formatted protocol strings to an FTP, SMTP, POP3, or HTTP server and view the incoming replies generated by the server. To do this, you will need to locate and download the appropriate protocol specification from the Internet to learn what the command strings and response codes are for a particular protocol. In the Telnet Connect dialog box, type the corresponding protocol in the Port combo box (yes, many of the protocols are not listed in the combo box, but you can type in whatever you want).

Although the socket paradigm is reasonably straightforward and eases the task of communicating via TCP and UDP, it can still be quite tedious to implement the various higher-level protocols in your applications. These protocols vary in their sets of possible states, their message tokens, and behavior. Furthermore, once you have written and debugged your application, you may still need to revisit it periodically to add support for new server implementations, new additions, or changes to the protocol specification. With FTP, for example, although the message tokens are standard, the formats used by various FTP servers to return requested directory listings can vary to some degree. The recently introduced set of Win32 functions called the Internet API address these problems. We will look at these functions next.

The Internet API Functions

If you need to write FTP, Gopher, or HTTP client applications, the Internet API may be ideal for your purposes. This set of functions eliminates the need for dealing with TCP/IP implementation details or for using Winsock-related structures and functions by encapsulating the FTP, HTTP, and Gopher protocols at the task level. Rather than requiring you to open and close sockets, and listen for and accept socket connections, this API lets you manage tasks, such as uploading or downloading files, using a single function call. Moreover, the Internet API encapsulates (hides) the differences in file-listing formats used by different FTP servers, so you can search for files and directories without worrying about which type of FTP server you’re connected to.

All the Internet API functions are fully re-entrant and thread-safe, so you can use them in multi-threaded applications without any problems. Another benefit to using the Internet API is that protocol-specific behavior, which is subject to change, can be isolated from application code and replaced by updating your copy of the Wininet.DLL file (which is the DLL that supplies Internet API functionality).

One drawback to using the Internet API is that it is intended only for Internet *client* applications. Microsoft documentation states clearly that these functions do not provide the appropriate level of control at the protocol or I/O levels to be usable in server implementations. Also, because of the protocol encapsulation and the abstraction of various differences in server behavior, there is likely to be at least a small performance hit as well. But if you need to quickly develop an FTP, Gopher, or HTTP client application, the Internet API will let you get your application up and running in a hurry.

The Internet API functions are listed in the following sections, grouped in seven categories:

- Connect and disconnect functions
- File finding, reading, and writing functions
- FTP-, Gopher-, and HTTP-specific functions
- Cookie functions
- URL-related functions
- Time and date functions
- Error and status functions

NOTE:

The Internet API automatically caches Internet URLs and recently transferred data. Thus, some functions may return repeat data faster than you might normally expect, because the information does not need to be retrieved again; it is transparently copied from a buffer.

Connect and Disconnect Functions

You interact with the various Internet API functions using a special Internet handle, called HINTERNET. Unlike the Winsock socket handles or Win32 handles created using CreateFile, the HINTERNET handles are completely unusable with the normal Win32 handle I/O functions, such as ReadFile or CloseHandle. Instead, you use the Internet API functions, such as InternetReadFile or InternetCloseHandle, to read from an opened resource or to close an opened handle.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

HINTERNET handles are created in a hierarchy, or tree. A top-level handle is returned by the InternetOpen function, which you must call before using almost any of the other Internet API functions. Next, you open a connection handle to a particular site by using InternetConnect. Finally, you open one or more FTP, Gopher, or HTTP file or search handles, which can then be used in calls to InternetReadFile, InternetFindNextFile, and similar functions.

InternetCloseHandle is used to close HINTERNET handles of any type, including Internet session, connection, search, and resource handles. Also, because these handles are hierarchical, any lower-level handles will be closed automatically when you close a parent handle. Thus, you could, with a single call to InternetCloseHandle (passing the handle returned from your InternetOpen call), close all other handles in your application that were opened using the Internet API functions. However, for consistency (and good programming practice), you may prefer to keep to the habit of closing each handle specifically when you're finished with it. Table 19.13 lists the Internet API connect and disconnect functions.

TABLE 19.13: Internet API Connect and Disconnect Functions

Function	Description
InternetAttemptConnect	Attempts to connect to the Internet. Typically, this should be called prior to using the other Internet functions. It returns ERROR_SUCCESS if connection is made; otherwise, it returns a Win32 error condition. The InternetAttemptConnect function operates rather differently in Windows 2000 and in Windows 98. In the later OS, the function simply returns TRUE if an open connection to the Internet exists (or FALSE if there is no connection.) Under Windows 2000, however, the same function call will attempt to open a dialup connection if necessary. For contrast, see the InternetAutodial function (Table 19.14).
InternetCheckConnection	Allows an application to check if a connection to the Internet can be established.
InternetOpen	Internet API initialization function. This must be called prior to using the Internet API functions. (Use InternetCloseHandle when finished.)
InternetConnect	Connects to the specified FTP, Gopher, or HTTP server, and returns a handle (use InternetCloseHandle when finished)—see “The FtpGet Demo,” later in this chapter, for an example of usage.

InternetCloseHandle	Closes handles (and subhandles) created by InternetOpen, InternetConnect, FtpOpenFile, GopherOpenFile, and HttpSendRequest.

In addition to the preceding Internet connect/disconnect functions, the Win32 Internet functions include seven functions to handle modem connections. These functions are described in Table 19.14.

TABLE 19.14: Internet API AutoDial Functions

Function	Description
InternetAutodial	Initiates an unattended dial-up connection
InternetAutodialHangup	Disconnects a modem connection initiated by InternetAutodial
InternetDial	Initiates a dial-up connection
InternetHangUp	Disconnects a modem connect initiated by InternetDial
InternetGetConnectState/ InternetGetConnectStateEx	Retrieves the current state of the Internet connection
InternetGoOnline	Prompts the user for permission to initiate a dial-up connection to the given URL
InternetSetDialState	Sets the current state of the Internet connection

The documentation provided with Visual Studio 6 also offers the comment: “Detailed information on how to use these functions will be included in a future version of this documentation.”

Opening a Dial-Up Connection

In the *SendMail* demo program—since assuming that everyone has an open network connection is simply not a reasonable expectation—provisions have been made to check for an Internet connection and, if necessary, to open a connection.

```
bConnected = ( InternetAttemptConnect( 0 ) ==
                ERROR_SUCCESS );
if( bConnected )
    ReportEvent( "Found open connection to Internet" );
```

As mentioned earlier (in Table 19.13) the `InternetAttemptConnect` function is a bit flaky. Under Windows 2000, this function opens a dial-up connection if necessary. Under Windows 98, performance is less consistent and, on some tests, `InternetAttemptConnect` has attempted to open a connection, but in other cases, the function has simply reported that a connection wasn’t found.

And then there’s the `InternetGetConnectedState` function that is called with one or more of four flags and, purportedly, should report the Connection State. In actual fact, however, the results returned are not very helpful. The following block of code is included in the *SendMail* program for your experimentation, but has been commented out:

```
//=====
//      The tests following do not appear to function
//      properly as of Windows 2000, Beta 3
//=====
/*
    DWORD          dwTestFlag;
```

The following test does report correctly if an open modem (dial-up) connection has been found.

```
dwTestFlag = INTERNET_CONNECTION_MODEM;
bConnected = InternetGetConnectedState(&dwTestFlag,0);
if( bConnected )
    ReportEvent( "Found open modem connection" );
```

TIP:

The calling parameter required by `InternetGetConnectedState` is a bit odd since the flag values cannot be passed directly, as with most functions—instead, the flag or flags must be assigned to a variable and the variable passed

by address.

The following modem-busy test is unreliable. The documentation does not state what is expected here, but this call will return TRUE even when a valid Internet connection is established.

```
dwTestFlag = INTERNET_CONNECTION_MODEM_BUSY;  
bConnected = InternetGetConnectedState(&dwTestFlag,0);  
if( bConnected )  
{  
    ReportEvent( "Modem connection busy" );  
    return;      // can't make a connection, so exit  
}
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The following LAN test is also unreliable since it returns TRUE for any open Internet connection...regardless of how the connection is made.

```
dwTestFlag = INTERNET_CONNECTION_LAN;  
bConnected = InternetGetConnectedState(&dwTestFlag,0);  
if( bConnected )  
    ReportEvent( "Found LAN connection to Internet" );
```

Like the previous LAN test, the following proxy test also reports TRUE for any valid Internet connection.

```
dwTestFlag = INTERNET_CONNECTION_PROXY;  
bConnected = InternetGetConnectedState(&dwTestFlag,0);  
if( bConnected )  
    ReportEvent( "Found proxy connection to Internet" );  
*/  
//=====
```

Although the InternetGetConnectedState function is somewhat unreliable—or, at least, difficult to decide what is actually being reported—the InternetAutodial function is quite straightforward and functionally useful.

In the *SendMail* demo, provisions for a forced AutoDial are made:

```
if( m_bDialUp && ! bConnected )  
{  
    if( ! InternetAutodial(  
        INTERNET_AUTODIAL_FORCE_UNATTENDED, 0 ) )  
    {  
        ReportEvent( "Dialup connection failed" );  
        return;  
    }  
    else  
    {  
        ReportEvent( "Opened dialup connection" );  
    }  
}
```

```

        bConnected = TRUE;
    }
}

```

After all of these complications, *SendMail* also includes a simple check to ensure that a connection is open before going any further:

```

if( ! bConnected )
{
    ReportEvent( "Can't send mail without connection" );
    return;
}

```

Finally, after *SendMail* has completed the transmission, the `m_bDialUp` flag is used to determine whether *SendMail* should close the Internet connection or leave it open.

```

if( m_bDialUp )
{
    if( ! InternetAutodialHangup( 0 ) )
        ReportEvent("Could not close dialup connection");
    else
        ReportEvent("Closed dialup connection");
}

```

The test used here, of course, is not very comprehensive since we really don't have any method to determine if a connection was already opened or if *SendMail* opened the connection and, therefore, is responsible for closing it. As an alternative, however, the `InternetGetConnectedState` function could be used—before attempting to test or open any connection—to set a separate flag to identify a preexisting connection.

File Locating, Reading, and Writing Functions

Table 19.15 lists the Internet API commands for finding, reading, and writing files.

TABLE 19.15: Internet API File-Handling Functions

Function	Description
<code>InternetFindNextFile</code>	Continues a search started with <code>FtpFindFirstFile</code> or <code>GopherFindFirstFile</code> and accepts a search handle created by either of these functions.
<code>InternetLockRequestFile</code>	Allows user to place a lock on a file in use.
<code>InternetQueryDataAvailable</code>	Queries the amount of data available from an HTTP, a Gopher, or an FTP resource.
<code>InternetReadFile/InternetReadFileEx</code>	Reads data from a handle opened by <code>InternetOpenUrl</code> , <code>FtpOpenFile</code> , <code>GopherOpenFile</code> , or <code>HttpOpenRequest</code> . (See also <code>FtpGetFile</code> and <code>FtpPutFile</code> .)
<code>InternetWriteFile</code>	Writes data to an open file.
<code>InternetSetFilePointer</code>	Sets the file position for the next <code>InternetReadFile</code> call. It returns the current file position if successful; otherwise, it returns a -1. Note that some servers and URL types do not support random access.
<code>InternetUnlockRequestFile</code>	Unlocks a file locked by <code>InternetLockRequestFile</code> .

FTP-, Gopher-, and HTTP-Specific Functions

Finding an FTP or Gopher file on a given host connection is very similar to finding a file on a local or network drive using the `FindFirst` and `FindNext` functions. Start searching an FTP or Gopher directory with a call to `FtpFindFirstFile` or `GopherFindFirstFile`, followed by repeated calls to `InternetFindNextFile`, re-cursing as necessary until you locate the file (or exhaust the available search directories). Remember to close the search handles returned by each `FtpFindFirst` or `GopherFindFirst` call. Table 19.16 lists the Internet API FTP-, Gopher-, and HTTP-related functions.

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

TABLE 19.16: Internet API FTP, Gopher, and HTTP Functions

Function	Description
FtpCommand	Sends commands directly to FTP servers.
FtpFindFirstFile	Starts file enumeration or searching in the current server directory. Use this with InternetFindNextFile and InternetCloseHandle.
FtpGetFile	Retrieves an entire file from the server—see “The <i>FtpGet</i> Demo,” later in this chapter.
FtpGetFileSize	Retrieves the size of an FTP resource.
FtpPutFile	Writes an entire file to the server.
FtpDeleteFile	Deletes a file on the server.
FtpRenameFile	Renames a file on the server.
FtpOpenFile	Opens a file on the server for reading or writing. (Use with InternetQueryDataAvailable, InternetReadFile, InternetWriteFile, and InternetCloseHandle.)
FtpCreateDirectory	Creates a new directory on the server.
FtpRemoveDirectory	Deletes a directory on the server.
FtpSetCurrentDirectory	Changes the connection’s current directory on the server.
FtpGetCurrentDirectory	Returns the connection’s current directory on the server.
GopherFindFirstFile	Starts enumerating a Gopher directory listing. (Use with InternetFindNextFile and InternetCloseHandle.)
GopherGetLocatorType	Parses a Gopher locator, returning its attributes.
GopherOpenFile	Opens a Gopher object on the server for retrieval. (Use with InternetQueryDataAvailable, InternetReadFile, InternetWriteFile, and InternetCloseHandle.)
GopherCreateLocator	Forms a Gopher locator for use in other Gopher function calls.
GopherGetAttribute	Retrieves attribute information on the Gopher object.
HttpEndRequest	Ends an HTTP request initiated by HttpSendRequestEx.

HttpOpenRequest	Opens an HTTP request and returns a request handle. (Use InternetCloseHandle to close.)
HttpAddRequestHeaders	Adds HTTP request headers to the HTTP request handle.
HttpSendRequest/HttpSendRequestEx	Sends the specified request to the HTTP server. (Use with InternetQueryDataAvailable and InternetReadFile.)
HttpQueryInfo	Retrieves information about an HTTP request.

Cookie Functions

The two functions listed in Table 19.17 deal with setting and retrieving cookie values. At first glance, this might seem to imply that the Internet API is being expanded to support writing server applications; however, these cookie functions deal only with the cookie database that is located on the client station. You could use these to set or retrieve cookie values that are in memory or have been saved to disk. However, for now, you cannot use them to access cookie data stored on a host or peer station.

TABLE 19.17: Internet API Cookie Functions

Function	Description
InternetGetCookie	Fills the supplied buffer with the cookie data for the specified URL and all of its parent URLs. Intended for interacting with the local cookie database.
InternetSetCookie	Sets cookie data for a given cookie name at a specified URL. Intended for interacting with local cookie database.

NOTE:

Cookies are handy strings of information stored by Web browsers on a user's local hard drive (or at least in RAM), whose values are set and then later retrieved by specific Web sites. This permits persistent storage of information between visits to that Web site. The term *cookie database* refers to the file(s), initially used by Web browsers (but now also by the Internet API) to save cookie values to disk between sessions. Netscape (and other browsers) use their own techniques and file(s) for saving cookie values. Netscape currently stores its cookies in a file called *Cookies.txt*. Internet Explorer and the Internet API currently store their cookies as separate files, in a directory called *\Windows\Cookies*.

URL-Related Functions

The URL-related functions are useful for creating, splitting, and combining URLs. These functions are handy for converting between a URL and its component pieces or for converting between relative and absolute URLs. Table 19.18 lists the Internet API functions associated with URLs.

TABLE 19.18: Internet API URL-Related Functions

Function	Description
InternetCrackUrl	Parses a URL string into components.
InternetCreateUrl	Creates a URL string from components.
InternetCanonicalizeUrl	Converts a URL to canonical form. This is used to convert a URL into a server-safe URL, because it converts to escape sequences any spaces or other characters that could be misinterpreted by specific servers.
InternetCombineUrl	Combines base and relative URLs.
InternetOpenUrl	Begins retrieving an FTP, a Gopher, or an HTTP URL.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Time/Date Conversion Functions

The Internet API also supplies functions for converting back and forth between (local) system time and HTTP Internet time strings. HTTP time/date strings are used, for example, in the “Expires:” optional header setting embedded in some HTML documents. The two time/date conversion commands are listed in Table 19.19.

TABLE 19.19: Internet API Time/Date Conversion Functions

Function	Description
InternetTimeFromSystemTime	Converts a Win32 SYSTEMTIME structure into a string, formatted according to the HTTP RFC 1.0 time/date string format.
InternetTimeToSystemTime	Converts an HTTP time/date string to a SYSTEMTIME structure.

Error and Status Functions

You can perform error checking by first checking the Boolean return value for each function, and then using the standard GetLastError Win32 API function to return a specific error code. More specific information can be retrieved by calling InternetGetLastResponseInfo, although in some cases, this tends to return more information than you need. You can also display Internet-specific information using the InternetErrorDlg function. To control protocol-specific settings, use InternetQueryOption and InternetSetOption. Table 19.20 lists the Internet API error and status functions.

TABLE 19.20: Internet API Error and Status Functions

Function	Description
InternetSetStatusCallback	Assigns a callback function to be called with status information. This is useful for monitoring functions that take some time to complete.
InternetErrorDlg	Displays predefined dialog boxes for common Internet error conditions.
InternetConfirmZoneCrossing	Detects when your application is moving either from a secure to an insecure URL, or vice versa.

InternetQueryOption	Queries the current setting of a given Internet option.
InternetSetOption	Sets an Internet option.
InternetGetLastResponseInfo	Returns a buffer containing data from the last reply; typically used to retrieve more detailed or specialized status or error information—see “The <i>FtpGet</i> Demo” later in this chapter.

MFC Wrapper Classes

The Microsoft Foundation Classes provide wrapper classes for the WinInet (Win32 Internet) functions and ActiveX technologies that can be used to make Internet programming easier. These are shown in Table 19.21.

TABLE 19.21: Internet Support Classes

Function	Usage
CInternetSession	Creates and initializes a single Internet session or several simultaneous sessions—optionally, describes the connection to a proxy server.
CInternetConnection	Manages a connection to an Internet server.
CInternetFile	The CInternetFile and derived classes provide access to files on remote systems supporting Internet protocols.
CFileFind	Performs both local and Internet file searches.
CInternetException	Represents an exception condition related to an Internet operation.
FTP Services	
CFtpConnection	Manages a connection to an FTP server.
CFtpFileFind	Aids in Internet file searches on FTP servers.
Gopher Services	
CGopherConnection	Manages a connection to a gopher server.
CGopherFile	Provides the functionality to find and to read files from a gopher server.
CGopherFileFind	Aids in Internet file searches on gopher servers.
CGopherLocator	Retrieves a gopher “locator” from a gopher server, determines the locator’s type, and makes the locator available to CGopherFileFind.
HTTP Services	
CHttpConnection	Manages a connection to an HTTP server.
CHttpFile	Provides the functionality to find and to read files from an HTTP server.

The *FtpGet* Demo: Simple FTP Transfer

The *FtpGet* demo uses the Internet API to perform a simple FTP transfer. As you’ll see, in comparison to implementing the FTP protocol directly in your application using Winsock calls, the Internet API is really a breeze to use.

The *FtpGet* demo downloads a given FTP file using the Internet API functions. It starts by calling `InternetOpen` to open an Internet session, then connects to the desired FTP server using `InternetConnect`. If a path has been appended to the specified filename, this directory is made the current directory on the FTP server, and then the file is downloaded to the client computer using `FtpGetFile`.

NOTE:

The *FtpGet* demo is included on the CD that accompanies this book, in the Chapter 19 folder.

Like the previous two demos, the *FtpGet* demo is a dialog-based application. In this case, however, the *FtpGet* project has two requirements that weren’t needed in the previous examples. First, the `FtpGetDlg.cpp` file needs to include the header file `wininet.h`, the Internet API header file.

```
#include <wininet.h>
```

The second requirement is to go to the Project ? Settings ? Link tab and, under Object/library modules, add wininet.lib.

The *FtpGet* program—see Figure 19.4—offers fields for the four data items necessary for an FTP download.



FIGURE 19.4 The *FtpGet* demo

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Taking these fields slightly out of order, the Domain name, of course, is the name of the FTP server where the request file is located, while the File name field is expected to contain both the directory and file information.

The first two fields—Username and Password—are self-identifying but do deserve some comment. For an FTP server that permits an anonymous login, the Username could be set to “anonymous” while the Password field should contain your e-mail address.

For a non-anonymous login, the Username would simply contain your username and the Password field your password. Note that, in this demo, the Password edit box does not have the “password” flag set—that is, the password will appear on screen exactly as it is typed rather than the characters being replaced by asterisks.

The *FtpGet* demo includes a *LastErrorMsg* function that returns a string (CString) containing the error message or status information returned from the last Internet API function call. The information is retrieved using a call to *InternetGetLastResponseInfo* and passing a pointer to the error code, a buffer to fill, and the size of the message buffer. In the event *InternetGetLastResponse* fails (as would happen if there were no *Wininet.DLL*, for instance), the *Win32 GetLastError* function is called to retrieve whatever *Win32* error code might be available.

```
CString CFTPGetDlg::LastErrorMsg()
{
    CString      csErrorMsg;
    static char  szErrorMsg[4096];
    ULONG       dwErrorNum, dwLength;

    dwLength = sizeof( szErrorMsg );
    if( ! InternetGetLastResponseInfo( &dwErrorNum,
                                       (LPSTR) szErrorMsg,
                                       &dwLength ) )
        csErrorMsg.Format(
            "Unable to get error information, Error# = %d ",
            GetLastError() );
    else
        csErrorMsg.Format( "%s", szErrorMsg );
    return csErrorMsg;
}
```

```
}
```

The information retrieved by `LastErrorMsg`—if any—will be displayed in the Status list shown in Figure 19.5.

Like the *SendMail* demo, *FtpGet* begins in the `OnOK` function by declaring the necessary variables.

```
void CFTPGetDlg::OnOK()
{
    HINTERNET hInternet, hHost;      // handles
    int        nPos;
    CString    csPath, csTempName, csErr;
```

The first two variables are two `HINTERNET` handles. These are “opaque” Internet API handle types, which are allocated by Internet API functions and are the only handle types suitable for use with the Internet API functions that take a handle as a parameter. (They are “opaque” in that there is no useful information directly accessible inside these handles, and they can be used only with Internet API functions.)

NOTE:

Note that `HINTERNET` handles actually include four types: an “Internet” handle, a session handle, a search handle, and a URL/file handle.

The `csPath` and `csTempName` will contain the path and file, once these have been split apart from the input path/filename supplied in the dialog box.

The next task is resetting the status list and retrieving the data from the edit fields before running the usual tests on the retrieved data.

```
m_lbStatus.ResetContent();
UpdateData( TRUE );
if( m_csUserName.IsEmpty() )
{
    ReportError( "User name required" );
    return;
}
if( m_csPassword.IsEmpty() )
{
    ReportError( "Password required" );
    return;
}
if( m_csFileName.IsEmpty() )
{
    ReportError( "Filename required" );
    return;
}
if( m_csDomainName.IsEmpty() )
{
    ReportError( "Domain name required" );
    return;
}
```

Next, since both the forward-slash (/) and backward-slash (\) characters are accepted in Internet path specifications, a little work is need here to search the supplied filename to separate the path specification from the filename proper. Since we need to search for both characters, the `FindOneOf` function is used and the process steps through the compound path/file specification until the last matching character is found.

```
    // get the path, if prepended to filename
    csTempName = m_csFileName;
    do
    {
        nPos = csTempName.FindOneOf( "/\\" );
```

```

        if( nPos >= 0 )
        {
            csPath += csTempName.Left( nPos + 1 );
            csTempName =
                csTempName.Right(
                    csTempName.GetLength() - ( nPos + 1 ) );
        }
    }
    while( nPos >= 0 );

```

Note that we do not need to worry about converting forward slashes to backward slashes (or vice versa). This is because the Internet API functions that take filenames know how to convert the slashes and to the form expected by the server.

Now, we initialize the Internet API for this application by calling `InternetOpen`. The second parameter tells WinInet that we will need a proxy server to connect. If the handle returned is zero, an error occurred and we display an error message and exit.

```

hInternet = InternetOpen( "FTPGet",
                        LOCAL_INTERNET_ACCESS,
                        NULL, 0, 0 );

if( ! hInternet )
{
    csErr.Format( "Error opening Internet:\r\n%s",
                GetLastErrorMsg() );
    InternetCloseHandle( hInternet );
    ReportError( csErr );
    return;
}

```

NOTE:

A *proxy server* is software that permits networked computers to access the Internet via a trusted server. The proxy server usually works with some level of firewall functionality and sometimes even using network protocols other than TCP/IP, with the appropriate proxy client software.

Having called `InternetOpen`, we are now ready to begin using the other Internet API functions. We first make a call to `InternetConnect`, which establishes a connection to the specified FTP host, using our supplied username and password. If an error occurs at this point, we display an error message and exit.

```

hHost = InternetConnect( hInternet, m_csDomainName,
                        INTERNET_INVALID_PORT_NUMBER,
                        m_csUserName, m_csPassword,
                        INTERNET_SERVICE_FTP,
                        INTERNET_FLAG_PASSIVE , 0);

if( ! hHost )
{
    csErr.Format( "Error connecting to %s:\r\n%s",
                m_csDomainName, GetLastErrorMsg() );
    InternetCloseHandle( hInternet );
    ReportError( csErr );
    return;
}

```

In the call to `InternetConnect`, the `hInternet` and the domain name arguments should be self-explanatory. The third argument, however, may look more like an error flag but is actually the number of the TCP/IP port on the server that will be used for the connection. Valid flags are shown in Table 19.22. Note that the flags set only the port to be used while the service is set by the value of the sixth argument (*dwService*), discussed in the following sections.

TABLE 19.22: Port Arguments for `InternetConnect`

Function	Description
INTERNET_DEFAULT_FTP_PORT	Uses the default port for FTP servers (port 21)
INTERNET_DEFAULT_GOPHER_PORT	Uses the default port for Gopher servers (port 70)
INTERNET_DEFAULT_HTTP_PORT	Uses the default port for HTTP servers (port 80)
INTERNET_DEFAULT_HTTPS_PORT	Uses the default port for HTTPS servers (port 443)
INTERNET_DEFAULT SOCKS_PORT	Uses the default port for SOCKS firewall servers (port 1080)
INTERNET_INVALID_PORT_NUMBER	Uses the default port for the service specified by <i>dwService</i>

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The `m_csUserName` (`lpszUsername`) argument is the address of a null-terminated string containing the name of the user to log on. If this parameter is NULL, the function uses an appropriate default, except for HTTP; a NULL parameter in HTTP causes the server to return an error. For the FTP protocol, the default is “anonymous”.

The `m_csPassword` (`lpszPassword`) argument is the address of a null-terminated string that contains the password to use to log on. If both `lpszPassword` and `lpszUsername` are NULL, the function uses the default “anonymous” password. In the case of FTP, the default password is the user’s e-mail name. If `lpszPassword` is NULL, but `lpszUsername` is not NULL, the function uses a blank password. Table 19.23 describes the behavior for the four possible settings of `lpszUsername` and `lpszPassword`.

TABLE 19.23: Username and Password Options

<code>lpszUsername</code>	<code>lpszPassword</code>	User name sent to FTP server	Password sent to FTP server
NULL	NULL	“anonymous”	User’s e-mail name
Non-NULL string	NULL	<code>lpszUsername</code>	User’s e-mail name
NULL	Non-NULL string	ERROR	ERROR
Non-NULL string	Non-NULL string	<code>lpszUsername</code>	<code>lpszPassword</code>

The sixth argument sets the type of service and can be one of three values:

<code>INTERNET_SERVICE_FTP</code>	FTP service
<code>INTERNET_SERVICE_GOPHER</code>	Gopher service
<code>INTERNET_SERVICE_HTTP</code>	HTTP service

In this example, because the objective is an FTP connection, the flag `INTERNET_SERVICE_FTP` has been used to designate the service type.

The seventh argument is a flag specific to the type of service. Therefore, with `INTERNET_SERVICE_FTP`, the flag value `INTERNET_FLAG_PASSIVE` causes the application to use passive FTP semantics.

Finally, the eighth argument (`dwContext`) is an application-defined value that is used to identify the application

context for the returned handle in callbacks. In this case, no callback is used and the argument is passed as 0 (or NULL).

Having established a connection to the host (hHost), the next step is to set the current directory on the remote server to point to the directory where the desired file is located.

```
if( ! FtpSetCurrentDirectory( hHost, csPath ) )
{
    csErr.Format( "Unable to find remote directory : %s",
                  csPath );
    ReportError( csErr );
    ReportError( GetLastErrorMsg() );
    InternetCloseHandle( hInternet );
    return;
}
```

If you were using an FTP program such as *WSFTP* or *CuteFTP*, provisions are included to display the directory/file structure of the remote computer and to change the selected directory interactively. Here, however, the remote directory must be specified in the supplied file request and provisions are included to report if the directory is not found. Once the correct directory has been found, *FtpGetFile* is called:

```
if( ! FtpGetFile( hHost, csTempName, csTempName, FALSE,
                  INTERNET_FLAG_RELOAD,
                  FTP_TRANSFER_TYPE_BINARY, 0 ) )
{
    csErr.Format( "Unable to transfer file : %s",
                  csTempName );
    ReportError( csErr );
    ReportError( GetLastErrorMsg() );
    InternetCloseHandle( hInternet );
    return;
}
```

The *FtpGetFile* function is called with the handle to the host (hHost) followed by the name of the file to retrieve and the name the file will be saved under locally. Note that *FtpGetFile* will allow you to include a path name on either (or both) the source and target filenames. If we didn't need to split the filename off for the destination directory, it would have been simpler just to let *FtpGetFile* handle the path name internally.

The fourth argument—*fFailIfExists*—is a Boolean flag indicating whether to proceed (overwrite) if a local file with the specified name already exists. Since we have specified *FALSE*, any local file will simply be overwritten.

The fifth argument—*INTERNET_FLAG_RELOAD*—forces a download of the requested file, object, or directory listing from the origin server, not from the cache (if any).

The sixth argument—*INTERNET_FLAG_TRANSFER_ASCII* in the demo program—is a flag controlling how the file download will be handled. Two sets of flags can be used with the first series of flag values (see Table 19.24) controlling the transfer conditions while a second series of flags (see Table 19.25) control caching.

TABLE 19.24: File Download Flags

Flag	Description
FTP_TRANSFER_TYPE_ASCII	Transfers the file using FTP's ASCII (Type A) transfer method. Control and formatting information is converted to local equivalents.
FTP_TRANSFER_TYPE_BINARY	Transfers the file using FTP's Image (Type I) transfer method. The file is transferred exactly as it exists with no changes. (This is the default transfer method.)
FTP_TRANSFER_TYPE_UNKNOWN	Defaults to FTP_TRANSFER_TYPE_BINARY.
INTERNET_FLAG_TRANSFER_ASCII	Transfers the file as ASCII.
INTERNET_FLAG_TRANSFER_BINARY	Transfers the file as binary.

Unless you are definitely transferring a text (ASCII) file, be sure to use FTP_TRANSFER_TYPE_BINARY to perform a binary file transfer; otherwise, your file will almost certainly fail to transfer the file correctly.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

One or more of the file caching flags shown in Table 19.25 can be combined with the transfer type flag.

TABLE 19.25: File Caching Flags

Flag	Description
INTERNET_FLAG_DONT_CACHE	Does not add the returned entity to the cache. Identical to the preferred value, INTERNET_FLAG_NO_CACHE_WRITE.
INTERNET_FLAG_HYPERLINK	Forces a reload if there was no Expires time and no Last-Modified time returned from the server when determining whether to reload the item from the network.
INTERNET_FLAG_MAKE_PERSISTENT	No longer supported.
INTERNET_FLAG_MUST_CACHE_REQUEST	Causes a temporary file to be created if the file cannot be cached. Identical to the preferred value, INTERNET_FLAG_NEED_FILE.
INTERNET_FLAG_NEED_FILE	Causes a temporary file to be created if the file cannot be cached.
INTERNET_FLAG_NO_CACHE_WRITE	Does not add the returned entity to the cache.
INTERNET_FLAG_RELOAD	Forces a download of the requested file, object, or directory listing from the origin server, not from the cache.
INTERNET_FLAG_RESYNCHRONIZE	Causes the FTP resource to be reloaded from the server.

The final argument used in the FtpGetFile call is an application-defined value that is used to identify the application context for the returned handle in callbacks. Again, in this demo, no callback is used and the argument is passed as 0 (or NULL).

Once the transfer is complete, all that remains (for demo purposes) is to report completion, and then to clean up by closing the two handles—the first to the host (FTP server), and the second to the Internet connection.

```
ReportError( "File transferred" );
// finished, clean up and exit
InternetCloseHandle( hHost );
InternetCloseHandle( hInternet );
```

}

NOTE:

It is possible to set a callback function using the `InternetSetStatusCallback` function, to be called periodically when Internet API functions that take an extended period of time are busy executing. This would be a nice addition if you need to transfer larger files.

Internet Applications

As you can see from the *FtpGet* demo, performing an FTP download using the Internet API is straightforward and quite trivial to implement.

One nice feature you could add to the *FtpGet* demo is a searching feature, so that the user would not need to specify the file path on the remote server. This would not be difficult to add; you could use the `FtpFindFirst` and `InternetFindNext` functions to recursively search each directory for the desired file. Another handy use for these find-first and find-next functions would be to implement a utility that transfers an entire FTP (or Gopher or Web) site from one server to another, by re-cursing through all directories and copying all files at each level from the host to the client.

Also, note that while the demo makes use of the `FtpGetFile` function, you could instead, almost as easily, use the `FtpOpenFile` function, followed by repeated calls to `InternetReadFile`, to read successive blocks from the server. This requires slightly more code but gives you the opportunity to calculate and display a progress indicator as the data is transferring.

Alternatively, instead of `FtpOpenFile`, you could use `InternetOpenUrl`, which has the added benefit of being protocol-independent. `InternetOpenUrl` is able to parse different URL types and automatically determine the type of server connection to open. In this way, you can use the same code to open various types of URLs. The protocol-independent data transfer is then performed by making repeated calls to `InternetReadFile`.

The Internet API is a powerful yet easy-to-use API, and indications are that support for additional protocols is being planned for future versions. Perhaps by the time you read this, there will also be functions for sending and retrieving e-mail. The Internet API has all the earmarks of becoming a widely used Windows API.

ActiveX Web Controls

While e-mail and FTP are important processes, the most popular Web operation is being able to access and view Web sites. Game and X-rated materials aside, there are still a myriad of reasons for accessing Web sites and, as well, a wide variety of reasons for providing web (or HTML) access from within an application.

NOTE:

One very good reason for supplying HTML access is discussed in my book *Developing Windows Error Messages* (published by O'Reilly) where HTML text is used for presenting error and correction information. At the same time, Internet access could be used to accumulate error reports for corrective analysis.

And, while you could, in a masochistic frenzy, sit down and write your own HTML interpreter/viewer utility to provide a Web view for your application, doing so would be rather akin to reinventing the wheel while sitting in a warehouse full of wheels.

The reason I refer to this as an exercise in futility is that you already have a full-featured HTML viewer and Internet access utility at your fingertips. This utility can be incorporated into your application—and under your application's control—with a minimum of effort and a maximum of effect.

The *WebView* Demo: A Simple Web Browser

The *WebView* demo shows how to use Microsoft's Internet Explorer—as an ActiveX (OCX) control—to create a simple but functional Web browser within an application. For simplicity, as with the previous Internet demos, we'll use a dialog-based application.

In this example, the simple way to create an ActiveX control is to insert it into a dialog and, in your own applications, this is probably the route you would choose for using an ActiveX control.

NOTE:

Adding an ActiveX control is essentially a three-step process, as illustrated in Figure 19.5.

1. After creating the dialog, right-click within the dialog box to display the pop-up menu, then select Insert ActiveX Control.
2. From the Insert ActiveX Control dialog box, scroll down and select the Microsoft Web Browser from the list box, then click OK.
3. Position the ActiveX control (and resize as desired) within the dialog.

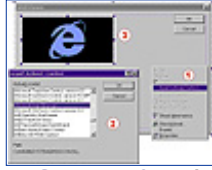


FIGURE 19.5 Adding an ActiveX Web Browser control

At this point, the dialog contains an ActiveX Web browser and two buttons (OK and Cancel), but this is just the bare bones. To make this useful, we need to add a few features—at a bare minimum, we need some method of passing a URL to the Web browser to initiate an Internet link.

In the *WebView* demo, we'll begin by providing a combo list box—so that a URL address can be entered directly and so that a list of locations can be stored—then add three buttons: Forward, Back, and Go. (The original OK and Exit buttons are superfluous and can be discarded.)

Next, using the ClassWizard, we create three member variables—one for the Web browser control and two for the combo list box:

IDC_EXPLORER1	CWebBrowser	m_cWebBrowser
IDC_URL_LIST	CString	m_csURL
IDC_URL_LIST	CComboBox	m_cbURLList

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The reason for creating two member variables for the combo list box is that one—the CString member—will reference the edit field while the CComboBox member allows access to the list box portion.

Also, using the ClassWizard, we need to create BN_CLICKED member functions for the three buttons and a CBN_SELCHANGE member function for the list box. For the Web browser, however, one more member function is needed for the NavigateComplete2 message.

Now, before writing code to implement these functions, a declaration is needed in the WebViewerDlg.h header as shown in bold:

protected:

```
HICON          m_hIcon;  
COleVariant   noArg;
```

The COleVariant variable noArg will be used as the equivalent of a NULL argument in several calls to the CWebBrowser::Navigate function.

Initializing the Web Browser

For demonstration purposes, the WebViewer is initialized with a link to the Sybex Web site (see Figure 19.6). In the OnInitDialog procedure, four lines of code initialize the link, add the Sybex site to the combo list box, and initialize the list.

```
m_csURL = "www.sybex.com";  
m_cWebBrowser.Navigate( m_csURL, &noArg, &noArg,  
                        &noArg, &noArg );  
m_cbURLList.SetCurSel( 0 );  
UpdateData( FALSE );
```

Notice that the Navigate function is called with the URL as the first argument while the remaining four arguments are noArg, meaning that we have no further arguments for this function.



FIGURE 19.6 WebViewer at work

NOTE:

The CWebBrowser class—and therefore the Navigate function—is not documented in the online help, but you should be able to refer to the CHtmlView class for hints. You will also find some documentation through the MSDN site.

In other circumstances, the dialog might be called with a link to a specific site, with a particular .HTML file, or even with a location within a file. And, of course, the combo list box and navigation buttons are purely optional. Depending on how you are using the Web browser—and this can be a purely local operation, requiring neither network nor Internet—you could pass the reference as <file:///directory/myfile.html>.

In any case, for demo purposes, the link is to an Internet site and (forgive mention of the obvious) you will need to have either network or dial-up access before the link can be completed.

The OnNavigateComplete2Explorer1 function is invoked automatically whenever a link is followed from an existing page.

```
void CWebViewerDlg::OnNavigateComplete2Explorer1(
    LPDISPATCH pDisp,
    VARIANT FAR* URL )
{
    m_csURL = m_cWebBrowser.GetLocationURL();
    if( m_cbURLList.FindStringExact( -1, m_csURL )
        == CB_ERR )
        m_cbURLList.SetCurSel(
            m_cbURLList.AddString( m_csURL ) );
    UpdateData( FALSE );
}
```

The implementation shown ensures that the followed links are added to the list maintained in the combo list box. The initial test—which looks for an existing match in the list—prevents duplicate entries. Again, depending on your usage, this function may not be needed by your application.

The OnSelchangeUrlList function is also optional and simply recognizes a selection from the list box and calls the Navigate function to surf to the selected site (or file, etc.).

```
void CWebViewerDlg::OnSelchangeUrlList()
{
    UpdateData( TRUE );
    m_cbURLList.GetLBText( m_cbURLList.GetCurSel(),
        m_csURL );
    m_cWebBrowser.Navigate( m_csURL, &amp;noArg, &noArg,
        &noArg, &noArg );
}
```

The OnGo button is similar to the OnSelchangeUrlList function except that it simply reads the edit box from the combo list box and uses that entry as the URL.

```
void CWebViewerDlg::OnGo()
{
    UpdateData( TRUE );
    m_cWebBrowser.Navigate( m_csURL, &noArg, &noArg,
        &noArg, &noArg );
}
```

The final provisions are the OnForward and OnBack functions that, respectively, call the CWebBrowser's

GoForward and GoBack functions.

```
void CWebViewerDlg::OnForward()  
{  
    m_cWebBrowser.GoForward();  
}  
void CWebViewerDlg::OnBack()  
{  
    m_cWebBrowser.GoBack();  
}
```

WARNING:

The GoForward and GoBack functions are actually a bit of an annoyance. While each works well enough if there is a forward or backward link, when there is no link to follow, a rather annoyingly blank error message is displayed. To improve on this, refer to the *DWebBrowserEvents2* class and the `CommandStateChange` function.

Summary

We've covered the Internet (and a lot of intranet operations) pretty thoroughly, discussing e-mail, FTP, and Web browsing. While the examples used have been relatively simple, these have also provided coverage of most of the major operations and have provided the basis for further refinements according to your needs and interests.

In the next chapter, we will look further into network operations, but this time, at LAN and WAN operations.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 20

Network Programming

- NetBIOS support implementation
- WNet API functions
- LANMan functions
- Winsock 2 for networking applications
- Named pipes and mailslots for networking
- An introduction to RPC and DCOM

As Windows has grown up, it has taken a variety of approaches toward network solutions. The good news is that much progress has been made. The not-so-good news is that it is not always obvious which APIs or technologies are best suited for a particular purpose.

Another unfortunate aspect to Windows network programming is that full functionality is supported only on Windows NT/2000 (Workstation and Server versions) and many of the APIs are not fully supported by Windows 98. This applies in particular to the LANMan functions and, to some degree, to RPC (Remote Procedure Calls), pipes, and mailslots. Mailslots, for example, ignore security attributes under Windows 98; named pipes work only on the client side in Windows 98; and RPC is supported on Windows 98, but with fewer underlying protocol choices than are available on Windows NT/2000. If you need networking support and your application must run on Windows 98 (or earlier), your best bet is to use the Winsock API or DCOM/ActiveX controls. However, you may find that for certain networking projects, you will be better served by using Windows NT/2000 (excuse the weak pun).

In this chapter, we'll look at the various networking APIs and their relative merits. Then we'll go over some examples of using the Winsock API, pipes, mailslots, and RPC.

The NetBIOS Functions

NetBIOS has the distinction of being the only networking API function that was built into DOS. For

programmers faced with porting old applications (which made use of NetBIOS DOS interrupt calls) to Win32, the Netbios API function can simplify the conversion process. Instead of calling the NetBIOS interrupt, you fill a Network Control Block (NCB) structure and pass a pointer to it in your call to the Netbios API.

However, except for the porting scenario, it is unlikely that you will want to use Netbios itself in your applications since a richer set of functions is available via named pipes, RPC, WNet functions, or Winsock, all of which are explained in this chapter. For applications that will never need anything but NetBIOS support, and for programmers already very familiar with it, the Netbios API provides an efficient way to implement NetBIOS support.

Here's how the Netbios function is defined:

```
UCHAR Netbios
(
    PNCB pncb    // address of network control block
);
```

The NCB structure is defined as follows:

```
typedef struct _NCB    // ncb
{
    UCHAR  ncb_command;
    UCHAR  ncb_retcode;
    UCHAR  ncb_lsn;
    UCHAR  ncb_num;
    PCHAR  ncb_buffer;
    WORD   ncb_length;
    UCHAR  ncb_callname[NCBNAMSZ];
    UCHAR  ncb_name[NCBNAMSZ];
    UCHAR  ncb_rto;
    UCHAR  ncb_sto;
    void (*ncb_post) (struct _NCB *);
    UCHAR  ncb_lana_num;
    UCHAR  ncb_cmd_cplt;
    UCHAR  ncb_reserve[10];
    HANDLE ncb_event;
} NCB;
```

The `ncb_command` field must always be set to one of the Netbios function values, which are listed in Table 20.1. `ncb_length` must always be set to the size of the NCB structure, and certain other fields must be set as appropriate for each function. Upon completion, Netbios returns the status code in `ncb_retcode`.

TABLE 20.1: NetBIOS Functions

Command	Description
NCBACTION	Enables extensions to the transport interface.
NCBADDGRNAME	Adds a group name to the local name table.
NCBADDNAME	Adds a unique name to the local name table.
NCBASTAT	Retrieves the status of the adapter.
NCBCALL	Opens a session with another name.
NCBCANCEL	Cancels a previous command.
NCBCHAINSEND	Sends the contents of two data buffers to the specified session partner.
NCBCHAINSENDNA	Sends the contents of two data buffers to the specified session partner without waiting for acknowledgment.
NCBDELNAME	Deletes a name from the local name table.
NCBDGRECV	Receives a datagram from any name.
NCBDGRECVBC	Receives a broadcast datagram from any host.
NCBDGSEND	Sends a datagram to a specified name.
NCBDGSENDBC	Sends a broadcast datagram to every host on the LAN.

NCBENUM	Enumerates LAN adapter (LANA) numbers.
NCBFINDNAME	Determines the location of a name on the network.
NCBHANGUP	Closes a specified session.
NCBLANSTALERT	Notifies the user of LAN failures that last for more than one minute.
NCBLISTEN	Enables a session to be opened with another name.
NCBRECV	Receives data from the specified session partner.
NCBRECVANY	Receives data from any session corresponding to a specified name.
NCBRESET	Resets a LANA adapter.
NCBSEND	Sends data to the specified session partner.
NCBSENDNA	Sends data to specified session partner and does not wait for an acknowledgment.
NCBSSTAT	Retrieves the status of the session.
NCBTRACE	Activates or deactivates NCB tracing. Support for this command is optional and system-specific.
NCBUNLINK	Unlinks the adapter.

A few additions and enhancements have been made to the Netbios function for better integration into Windows programs. Functions can be executed asynchronously either by posting (using `ncb_post`) or by using signal-completion events (using `ncb_event`).

In the posting approach, the function specified in `ncb_post` takes only a single parameter, which (like Netbios) is a pointer to an NCB. This makes your implementation much more consistent and portable across different versions (or platforms) of Windows.

In the events approach, the `ncb_event` is set to the non-signaled state when an asynchronous NetBIOS command is requested, and is set to the signaled state when the command has completed.

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

The WNet API

The WNet functions, listed in Table 20.2, first appeared in Windows for Workgroups, primarily as a higher-level API. Occasionally, these functions, like the LANMan functions, provide an easier API for performing protocol-independent simple networking tasks. However, most (if not all) of the functionality is duplicated (or expanded) in the other Win32 network APIs.

TABLE 20.2: WNet Functions

Function	Description
WNetAddConnection	Connects a local device to a network resource. This creates a persistent connection if successful. This command has been updated to WNetAddConnection2 and WNetAddConnection3.
WNetAddConnection2	More recent version that replaces WNetAddConnection.
WNetAddConnection3	Similar to WNetAddConnection2, but it also takes a handle to a window that the network provider can use as an owner window for dialog boxes.
WNetCancelConnection	Breaks an existing network connection. This command has been updated to WNetCancelConnection2.
WNetCancelConnection2	Replaces WNetCancelConnection. It removes persistent network connections that are not currently connected.
WNetCloseEnum	Ends a network-resource enumeration started by the WNetOpenEnum function.
WNetConnectionDialog	Starts a general browsing dialog box for connecting to network resources.
WNetDisconnectDialog	Starts a general browsing dialog box for disconnecting from network resources.
WNetEnumResource	Continues a network-resource enumeration started by the WNetOpenEnum function.
WNetGetConnection	Retrieves the name of the network resource associated with a local device.
WNetGetLastError	Retrieves the most recent extended error code set by a Windows network function.
WNetGetUniversalName	Takes a drive-based path for a network resource and obtains a data structure that contains a more universal form of the name.

WNetGetUser	Retrieves the current default username or the username used to establish a network connection.
WNetOpenEnum	Starts an enumeration of network resources or existing connections.

WNet and Network Resources

In the WNet way of doing things, any network resource is represented by means of a NETRESOURCE structure, which is defined as follows:

```
typedef struct _NETRESOURCE
{
    DWORD   dwScope;
    DWORD   dwType;
    DWORD   dwDisplayType;
    DWORD   dwUsage;
    LPTSTR  lpLocalName;
    LPTSTR  lpRemoteName;
    LPTSTR  lpComment;
    LPTSTR  lpProvider;
} NETRESOURCE;
```

In its original form, the WNet API provided for display of network dialog boxes that the user could use to browse network resources, and connect or disconnect by using the mouse. Although the dialog boxes are still supported in Windows 2000 (and Windows 95/98/NT), the functions that display them seem to have been phased out. The Win32 SDK documentation discourages using them, and new functions have been added to replace them and provide a similar yet expanded functionality. Win32 programs should make use of the WNetAddConnection2 API function, as shown in this sample program, discussed in a bit. However, you can still display browsable network connect and disconnect dialog boxes, using the WNetConnectionDialog and WNetDisconnectDialog functions, respectively.

The *WNetDemo* Demo: Mapping a Drive

The *WNetDemo* example is a simple dialog-based program that uses the WNetAddConnection2 function to map a user-specified network drive to a local name.

NOTE:

The *WNetDemo* demo is included on the CD accompanying this book, in the Chapter 20 folder.

The dialog—see Figure 20.1—uses four edit boxes to request the remote drive specification, a local resource name, a user name, and a password. The list box at the bottom is used to report success if the connection is established or, alternately, to display error reports when something goes wrong.

Since the process of making a connection is relatively simple, the bulk of the process is found in the OnConnect function, which begins by declaring a variable (nr) of the type NETRESOURCE.

```
void CWNetDemoDlg::OnConnect()
{
    NETRESOURCE    nr;
```

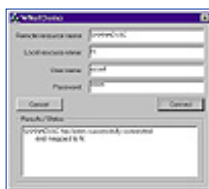


FIGURE 20.1 Mapping C: from Xanadu to N: on Tirnanog

Before doing anything further, there are a few setup provisions followed by four tests to perform to ensure that the remote resource name, local name, user name, and password are not empty strings—or, if any of these are empty, to report the error and return. However, assuming that these four entries were supplied in the dialog edit fields and it's appropriate to proceed, the first step is to set the NETRESOURCE structure to nulls:

```
memset( &amp;nr, 0, sizeof(nr) );
```

If this is not done, there appears to be a very good possibility that your system will hang irrecoverably (the reasons for this are uncertain and difficult to discover).

After filling the structure with nulls, the dwType field is assigned to RESOURCETYPE_DISK and the lpLocalName and lpRemoteName fields are pointed to the strings retrieved from the edit fields.

```
nr.dwType = RESOURCETYPE_DISK;                // required
nr.dwDisplayType = RESOURCEDISPLAYTYPE_SHARE; // optional
nr.dwScope = RESOURCE_GLOBALNET;              // optional
nr.lpLocalName = m_csLocal.GetBuffer( m_csLocal.GetLength() );
nr.lpRemoteName = m_csRemote.GetBuffer( m_csRemote.GetLength() );
```

Notice that while the dwType field specifies a disk resource, the next two assignments above are commented as “optional”. In most cases, the nr.dwDisplayType and nr.dwScope fields are unnecessary and are included in this example only for demonstration; refer to the Visual Studio online documentation for the NETRESOURCE structure for more information.

Now the call to WNetAddConnection2 can be made. Note that parameters two and three could specify another username and password. Alternately, both the password and username arguments could be set to NULL to default to using the current user’s password and name.

```
if( WNetAddConnection2( &amp;nr, m_csPW, m_csUserName, 0 ) == NO_ERROR )
```

The final argument—0 in this example—is a dword flag value that specifies connection options. At the present time, the only option defined is CONNECT_UPDATE_PROFILE, which instructs the system to remember the network resource connection and to automatically attempt to restore the connection when the user logs on again. Upon success, the connection is reported in the list box:

```
{
    CString csMsg;

    csMsg.Format( "%s has been successfully connected", m_csRemote );
    m_lbStatus.AddString( csMsg );
    csMsg.Format( "\\tand mapped to %s", m_csLocal );
    m_lbStatus.AddString( csMsg );
    UpdateData( FALSE );
}
else
    ReportError();
}
```

Alternately, if WNetAddConnection2 fails, the ReportError procedure is called to check the error code and report the reason for failure.

That’s it! These WNet functions are quite easy to use. You simply set and/or retrieve fields in a NETRESOURCE structure and let the function (or your end user!) make the desired connection.

Previous	Table of Contents	Next



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The LANMan Functions

The local area network (LAN) Manager API (LANMan) has been around since Microsoft's LAN Manager, predating Windows NT. The function set has been modified and expanded with each consecutive version of Windows NT. For the most part, these functions are duplicated in various other Win32 API functions, which often include additional features and parameters. For example, if you read the section on the security API in Chapter 11, "Security and Cryptography," you may have noticed that the user and group security functions provided in the LANMan function set are simpler to use but leave out important aspects of Windows NT security. There are no parameters, for instance, which take the ACCESS_TOKEN or allow you to impersonate a user. On the other hand, if you do not need these aspects of security, you may want to take advantage of the LANMan functions' ease of use.

NOTE:

You may be wondering (because of the functionality overlap with other parts of the API) whether the LANMan functions are on the way out. In fact, this does not appear to be the case. There is no mention in the SDK documentation of these functions being phased out. Moreover, a couple functions were even added in the Windows NT 4.0 release.

You might want to use these functions where convenient in smaller applications. But before you do, check to see if there is a better alternative provided elsewhere in the Win32 API; this will often be the case. Additionally, when you're implementing security, some things can be done only using the security API. For consistency, it is probably better to avoid mixing calls to both types of security functions in the same program.

If you decide to use the LANMan functions (which require Windows NT), keep in mind that, at least currently in Windows NT/2000, the functions that use strings always use Unicode-style strings. You'll need to convert strings back and forth using the MultiByteToWideChar and related string-conversion functions.

NOTE:

The LANMan functions are not supported by Windows 98. In some cases, the function names are exported, but with empty functions that only return an error code. Other functions are simply not callable unless you're running the application on a Windows NT/2000 station. If you want more information about the LANMan functions,

Winsock 2.0 for Network Programming

We discussed the Winsock 2.0 and WinInet functions in Chapter 19, "Internet Support." Here, we will look at the WinSock API from a LAN application point of view.

Although Winsock started life as a means of unifying different vendors' implementations of TCP/IP for the Windows platforms, one of the stated goals of the Winsock consortium was to make Winsock be protocol-independent. With Winsock 2.0, this has largely been accomplished. I say "largely" only because there are certain protocol-specific details you must still be aware of, although admittedly, these have to do with things such as dealing with different network address sizes, which differ among the different network protocols. Also, areas of functionality that are unique to certain protocols are supported by using Winsock's protocol-dependent constants and structures, typically defined in an extra protocol-specific header file.

In reality, Winsock 2.0 is probably as network-independent as you can get, that is, without taking some form of lowest-common-denominator approach. In each protocol, you communicate by sending and receiving data using the sockets model. Initializing and querying of protocol-specific information is done via common functions but by making use of protocol-specific structures and settings. Additional protocol-specific functionality is accessed by using the `setsockopt` "set socket option" function (as demonstrated in the *MacSock* demo).

TIP:

Winsock version 1.1 shipped with Windows 95 and earlier versions of Windows NT. Winsock 2 ships with Windows 98 and NT/2000, and is automatically upgraded in older Windows 95 installations by installation of the Windows 95 Service Pack or certain other upgrade files that are available for download via Microsoft's Web site (www.microsoft.com).

New Features of Winsock 2

New features of Winsock 2 include the following:

- Faster performance using overlapped I/O
- Protocol-independent multi-point/multi-case support
- Quality of service (QoS)
- Simultaneous support for multiple network protocols
- Encrypted socket traffic via SSL (Secure Socket Layer) and PCT (Private Communication Technology)

Just as `CreateFile`, `ReadFile`, and `WriteFile` can be used to perform overlapped (or asynchronous and concurrent) file or pipe I/O, Winsock sockets can also be read from and written to asynchronously, with (if you wish) multiple sockets transferring data simultaneously. This permits each socket to transfer data at its own rate of speed, unencumbered by any throughput problems on other network protocols or hardware connections.

The built-in protocols supported by Winsock 2 currently include TCP, UDP, NetWare IPX and SPX, NetBEUI (using NetBIOS), AppleTalk, and TP4. Support for additional protocols can be added by other vendors, who conform to Winsock's SPI (Service Provider's Interface) specification.

The transport protocol annex of Winsock 2 handles the area of additional protocol-specific functionality. The SPI allows network service providers to supply their own name space resolution code and supports simultaneous network communications via multiple protocols.

QoS is another 2.0 feature that allows applications to adjust for throughput speeds and connection quality, and to request changes in connection quality (such as by opening another ISDN channel when necessary).

During this last year, Winsock has also gained support for secure data transmission via encryption using protocols such as SSL and PCT. This means you can have your Internet application negotiate a secure connection to an SSL supporting Web server, and then retrieve data over a secure socket. In practice, SSL and PCT support in Winsock make heavy use of the `WSAIocctl` API function, and once the desired socket has been configured in this way and the secure connection has been established, encrypted data can be sent and received transparently by means of the various Winsock functions.

The MacSock Demo: Winsock 2 and AppleTalk

Chapter 19 included a couple of examples of using Winsock for TCP programming. Here, we'll go through a simple example of using the Winsock API to do AppleTalk programming. This example demonstrates how Winsock handles other protocols gracefully.

NOTE:

The *MacSock* demo is included on the CD that accompanies this book, in the Chapter 20 folder. You will need the AppleTalk protocol installed on a Windows NT system, plus one or more Macintosh computers on your network, in order to actually run the *MacSock* demo. If you don't have a Mac or an NT system, you can still follow along with the example to see how it works.

This demo simply creates a Name Binding Protocol (NBP) AppleTalk datagram socket, binds it to an AppleTalk address, and then registers a network name on a specific AppleTalk zone.

For convenience—primarily to display a list of actions but also to show any error results that occur—we've used a dialog-based application (see Figure 20.2, later in this chapter) to present a list box and a single button.

NOTE:

When AppWizard was used to create the *MacSock* demo project, support for WinSock was included by request, as discussed in Chapter 19.

The dialog code is quite simple and contains two procedures: `OnConnect` to create the connection and `ReportError` to display appropriate error messages. However, since the demo relies on the AppleTalk protocol, we also need to include the `Atalkwsh.H` header file, which defines AppleTalk-specific structures and constants. Inside the `OnConnect` function, we define the five variables that will be used:

- `nError`, `nStatusUsed` for result (or error) status codes from the various Winsock functions
- `hSocket` Handle to the AppleTalk socket that is created
- `SocketAddr` An AppleTalk-specific socket address structure
- `RegName` An AppleTalk-specific structure that is used for registering a name in a given zone on the network

Because we're using Winsock support provided by MFC, there is no need to call `WSAStartup` to initialize Winsock or to check the Winsock version. Instead, our first requirement is simply to open an AppleTalk NBP datagram socket with the protocol family specifier set to `PF_APPLETALK`.

```
hSocket = socket( PF_APPLETALK, SOCK_DGRAM, DDPPROTO_NBP );
if( hSocket == INVALID_SOCKET )
{
    nError = WSAGetLastError();
    ReportError( "Error creating socket:", nError );
    return;
}
else
    m_lbStatus.AddString( "Created socket" );
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

If `hSocket` is returned as `INVALID_SOCKET`, then a call to `WSAGetLastError` will return the actual error code to allow the `ReportError` procedure to add the location message and an error message to the list box. Or as we hope, on success, a simple location message is added to the list.

Assuming that we've successfully opened a socket, the next step is to prepare a `SocketAddr` structure (remember, this is an AppleTalk-specific socket address structure).

TIP:

Contrast the `SocketAddr` structure used here with the `SocketAddr` structure used in the *SendMail* demo introduced in Chapter 19.

```
// Prepare SocketAddr structure to bind to the socket.
SocketAddr.sat_family = PF_APPLETALK; // AppleTalk family
SocketAddr.sat_net    = m_nNetNumber; // network number
SocketAddr.sat_node   = m_nNetNode;  // the node number
SocketAddr.sat_socket = hSocket;     // the AppleTalk socket
```

The *MacSocket* dialog includes edit fields to allow you to enter values for the net number and node. The default values of 0, however, should be satisfactory for most small systems. However, if you're testing this application on a large AppleTalk network, you may need to set the network and node numbers to something more specific than zero.

Next, we need to bind the socket to the address specified in our `SocketAddr` structure. Note that the third parameter to bind specifies the size of `SocketAddr`, which in this case will be different from a "normal" TCP address structure. In a similar way, if you were binding to an IPX socket, you would use a different-sized address structure that is unique to IPX.

```
nStatus = bind( hSocket, (LPSOCKADDR) &SocketAddr, sizeof(SocketAddr) );
if( nStatus == SOCKET_ERROR )
{
    nError = WSAGetLastError();
    ReportError( "Error binding address to socket:", nError );
}
```

```

        return;
    }
    else
        m_lbStatus.AddString( "Bound address to socket" );

```

In the next step, the RegName structure is filled with an object name, type name, and zone name. The object name and type can be whatever you want, but you will need to set the zone name to a valid zone on your AppleTalk network if you are actually running this application. Each corresponding string-length field also must be set accordingly.

```

        // prepare RegName struct to register our name on network
        strcpy( RegName.ObjectName, "MacSock Server" );           // network name
        RegName.ObjectNameLen = strlen( RegName.ObjectName );
        strcpy( RegName.TypeName, "WinSock2 Type" );             // type name
        RegName.TypeNameLen = strlen( RegName.TypeName );
        strcpy( RegName.ZoneName, "Jonny Appleseed" );           // zone name
        RegName.ZoneNameLen = strlen( RegName.ZoneName );

```

Once the RegName structure is prepared, the Winsock setsockopt call is used to register our name on the specified AppleTalk zone. Incidentally, some sources suggest that this function call may take as long as 10 seconds (or perhaps even longer) to execute, so be patient!

```

        nStatus = setsockopt( hSocket,           // AppleTalk socket
                             SOL_APPLETALK,     // option level
                             SO_REGISTER_NAME,  // option value requested
                             (char*) &RegName,  // pointer to option buffer
                             sizeof(WSH_REGISTER_NAME) ); // buffer size

```

The second parameter specifies that the option we're requesting is part of the "AppleTalk level," and the third parameter specifies that we want the "register name" option. Parameter four is a pointer to our option structure. The fifth parameter specifies the size of this structure.

```

        if( nStatus == SOCKET_ERROR )
        {
            nError = WSAGetLastError();
            ReportError( "Error registering name:", nError );
            return;
        }
        else
            m_lbStatus.AddString( "Name registered successfully" );

```

Unfortunately, in tests, this is also the point where the experiment fails with an error #10022—an error that can't be found in any ready documentation on either the Windows or Macintosh systems. The results are shown in Figure 20.2.



FIGURE 20.2 Attempting a connection using AppleTalk (unsuccessfully)

The problem here, apparently, lies in how AppleTalk is set up on the target (Macintosh) system and whether there is a corresponding zone to be logged to. The problem, to quote my expert (a high-dollar troubleshooter for an international communications company—both names withheld to protect the innocent) is that "...we don't use the AppleTalk protocols. We use TCP/IP for network comm..."

Successful or otherwise, at this point, we can close the socket. Again, there's no need to call WSACleanup to tell Winsock we're finished, since this is handled automatically.

```

        closesocket( hSocket );
    }

```

That's it, as far as this example goes. In real life, you would obviously proceed to do something on the network—as either a client or a server (or perhaps both). You should get a successful return code when registering the name, but instead of exiting immediately, you would go on to do something else, then eventually de-register your name when finished, and finally exit.

Because Winsock2 not only supports other protocols, but also supports using multiple protocols in the same application simultaneously, you could create a server application that serves Apple, NetWare, and TCP/IP clients, perhaps using some new user-level protocol you've devised. Or, you could use Winsock to build a software router, reading stream or datagram data from one network type, and forwarding it over a second protocol. Depending on the combination of protocols used, however, this could mean needing to convert different-sized data packets that would require using an intermediate packet-repackaging strategy.

Again, however (as mentioned by my consultant), the TCP/IP protocols—discussed in Chapter 19—are in more common use.

TIP:

For Mac users who are confused by Macintosh error messages, Bleu Rose Ltd. offers Black & Bleu" version 1.1, an application that shows Apple's explanation of all of the common and most of the unusual Macintosh OS error codes. Black & Blue is available from the company's Web site at www.bleurose.com.

Named Pipes and Mailslots for Network Programming

Named pipes and their cousins the mailslots initially made their appearance in OS/2, and they continue to be an excellent high-level means of passing data and messages between Windows NT stations connected on a network. Mailslots and named pipes are similar in the way that they are created, read from, and written to. They differ in the respect that mailslots are one-way and connectionless; named pipes can be two-way and require that a connection be established on each end prior to use. If you're familiar with TCP/IP, mailslots are analogous to UDP—both are used for connectionless, datagram transfers. Named pipes are analogous to TCP—both of these are used for connection-oriented messages or stream I/O.

One drawback to using named pipes is that currently Windows 98 (and Windows 95 and Windows for Workgroups) supports only client-side pipes. This means that any server applications you write that use named pipes must run on a Windows NT/2000 workstation or server. Also, while Windows 98 does support mailslots, the security descriptor parameter is ignored unless your software is run on a Windows NT/2000 station.

Another drawback is that pipe and mailslot communications are only as dependable as the network protocol that is being used to support them. In most cases, this should not be a problem, but for transferring very large amounts of data completely error-free, the Win32 file-system functions, for example, would provide better data integrity.

On the other hand, there are several benefits to using pipes and mailslots: They are easy to implement, protocol-independent, and can transfer data asynchronously using overlapped I/O. Because they are implemented as devices in the operating system, they can also be used to redirect I/O to or from other devices, either locally or across a network.

NOTE:

There are also unnamed (or anonymous) pipes, which work very much like named pipes and can be used for inter-process and inter-application communication on the same workstation. Anonymous pipes are supported by Windows 95/98 as well as Windows NT/2000. These differ from named pipes in that they can be read from and written to locally only on the station that creates them. Use the `CreatePipe` function to create an anonymous pipe. Pipes and mailslots, and the functions that are pertinent to creating and using pipes, are discussed in Chapter 7, "Creating Processes and Pipes."

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Table 20.3 lists the Win32 API functions related to mailslots and I/O functions supporting pipes and mailslots.

TABLE 20.3: Mailslot-Specific Functions and I/O Functions Supporting Pipes and Mailslots

Function	Description
CreateMailslot	Creates and returns a handle to a mailslot with the specified name
GetMailslotInfo	Retrieves information about the specified mailslot
SetMailslotInfo	Sets the time-out value used for read operations on the specified mailslot
CreateFile	Creates a pipe or mailslot
ReadFile	Reads from a pipe or mailslot
WriteFile	Writes to a pipe or mailslot
ReadFileEx	Allows overlapped (asynchronous) reading from a pipe or mailslot
WriteFileEx	Allows overlapped (asynchronous) writing to a pipe or mailslot
CloseHandle	Closes a handle to a pipe or mailslot

Remote Procedure Calls

Remote Procedure Calls (RPCs) provide a mechanism that makes it possible for programs or code located on one computer to call functions and procedures located on another computer, and to do so in a way that makes the calling syntax exactly like calling procedures located in a standard DLL on the local station.

With the debut of DCOM in Windows NT 4.0, RPC is taking on significance not only in its own right, but also because it is the means used by Microsoft to make the Windows implementation of COM (Component Object Model) work remotely across networks (called DCOM). In other words, it is worth your while to learn RPC, not only to be able to make remote procedure calls, but also to be able to make and access remote *objects*.

The RPC “glue” implemented in Windows 2000 conforms to the Open Software Foundation (OSF) specifications. This means that conforming to RPC (or DCOM) in your application ensures that functions you

call can be located on a remote computer running a completely different operating system or, conversely, that procedures or object methods you've implemented on Windows 2000 (or Windows 95/98/NT) can be accessed by applications running on other operating systems—even using different bit-size hardware or a different byte-ordering scheme.

Along with these benefits, as you might guess, comes a measure of added complexity. Although implementing RPC and DCOM has gotten somewhat easier with recent compiler tools, understanding how these technologies work is still more difficult than perhaps it should be. In any case, we will see how RPC and DCOM work, and show you how to get started with implementing these technologies into your applications.

As mentioned earlier in this chapter, certain aspects of the networking APIs are not as fully implemented in Windows 98 as in Windows NT/2000. As with other forms of networking communications, this affects RPC in the sense that although you can create RPC clients and servers for Windows 98, you cannot add the same level of security to RPC servers as you could when the servers are running on Windows NT/2000. Additionally, there are certain network-transport mechanisms that are fine for RPC use but are supported only client-side for Windows 98 (named pipes are an example).

RPC Concepts

Before going into the details of what you'll need to use RPC, you should understand some RPC-related concepts and terms. These include UUID, GUID, interface class, and marshaling. Here, we look at how these concepts apply to networking applications.

NOTE:

These topics are also discussed in Chapter 22, "Building and Using COM Servers in VC++," where they relate to COM operations.

UUIDs and GUIDs

A UUID is a *universally unique identifier*, also known as a *globally unique identifier*, or GUID. In Window systems, a UUID or GUID is used to ensure that RPC *interfaces* and COM *classes* are absolutely, positively guaranteed to have a unique identifier entry in the system registry. (The system registry, among other things, maintains settings information for each interface and class that is registered with the system.)

UUIDs or GUIDs are guaranteed to be unique, provided all RPC and (D)COM programmers abide by two simple rules:

- Every interface and class must be assigned a separate and unique UUID. This UUID is always a 128-bit (16 byte) value, which you can produce either by running a program such as UUIDGEN or GUIDGEN, or by submitting a request to Microsoft for a "block of UUIDs." (Microsoft usually responds to such a request quickly, and you typically get a block of 100 UUIDs via e-mail or floppy.)
- Anytime you change an interface definition (or a COM or DCOM class), you must leave the old interface intact and define a new interface with a new UUID.

Provided every programmer follows these rules, UUIDs or GUIDs are extremely unlikely to conflict in the registry. This is good, because the Windows system registry would become very confused if two or more interfaces (or classes) tried to refer to themselves using the same identifier.

NOTE:

In case you're wondering, the UUIDs are generated via an algorithm that mixes together the address of your Ethernet network adapter (which for other reasons, is also supposed to be completely unique; every network card manufacturer is assigned a different range of such addresses to draw from), plus system information. This algorithm produces a number that is extremely unlikely to ever occur twice on either the same or different machines.

Interface Classes

An RPC/COM interface is a group of one or more related functions. RPC applications usually have only a single interface, although they could have more. COM classes also have at least one interface, and they most frequently have at least two or three. Again, each interface is one group of one or more related functions.

Each interface can have a string name as well as the UUID. The string name you assign to an interface is not

nearly as important as the unique UUID that you give it. However, to keep things simple, you should try to assign descriptive string names for your interfaces as well as unique UUIDs.

For example, if you needed to implement an RPC server that could execute a set of mathematical functions and also some string-manipulation functions, you might implement a `mymath` interface and a `mystring` interface. In this case, the client RPC application would then use either the `mymath` interface to access your math functions or the `mystring` interface to access your string functions. An application can certainly use both interfaces, but each interface can be used to access only functions that are part of that particular interface.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Marshaling Code

The way RPC differs from local procedure calling is primarily in the extra steps that are required to package (and unpack) the function parameters so they can be shipped across the network each time a function is called. At runtime, these parameter-handling steps are performed by a set of functions that have been linked into your client (or server) application. These functions know all about the function prototypes being called and how to pack and unpack the parameters correctly. This process of packaging and unpacking function parameters for transport across the network is called *marshaling*. As you'll see, much of the work involved in implementing RPC has to do with the steps required to create the marshaling code stubs that are linked into your application.

Marshaling includes taking care of pointer and string parameters so that they work as expected on both ends. If you think about it, passing a pointer to a string across a network means that the string itself needs to be sent across the network. A new chunk of RAM must be allocated for the string on the other computer, the string contents need to be copied into this new RAM, and a new pointer variable must be created. This pointer variable has the same data type as the original pointer, but points to this new string copy. If this pointer-to-a-string parameter is defined in our IDL files as an [in, out] parameter (as described shortly), then the string-copy process needs to happen a second time to send the (presumably) modified string back to the original client program. Finally, marshalling code on the client side needs to copy this version of the pointed-to string back into the original string pointed to by the client's string pointer.

As you can see, RPC (and for the same reason, COM and DCOM) completely changes the normal perception of pointers as a very efficient way to pass parameters. When sent remotely, pointers are actually very inefficient! But the bit of time and network traffic eaten up by passing this parameter (and copying it) are the price for the flexibility and power that remote procedure calling (and DCOM) provide.

TIP:

It is still best to take more than the normal care when designing functions that will be called over RPC. Engineer them for efficient parameter passing across the network. In addition to decreasing the amount of data passed in parameters, you should also look for opportunities to reduce the number of times individual functions are called. This may mean packing several function calls together into one function, perhaps passing an array or structure of parameters, rather than using more function calls using fewer parameters.

The RPC Network Connection

The actual network connection required for making RPC calls is determined by either making an explicit choice of a networking protocol in your software (which must be consistent on both the client and server sides), or else by allowing the computer to search the network namespace for the required server and protocol, and connecting accordingly. There are several things to note here:

- Because the name space is not maintained by Windows 98 stations, if you wish to go this route, you will either need to run the server side of your application on a Windows NT machine or else at least configure the Windows 98 station to use a Windows NT station for its name space server.
- Although RPC servers can be run on Windows 98, you must take care not to specify a protocol that requires Windows NT in order to function in a server capacity. For example, creating a server-side named pipe requires Windows NT/2000 (although a Windows 98 system can connect to and communicate with such a server-side pipe as a client).

One benefit to using the name-space approach is that locations (and protocols) do not need to be hard-coded into your program, and thus both client and server RPC software can more easily be moved from one machine to another. Disadvantages of using the name-space approach are its somewhat greater complexity, that the search itself is initiated somewhat differently between Windows NT/2000 and Windows 98, and that under certain conditions you may need to specify which domain to search in. Also, the search itself can take time, causing an irritating delay as your program attempts to connect. This is especially true when you do not specify particular protocols and the connection process proceeds to cycle through all the installed protocols.

Creating an RPC Project

To demonstrate RPC operations, we'll create a project consisting of three separate applications: a server application, a client application, and the remote procedure library itself. For simplicity, the first step will be to create a blank workspace and to then create the three projects within the single workspace.

The advantages in this approach are simple; by keeping all three projects in a single workspace, switching between files is convenient and we don't need to go looking through multiple directories to find the various components. Each project within the workspace will, of course, have its own separate project directory, but since these are unified by the workspace, access remains simple.

Before working on the client or server sides, however, our first step is to define the RPC services themselves and the interface providing access to the services.

Mathlib.CPP: A Function Library

Before you can write an application that uses RPC, you need to come up with the set of functions that will be called remotely. These do not need to be in a separate file, but it is very wise to set them up that way. This lets you test and debug the functions in a "normal" program, before you begin implementing the RPC portions of your client and server programs.

The *RPC* demo applications use a very simplistic set of math functions. These are placed in a separate source file, which is named Mathlib.CPP.

NOTE:

Obviously, the small amount of work these simple math functions do wouldn't justify the overhead of using RPC. But in choosing the functions for this demo, the goal is to keep them as simple as possible while passing and receiving integer and float values as well as returning strings.

The first thing to note about the math library is that it includes the `Rpc.H` header file. Initially when debugging your functions, you may want to comment this out, because you'll presumably be statically linking the functions to your test program.

The `Mathlib.H` header file bears mention also. Initially, you will want this to be a normal function prototype header file, which you can easily make just as you would any other normal header file (by simply copying the function header from your source file and placing semicolons at the end of each). Once you've verified that each function works correctly, you will want to rename this header file to get it out of the way, since one of the steps to building an RPC application results in generating a new header file for this set of functions. However, it's a good idea to save your "normal" header file, in case you need to do further debugging down the road.

```
#include <iostream.h>
#include <rpc.h>
#include "mathlib.h"
```

The first of the functions declared simply prints to standard output the client name that is passed to it. Remember that eventually these functions will execute on the RPC server, so the client will call SendClientName, passing a string, and the server will actually execute the call.

```
void SendClientName( char *pszClientName )
{
    cout << endl << "Client: " << pszClientName << endl;
    cout << flush;
}
```

As with previous applications, the output stream is flushed here because otherwise, you would not see the display update immediately as each function is called.

The next function returns a string back to the calling program. Again, this will be the RPC client application, although the function will actually execute on the server.

```
char *GetMathServerName( void )
{
    static char szServerName[64];

    strcpy( szServerName, "MathLIB Server" );
    return szServerName;
}
```

Next, we have the four math functions, each of which takes two long integers, perform a simple calculation with them, and return a long result value back to the calling program. These examples are simple, but you can make your RPC functions perform as much complex number crunching (or other processing) as you wish. Perhaps the server machine will have a coprocessor, or multiple CPUs, or MMX support, which the client machine can take advantage of by offloading to it the most CPU-intensive functions.

```
long Add(long lNum1, long lNum2)
{
    return( lNum1 + lNum2 );
}
long Subtract(long lNum1, long lNum2)
{
    return( lNum1 - lNum2 );
}

long Multiply(long lNum1, long lNum2)
{
    return( lNum1 * lNum2 );
}
float Divide(long lNum1, long lNum2)
{
    return( lNum1 / lNum2 );
}
```

Incidentally, another thing this demo does that you may want to avoid in a real application is update the display in each function call. In real applications, at least when your functions are likely to get called many times, it might make sense to move any function-related display updating to a different thread (or eliminate it altogether). Anything that can be done to increase the speed of your function execution will allow the function result to be returned back to the calling program that much sooner.

This completes the file that defines the functions that will ultimately get called via RPC. Again, for testing and debugging, execution profiling, and other maintenance, you will usually find it much easier to prepare a normal header file and call these functions directly, without further complicating things by using RPC. Once

your functions are working to your satisfaction, you can switch back to using RPC quite easily. In fact, you may want to use your RPC client application as the test program that calls your functions statically in “test” mode, and via RPC in “deployment” mode.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Mathlib.IDL: The IDL Function Interface Definition

In order to create the code that will “know” how to properly pack and unpack the function parameters (which then get sent over the network), it is necessary to create an IDL file. The IDL file gets its name from the abbreviation for the Interface Definition Language, in which it is written. Why do these functions need to be declared using IDL rather than C or C++? Well, one answer is that that’s what the OSF stipulates in its distributed-computing specifications. Another answer is that perhaps a bit more information is needed to be known about each function than C or C++ normally provides. In fact, if you look at an IDL file, it rather closely resembles C function prototypes, but with the addition of zero, one, or more attributes enclosed inside square brackets, which precede various function parameters and return values. These attributes most frequently simply identify whether the parameter is an input and/or output parameter:

- An input parameter is signified by [in].
- An output parameter is signified by [out].
- A parameter that is passed in and is modified is signified by [in, out].

The attribute information might also include additional data-typing information, as we’ll see with the string parameters in our example. Furthermore, starting out the IDL file is a header that includes a UUID. In the case of our Mathlib example, the corresponding IDL file starts with a header that looks like this:

```
[  
  uuid (d6725740-1b8e-11d3-9c00-004005360dec),  
  version(1.0),  
  pointer_default(unique)  
]
```

The UUID value shown was generated by the UuidGen utility.

At this point, if your compiler IDL tool is configured as described in the previous sections (and if you have a network adapter installed), you should be able to go to your Tools menu, select the Generate GUID option, and have a new GUID appear. Each time you do this, note that a somewhat different identifier appears. This is how you generate new interface and class GUIDs.

NOTE:

The UuidGen utility uses a time value and an ID code contained in your machine's network card to generate a UUID value that is guaranteed to be unique. Thus, the number in brackets after the `uuid` was created by a copy of UuidGen, running on a specific computer. To make changes to the *RPC* demo programs and recompile them, simply generate a new GUID for the `MathLib.idl` file, then use the IDL compiler to generate the remaining source files. Besides, if you are changing the *RPC* demo, you'll want to begin with the `.idl` file and use it to recreate the `MathLib.h`, `MathLib_s.c` and `MathLib_c.c` files.

The blank space (and blank lines) between the various parts enclosed between the square brackets is optional, but typically is used to make the header more human-readable.

The second part of the `Mathlib.idl` file is the interface definition. Again, this looks very much like a C++ class definition, except that you'll notice there are square brackets containing *attribute* information, preceding certain parameters or variables. The `[unique, string]`, for example, makes it clear that the `GetMathServerName` function returns a string type, and that the string value is to be treated as a unique value (not copied into a string-duplicates table that replaces identical strings with duplicate references, for example). The `[in, string]` makes it clear that the parameter being passed to `SendClientName` is a string and can be ignored after the function executes. If it included the `out` attribute, additional code would be generated that would copy the string back again after the function executed, and then send that string value back to the client. (If you're wondering *who* is generating *what* code, we'll get to that shortly.)

```
interface mathlib
{
    [unique, string] char *GetMathServerName ( void );
    void SendClientName ( [in, string] char *string );
    long Add ( long lNum1, long lNum2 );
    long Subtract ( long lNum1, long lNum2 );
    long Multiply ( long lNum1, long lNum2 );
    long Divide ( long lNum1, long lNum2 );
}
```

The MIDL compiler will use the `Mathlib.IDL` file to create `Mathlib.H` and a marshaling stub file for the RPC server and client applications. But before we can use `Mathlib.IDL`, we need to also create a file called `Mathlib.ACF`, which is what we'll do next.

Mathlib.ACF: The Marshaling Interface Handle File

The next thing we need to do is create a marshaling interface handle file. For our example, we'll use an automatic interface handle. You could define your own interface handle, but this would require that you write your own marshaling code as well. Since we are using "standard" data types, we can use automatic marshaling for this project, and our `.ACF` file should contain the following interface handle declaration:

```
[auto_handle]
interface mathlib
{
}
```

If we were planning to support multiple RPC interfaces in our project, we would need to add multiple interface declarations, rather than just the `mathlib` one we've declared in this file.

Mathlib.H, Mathlib_c.C, and Mathlib_s.C

Armed with `Mathlib.IDL` and `Mathlib.ACF`, plus our MIDL Tool entry, we are now ready to "compile" the `mathlib` interface definition to produce an RPC header file and client and server proxy stubs.

Assuming there are no typos in either the `Mathlib.IDL` or `Mathlib.ACF` file, you should be able to do this easily. Open the `Mathlib.IDL` file in your C++ IDE and make sure it's the selected window. Then choose the `Compile IDL File` option from the Tools menu. If all goes well, you should get three new files:

Mathlib.H An "RPC-icized" header file that the RPC client and server applications will need to include

Mathlib_c.C The client proxy stub that must be added to the RPC client application

Mathlib_s.C

The server proxy stub that must be included in the RPC server project

The Mathlib_c.C and Mathlib_s.C files contain the code that makes the RPC mechanism properly handle the parameters being sent out and received back in each function call. When the client and server applications link with these files, they think the function is being linked locally, but in reality, hooks to the RPC mechanism are set up to be invoked to process each function call at runtime. Note that outside of adding __RPC_x entries to type the arguments and return values in some of the functions, the header file looks pretty much like a normal header file would.

The client and server proxy stubs are lengthier. You can look at those listings (included on the CD that accompanies this book) or generate the code with the MIDL compiler to see the proxy stub code. Fortunately, it's not necessary to understand this code in order to compile and link it into your client and server application projects.

NOTE:

The *RPC* demos are included on the CD that accompanies this book, in the Chapter 20 folder. Remember, while the RPC client application may run on Windows 98, since the RPC server uses named pipes for communication, it should be on Windows NT/2000.

The *RPC* Client Demo: A Client Application

The next step in putting together this RPC project is to write the *RPC* Client application. This is reasonably straightforward, although there are a few items for you to note.

NOTE:

The *RPC* Client demo is included on the CD that accompanies this book in the Chapter 20 folder.

First, we include the Mathlib.H file, which was generated by compiling the .IDL file. The project file does *not* include the Mathlib.CPP file but does need to include the Mathlib_c.C file (which essentially fools the compiler into thinking we're linking to these functions locally).

```
#include <iostream.h>
#include <rpc.h>
#include "..\mathlib.h"
```

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Another thing we must do in both the client and server applications is define a pair of memory allocation and freeing functions, which will be called at runtime by the RPC mechanism when it needs to create local storage for parameters. This is how the RAM pointed to by the RPC pointer copies can have valid addresses inside the client (or server) processes. It actually *is* allocated by the process whenever RPC makes the allocation or deallocation calls.

```
void __RPC_FAR *__RPC_API midl_user_allocate( size_t len )
{
    return( malloc(len) );
}
```

```
void __RPC_API midl_user_free( void __RPC_FAR *ptr )
{
    free( ptr );
}
```

The `__RPC_API`, `__RPC_FAR`, and `_CRTAPI1` should be included just as they are here, although oddly, when you look these up in the `Rpc.H` header file, it appears that they are defined to nothing. Nevertheless, the use of these in the SDK examples suggests that it is wise to follow their lead for future and/or backward compatibility.

```
void _CRTAPI1 main(void)
{
    int a = 100;
    int b = 25;
```

Inside the main function, the `a` and `b` variables are defined and initialized. These will be the two integer parameters in calls to the RPC server Mathlib functions.

Next, we have what looks like the start of exception-handling code, except we use `RpcTryExcept`. This is exception handling, but done RPC style.

WARNING:

Exception handling is optional in standard C++ Win32 programs, but it is *mandatory* in RPC client programs. Leave out `RpcTryExcept`, and you won't have a working RPC application.

`RpcTryExcept`

```
SendClientName( "Client's name" );
cout << "Connected to Math Server " << GetMathServerName()
    << ".\n";
cout << a << " plus          " << b << " = "
    << Add(a, b) << endl;
cout << a << " minus          " << b << " = "
    << Subtract(a, b) << "\n";
cout << a << " times          " << b << " = "
    << Multiply(a, b) << "\n";
cout << a << " divided by " << b << " = "
    << Divide(a, b) << "\n";
```

After the `RpcTryExcept`, the client application tries out each of the `mathlib` interface's functions, displaying the results of each call. If any RPC runtime exceptions occur, they will be handled by the code after `RpcExcept`, which simply displays an error message in this example.

```
    RpcExcept(1)
        cout << "RPC Runtime error occurred.\n";
    RpcEndExcept
}
```

Finally, `RpcEndExcept` ends the exception handling, and the client RPC program terminates.

TIP:

For debugging purposes, you may want to make a `DebugRpc.H` header file, which defines the RPC exception handling as empty macros. If you then add the `Mathlib.CPP` source into the client project and temporarily remove the `Mathlib_c.C` file from the project, the resulting application will let you run, test, and debug the functions locally.

The *RPCServ* Demo: A Server Application

The last source file that must be written is the RPC server. This program may strike you as rather odd, in that apparently, nowhere are any of the `Mathlib` functions being called.

NOTE:

The *RPCServ* demo is included on the CD that accompanies this book in the Chapter 20 folder.

Actually, the server application's primary duties are to register its existence with the RPC mechanism, specify what network protocol(s) it will use to listen on, and then wait around for clients to connect and make RPC calls. The actual calls are then intercepted via the RPC mechanism, which causes the `Mathlib` functions exported by the server application to be called.

```
#include <iostream.h>
#include <rpc.h>
#include "..\mathlib.h"

void _CRTAPI1 main(int argc, char *argv[])
{
    RPC_STATUS          status;
    RPC_BINDING_VECTOR    *pBindingVector;
```

As with the client application, we include `Iostream.H` and `Rpc.H` header files, plus the `Mathlib.H` header file, which was generated via the MIDL compiler. The `RPC_BINDING_VECTOR` can be thought of as a table of function pointers to all the functions defined in the `mathlib` interface. However, `pBindingVector` will not be initialized until several steps into our program.

```
UCHAR *pszProtocolSequence = (UCHAR *) "ncacn_np";  
WORD  wMinCalls             = 1;  
WORD  wMaxCalls             = 30;  
UCHAR *pszEndPoint          = (UCHAR *) "\\pipe\\auto";  
WORD  fWaitFlag             = FALSE;  
UCHAR *pszEntryName         = (UCHAR *) "/./Autohandle_mathlib";
```

The pszProtocolSequence variable specifies that named pipes are the method this RPC server will use to accept RPC client calls. The wMinCalls and wMaxCalls specify the minimum and maximum simultaneous calls this server will handle. The pszEndPoint variable specifies the name of the pipe used to listen for client calls, which uses the named pipe of auto for automatic marshaling.

The fWaitFlag will be used to tell RPC not to let this server terminate, but to wait, listening for client calls. The pszEntryName variable contains the name server entry that the mathlib interface will be registered as, so that it can be located by name.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The first RPC function is a call to `RpcServerUseProtseqEp`, which in unabbreviated form means “use specified protocol sequence and endpoint” for incoming RPC calls. So in this case, we’re specifying that we want a named pipe for clients to connect through.

```
status = RpcServerUseProtseqEp(
    pszProtocolSequence,    // protocol sequence
    wMaxCalls,              // maximum concurrent calls
    pszEndPoint,            // end point (using pipe)
    NULL );                 // security (default used)

if( status )
{
    cout << "error "<< status << " setting endpoint!\n";
    exit( status );
}
```

TIP:

The line `exit( status );` is optional and a simple `exit;` would function just as well. The compiler, however, will issue a warning statement if the latter form is used.

Next, we need to register the `mathlib` interface via a call to `RpcServerRegisterIf`. The first parameter is the interface ID defined by the MIDL compiler. The `NULL` second parameter specifies to use a nil-type manager `UUID`. This and the `NULL` third parameter sets the server to use the default manager and entry-point vector as generated by our MIDL-compiled proxy stub.

```
status = RpcServerRegisterIf(
    mathlib_v1_0_s_ifspec,  // Interface
    NULL,                  // Default/nil MgrTypeUUID
    NULL );                // MIDL-generated Entry-point vector

if( status )
{
```

```

        cout << "error " << status << " registering server!\n";
        exit( status );
    }

```

Then we request a pointer to the binding handles (vector table), through which RPC calls can be received.

```

status = RpcServerInqBindings( &ppBindingVector );
if( status )
{
    cout << "error " << status << " retrieving bindings!\n";
    exit( status );
}

```

The final preparatory function we need to call is `RpcNsBindingExport`, which establishes an entry in the name-service database that includes the binding handles we've just retrieved in the previous call.

```

status = RpcNsBindingExport(
    RPC_C_NS_SYNTAX_DEFAULT,    // name syntax type
    pszEntryName,               // Name Service name
    mathlib_v1_0_s_ifspec,     // interface handle
    pBindingVector,             // binding vector (we received)
    NULL );                    // UUID object vectors (none)
if( status )
{
    cout << "error " << status << " Exporting bindings!\n";
    exit( status );
}

```

Having registered and advertised our RPC interface "service," we now only need to tell the RPC runtime library to listen for incoming RPC calls.

```

cout << "Listening for RPC calls..." << endl << flush;
status = RpcServerListen( 1,          // mininum calls
                        30,          // maximum calls
                        FALSE );      // wait value (FALSE = DO wait)

if( status )
{
    cout << "error " << status << " during listen!\n";
    exit( status );
}

```

If we call `RpcServerListen` and tell it to return immediately, then the following code needs to execute instead, to prevent this server application from exiting before RPC calls arrive. With the last parameter set to `FALSE`, however, the following section of code can be ignored.

```

if( fWaitFlag )
{
    status = RpcMgmtWaitServerListen();
    if( status )
    {
        cout << "error " << status << " during listen!\n";
        exit( status );
    }
}
}

```

The final code in the server is again a pair of memory allocation and freeing functions, identical to those defined in the RPC client. The RPC runtime code calls these functions when it needs to allocate local memory for copying buffers, arrays, strings, and pointers.

```

// These get called by MIDL

```

```

void __RPC_FAR *__RPC_API midl_user_allocate( size_t len )
{
    return( malloc(len) );
}
void __RPC_API midl_user_free( void __RPC_FAR *ptr )
{
    free( ptr );
}

```

This concludes the server source file. After compiling both the server and client applications, we can test them.

RPC Client and Server Testing

Assuming the compiling went as planned, testing these two programs should be very easy. Just start up the *RPCServ* application in one prompt window, and start *RPCClient* in another. The client should locate the server, display the results of each function call, and terminate. Figure 20.3 shows an example of the client screen output. The server should display each RPC call coming in, then wait for more calls. Figure 20.4 shows an example of an RPC server screen.

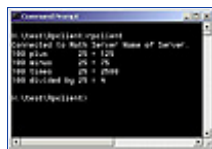


FIGURE 20.3 An RPC client screen

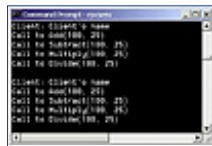


FIGURE 20.4 An RPC server screen

If the test works correctly, then once you remove the default security restrictions on named pipes, you should be able to run the client and server programs on different workstations. There are three ways to do this:

- Make an entry in the registry file corresponding to the auto named pipe, and then allow access to this pipe via the Windows NT User Manager (or Windows User Manager for Domains).
- On a Windows NT domain, you could test by using the identical login account on both systems.
- Apply the information in Chapter 11 concerning creating a NULL DACL (Device Access Control List), attach this to an SD, and then use this SD rather than NULL in the security parameter for the auto named pipe.

NOTE:

As explained in Chapter 11, security is enabled by default, and it takes explicit removal of restrictions before a device can be accessible to some (or all) users. If you attach a NULL DACL to your device's SD, this tells Windows NT to permit access by all users.

Summary

The past two chapters have provided an introduction to network and Internet operations, offering the basics for network communications. Obviously, given the importance of network operations, the coverage at this point has only skimmed the surface. This topic will be continued in the following chapters covering COM+ network operations.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PART VI **COM, COM+, and Active Directory**

- CHAPTER 21: COM / COM+ Overview
- CHAPTER 22: Building and Using COM Servers in VC++
- CHAPTER 23: New COM Features in Windows 2000
- CHAPTER 24: Active Directory Operations

CHAPTER 21 **COM / COM+ Overview**

- Advantages and disadvantages of COM
- COM components and interfaces
- COM clients, COM servers, and ActiveX controls
- COM and object orientation
- Windows Distributed interNet Applications Architecture (Windows DNA)
- The Component Services tool
- COM+ transactions
- The Queued Components (QC) feature
- Dynamic load balancing and object pooling

Microsoft's Component Object Model (COM) is widely used component software. It provides a rich set of integrated services, a wide choice of easy-to-use tools, and a large set of available applications. In addition, it provides the only currently viable market for reusable, off-the-shelf, client and server components.

In this chapter, we will explore what COM really is and how it works. We will discuss the general rules that make a piece of code a COM component. We will also look at the various types of COM components that we

can build.

Since COM is language-independent, we will be looking at COM from a Visual Basic (VB) as well as a Visual C++ (VC++) point of view. Although there are differences in the techniques for COM implementation in VB and VC++, the components will work identically after they are built. This is because COM is a binary-compatible component architecture.

What Is COM?

When Rapid Application Development (RAD) tools such as VB were introduced, the world of Microsoft Windows programming was opened to the masses. These tools provided a simple model of event-driven programming that anyone could use. No longer were you required to learn more complex programming languages and write large quantities of code in order to develop a Windows program. You could create a useful program in a fraction of the time it used to take. Even your little brother or sister could write a simple program.

To some degree, this is still true. VB makes Windows programming much easier and more productive for all programmers, from beginner to expert. The basic concept of constructing a user interface with the VB Forms Editor and then developing the code to interact with it is still a major part of the product. However, the tasks we are asked to perform as developers are now much more complex than they used to be. Our software must support multiple simultaneous users, access more complex databases, operate in multiple-computer environments across networks, and take advantage of new technologies. VB (as well as VC++) has evolved with the world of advanced software requirements and new technology, but it can't handle everything by itself.

In order to address some of the modern business requirements, such as easily modifiable and reusable functionality, the concept of software components was devised. A *component* is a discrete piece of software that does some specific, predefined work. Any program can use the component. Because a component is self-contained, it is easily replaceable. The idea is great because it allows you to write a component once and then use it everywhere. Then you can correct or enhance the component's functionality simply by updating and replacing the component. Making this system work, however, requires a formal standard that everyone can use to create components, ensuring that they are compatible and interchangeable.

COM is the Microsoft standard for creating software components. It is a specification for the construction of binary-compatible software components. This means that COM is not a programming language, a library of code, or a compiler. Rather, because COM is a binary specification, it allows you to build components that can communicate with each other regardless of the programming language or tool with which you choose to build them. COM allows you to concentrate on developing your application, and leaves it up to the compiler manufacturer to create the compiled components.

In short, COM is kind of like a rulebook. If you follow the COM rules, your components will be able to work with other COM components, no matter who wrote them.

COM Terminology

With every new technology or language comes a myriad of new terms. COM is no exception. Before we go any further, let's start with the following four basic concepts. Later in this chapter and in the next chapters, we will be digging deeper and defining other COM terminology.

Component code This is the actual work your component does. Once you set up the component infrastructure, your component needs to do something. For example, if your component is intended to perform work related to invoices, then you'll need to create component code to calculate totals, do inventory lookups, and so on.

Interface An interface allows any program to access the functionality of your component. An interface is a collection of public function definitions that your component makes available to your program and other components. The interface tells the rest of the world what your component can do and how to make use of its functionality.

GUID A Globally Unique Identifier (GUID) is assigned to every COM component and interface created, and it uniquely identifies the component to the operating system and other software. When you change your component or interface in any way, a new GUID is generated for it. A GUID is a unique 128-bit integer; for example, the GUID for the Microsoft MSFlexGrid control is 6262D3A0-11CF-91F6-C2863C385E30.

NOTE:

Usually, you don't need to worry about GUIDs. VB will create one for you, and VC++ has a GUID-generation program for this purpose. The algorithms used by these programs are guaranteed to generate unique GUIDs every time they are run.

Binary compatibility It is important to note that COM components conform to a binary standard. This means that regardless of the language you use to create your COM component, it will be compatible and usable by other COM components. For example, if your loan-calculator component is written in VC++ and your user interface component is written in VB, these components can still use each other's services.

COM Pros and Cons

Like other programming techniques, COM has its own pros and cons. It's important to know when to use COM and when not to use it. The information and guidelines presented in this section will help you to distinguish between a situation that calls for COM and one that does not.

The Benefits of COM

Understanding the benefits of COM will make it easier to decide when to use COM for your software project. When you see all that COM has to offer, you will be convinced of its usefulness.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

COM Components Are Easily Replaceable

There used to be only two sure things in life: death and taxes. We have changed that cliché to add one more thing that you can be sure of: software changes. No matter what the reason, software will require changes. When it does, you will be called on to make those changes.

Without COM, you would need to update your entire program to make even the smallest change. Typically, this would require making changes to code built into a large application installed on all your users' machines. Once the changes were made and the program was rebuilt, you would need to assemble a new release to be installed on all the client computers.

With COM, you don't need to update the whole application. You simply rebuild a single component and ship it.

COM Components Are Ideal for Changing Business Requirements

The business requirements of software are usually fluid. During development, as well as after software has been deployed, new requirements and changes to existing ones are constantly flooding in. Users' requests, bug fixes, and new requirements from marketing and management need to be accommodated. Your particular application may also require updates and frequent changes to business rules.

Because COM components are easy to replace, you can localize your business rules into a few components. If the business rules change, you can make the changes in the components, rebuild them, and distribute the new components. Your updates will be localized, and so will the likelihood of errors propagating through the program.

For example, a program that calculates taxes needs to be updated at least once a year to accommodate changes to the tax laws. Suppose you built those tax-law calculations into a single component called TaxCalcs. Your updates for the end of the year would be to that single component, which you could then send to your users. The component would isolate the changes, so that only that component and the code that directly uses it would need to be retested, not the complete program.

COM Components Make Reusability a Reality

One of the best things about COM is that it makes it easy to write some code once and then use it in many applications. For example, you could create a component that handles all your string functions. Any application could make use of your string-handling component. Over time, you could make corrections or improvements to the component, without needing to change any of the applications that use it.

COM Components Aid Parallel Development

Typical COM development means that you develop the component interfaces first, to make sure that everything will work together. Then you implement everything that the interfaces say you should. The nice thing about this is that once the interfaces are designed, you can distribute them to several programmers, and the implementation of the components can proceed in parallel. You might have a string-handling component, a calculations component, and a data-retrieval component. All of them can be built at the same time, and if the interfaces are designed properly, they will all work together when they're done.

The Limitations of COM

COM is good, but COM is not perfect. There are some annoying things about the mechanics of COM that you need to pay attention to, or COM will bite you.

COM Component Versions Are a Pain

Every COM component gets a GUID, a unique ID that the operating system uses to identify your component. These GUIDs are stored in the Windows Registry. Usually this would not be a problem. However, every time you change your component's interface, a new GUID is assigned to it. This mechanism is used as a version number to indicate to the software using the component that the interface has changed. It ensures that once you start using a component, you'll always be using the same version of the component. If you want to use an upgraded version, you get its new GUID.

The problem is that these component versions cause as much trouble as they avoid; however, they are a necessary evil. During development, you need to pay careful attention to the versions of the components you are using. Every new build of the component results in a new version. This is the case with VB components because you must manually configure your project to be binary compatible. It is easy for component versions to get out of sync, which can cause real debugging problems. Make it a common practice to check your component versions during development each time you create a new build.

Old Interfaces Must Hang Around

One of the requirements of COM is that once an interface is created, it never goes away. This guarantees that once a program makes use of a specific version of a component, that version's functionality is always supported. For example, if you use a specific function in a component, such as a quadratic-equation function, you are guaranteed that this function, with the same number and type of parameters, will be there. You can be assured that, regardless of the component version, the functionality will remain available.

As COM developers, we are required to keep old versions of component interfaces intact. You can add functionality to a component or update existing functionality, but you cannot remove existing functionality. The only real problem this causes is the evolution of a bloated component. At some point—if your component grows too large or has too much old functionality dragging it down—you may need to move your functionality to a new component.

COM Interfaces Require Careful Planning

As you'll learn in the next section, interfaces are an integral part of COM components. The interfaces are the communication mechanisms that components use to access each other. Every time a component or its interface changes, a new version is associated with the component.

Careful planning will help prevent too many versions of your interfaces from proliferating during development. If the components and the software that makes use of them are to have any chance of working together, the interfaces must be designed correctly the first time. One or two changes in an interface are usually not a problem. However, the more often you make changes, the more versions you get, and the more confusing it becomes to keep track of which one you should be using.

COM Components and Interfaces

A COM component is nothing without its interface. The interface defines the public functionality that other software and components can use. Typically, when you are planning your components, you design the interfaces first. This is important because the interaction between components is the most complex part of building a component-based application. It takes time and planning to define all the interfaces so that the components provide all the functionality that the rest of the software will need.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

PC hardware serves as a good analogy to help you understand COM components and interfaces. Your computer is expandable, allowing you to add or remove functionality in the form of additional hardware. If, for example, you need more display capabilities, you can replace the video board with a more advanced one. All you need to do is buy a new video board and put it into an open bus slot in the computer.

The same thing applies to software components. You first create a component that has an interface defined for it. For example, you might create a component that calculates the interest for a loan. Part of your interface would include access to the method `CalculateInterest`, which takes `Principal` and `InterestRate` as input parameters. It might look like this:

```
Public Function CalculateInterest( Principal As Currency,  
                                òInterestRate As Long) As Currency
```

Other parts of your program, including other components, can access this functionality by calling the component through the interface. Now suppose that you change the interest-calculation algorithm and need to update the functionality of the `CalculateInterest` method. As long as you don't change the interface, you can update the component code, recompile the component, and replace only the component. Everything else in the program—components or otherwise—now has access to the new functionality without requiring changes.

If part of your plan is to accommodate frequently changing business functions, you'll want to be able to simply drop in a new component that updates the software. If your interfaces do not change, then this will be a piece of cake.

What Are Interfaces?

A COM interface allows applications and other components to communicate with the functions of a COM component. The functions of a component are accessed through a *virtual function table*, also known as a *vtable* or *VTBL*. The vtable does not contain the actual functions. Instead, it contains a set of pointers to the functions of the component. A component that wants to access a function of another component must go through the vtable.

Clients can't access the vtable directly. Another pointer, called an *interface pointer*, adds an extra level of indirection to the interface, which makes implementing this interface possible. This means that a client will see a pointer to a pointer in the vtable. Figure 21.1 shows the structure of a COM interface.

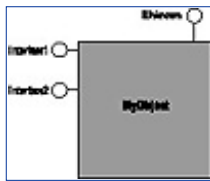


FIGURE 21.1 The structure of a COM interface

The only requirement for the vtable of a COM interface is that the first field of the table must be a pointer to IUnknown. IUnknown is the only interface that any component must implement to become a COM component. It's basically the door to all other interfaces, because all other interfaces inherit from IUnknown. When it comes to creating COM components in VB, you really don't need to do anything other than create a COM component; the rest is done for you transparently. We'll talk about IUnknown in more detail later in this chapter, as well as in later chapters.

In the meantime, we can explore how a COM interface works. To understand, we need to look at COM's calling convention. Methods of an object have one additional parameter: the object they belong to. This self-parameter is called *Me* in VB; VC++ programmers use the self-parameter *this*. A self-parameter to any of the interface's operations is passed to the interface pointer, which passes it to the vtable. Passing the parameter allows operations in a COM interface to exhibit true characteristics of a calling convention.

A COM component is free to contain implementations for any number of interfaces. The entire implementation can be a single class, but this does not need to be the case. Because a component can contain many classes, these classes can instantiate as many objects of as many kinds as they require. An object is a live instance of these classes. Objects collectively provide the implementation of the component's interfaces.

Components depend on interfaces to communicate. If the interfaces no longer exist or have been changed, then the communication lines are broken. Once an interface is defined and published (made available to the world at large), it cannot be changed. One of the rules of COM guarantees that software can rely on an interface remaining the same once it has been published. If you change an interface, you must publish it as another, new interface, so the previously published interface remains unchanged.

Because of this immutability, it is important for you to put some thought into your interfaces before creating them. If you do a good job of planning your interface, you will reduce the likelihood of needing to change the code that uses a particular component. As long as the interface does not change, the code that uses the interface does not need to change. If you do not plan your interfaces well, then it is more likely that they will change, requiring changes to the surrounding code as well.

Notes for Developers

If you are a VB developer, you are fortunate, because VB hides all the inner workings of COM interfaces from you.

However, if you are a VC++ developer, you will need to do some extra work, which we will explore in Chapter 22, "Building and Using COM Servers in VC++," when we discuss the Interface Definition Language (IDL) for VC++ components.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Characteristics of Interfaces

Because interfaces are essential for communicating with your components, it is important to understand the characteristics of interfaces and be aware of how they work when you code your applications, whether you are developing in VB or VC++.

There are four points to note when working with interfaces:

An interface is not a class. An interface is not a class in the normal definition of a VB or C++ class. While an object is created from a class, an interface cannot be instantiated because it carries no implementation. A class implements an interface, and that class must be instantiated for there to be an object that exposes the interface.

An interface is not an object. An interface is just a related set of functions and the method by which clients and objects communicate. When a client has access to an object, it has nothing more than an interface pointer through which it can access the functions in the interface. The pointer is opaque, meaning that it hides all aspects of internal implementation. You cannot see any details about the object, such as its state information. This is in contrast to C++ *object pointers*, through which a client may directly access the object's data. In COM, the client can call only functions of the interface to which it has a pointer. But instead of being a restriction, this is what allows COM to provide the efficient binary standard that enables *location transparency* (for more on this topic, refer to *Mastering COM and COM+* from Sybex).

Interfaces have strong types. Every interface has its own interface identifier (a GUID) that is unique to it. It is a guarantee that there will be no conflict with other interfaces.

Interfaces are immutable. Interfaces are never versioned, thus avoiding versioning problems. A new version of an interface, created by adding or removing functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore, a new interface does not conflict with an old interface, even if all that changed is the implementation.

If you are going to provide services through interfaces, it is important that your interface have these characteristics to ensure that the services being provided match the expectations of the users of these services. Understanding these characteristics will help you design better interfaces for your components.

Types of Interfaces

As noted earlier, IUnknown is the only interface that a COM component must implement. If your components will be accessed from a scripting language, you may also implement the IDispatch interface. These two interfaces, as well as custom and dual interfaces, are discussed in the following sections.

The IUnknown Interface

IUnknown is the most significant of interfaces. Every COM component must implement this interface. All COM objects must implement the IUnknown interface because it manages all of the other interfaces that are supported by the object.

The IUnknown interface contains three methods: QueryInterface, AddRef, and Release. These methods were developed for the VC++ programmer. In VC++, QueryInterface is used to find out about all of the available interfaces that an object exposes. When you want to use one of these interfaces, AddRef is called. When you are finished with that particular interface, Release is called.

NOTE:

VB hides interaction with the IUnknown interface. When we want to call an interface using VB, we declare an object. By typing a period (.) after the object name, we are presented with all the exposed interfaces. When we're finished using the object, we set it to Nothing, which is equivalent to using the Release method.

The IDispatch Interface

The IDispatch interface derives from the IUnknown interface. This interface was created primarily for the benefit of scripting languages. It is not recommended (due to the overhead) if you are developing applications that will never be accessed from scripting languages.

IDispatch consists of functions that allow access to the methods and properties of COM objects. IDispatch enables VB and other scripting languages to manipulate the properties and methods of an object. IDispatch has two important methods: Invoke and GetIDOfNames. The GetIDOfNames method receives the ID of an element to manipulate. The Invoke method gets the job done.

For example, suppose that you want to change the BackColor property of a label control. In VB, you call that label control, or write some code to search for that control, and then invoke the BackColor property on it. If you are a VC++ developer, invoking this function takes several lines:

```
DISPID PropList::GetDispIDFromName( LPDISPATCH lpDisp,
                                     LPWSTR strProperty )
{
    if( lpDisp == NULL )
    {
        OutputDebugString( _T("Bad Dispatch Pointer in"
                               " GetDispIDFromName.\n" ) );
        return FALSE;
    }

    DISPID dispid;
    LPWSTR lpPropName[1] = { strProperty };
    HRESULT hr = lpDisp->GetIDsOfNames( IID_NULL,
                                         lpPropName, 1,
                                         LOCALE_SYSTEM_DEFAULT,
                                         &dispid);

    return dispid;
}

BOOL PropList::GetValueFromDispID( LPDISPATCH lpDisp,
                                    DISPID dispid,
                                    VARIANT * va )
{
    DISPPARAMS dispParams = { NULL, NULL, 0, 0 };
```



```
HRESULT hr;  
VariantInit(va);  
  
    // if the above fails, so will this  
hr = lpDisp->Invoke( dispid, IID_NULL,  
                    LOCALE_SYSTEM_DEFAULT,  
                    DISPATCH_PROPERTYGET |  
                    DISPATCH_METHOD,  
                    &amp;dispParams, va, NULL, NULL );  
return SUCCEEDED(hr);  
}
```

Invoking this same function in VB takes only three lines:

```
If TypeName(Label1) = "Label" Then  
    Label1.BackColor = vbRed  
End If
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

Custom Interfaces and Dual Interfaces

A custom interface is an interface that the system doesn't already support. For instance, most OLE interfaces are defined by the operating system and have built-in marshaling support. If your interface is used only to talk to an in-proc server (a server that runs in the client's process), you don't need to provide marshaling code to package parameters between two process spaces. On the other hand, if your interface is a custom interface that is not supported by the operating system, and it might be used between two processes or machines, you need to provide marshaling code to transfer parameter data across process boundaries. Usually, this is a really tedious, arcane process; however, with the latest release of the Microsoft RPC MIDL (which stands for Remote Procedure Call Microsoft Interface Definition Language) compiler, you can automatically generate the necessary dynamic-link library (DLL) code by describing your interface with the IDL.

NOTE:

Marshaling code is discussed in *Mastering COM and COM+* from Sybex, which also covers DCOM (Distributed COM).

When exposing an object for Automation, Microsoft strongly recommends that objects provide a dual interface.

When an Automation object implements both an IDispatch and a vtable interface, it is called a *dual-interface* component. In a dual interface, the first three entries in the vtable are the members of IUnknown, the next four entries are the members of IDispatch, and the subsequent entries are the addresses of the members of the dual interface. All VB components support dual interfaces, so you don't have to do any additional work to implement a class that exposes a dual interface.

Figure 21.2 shows the vtable for an object that supports a dual interface named IInterface.

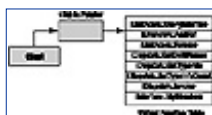


FIGURE 21.2 A vtable for an object with a dual interface

In addition to providing access to objects, Automation also provides information about exposed objects. By using IDispatch or a *type library*, a client or programming tool can determine which interfaces an object supports, as well as the names of its members. Type libraries are files or parts of files that describe the type of one or more COM objects. Type libraries are especially useful because they can be accessed at compile time. In VB, type libraries are combined into the same binary file.

What Are Type Libraries?

A type library is a binary file that describes your applications' object model. A type library is a file or a part of a file that describes the type of one or more ActiveX objects. Type libraries are typically embedded as a resource inside an ActiveX EXE or a DLL.

Type libraries do not store objects; they store type information. By accessing a type library, applications and type-library browsers can determine the characteristics of an object, such as the interfaces supported by the object and the names and addresses of the members of each interface. As we explain how to build components throughout this book, we will be discussing how to build type libraries and declare interfaces.

Interface Rules

Throughout this book, IDL syntax is used to describe interfaces written in VC++. Chapter 22 discusses in detail how IDL works and how it can be used to define interfaces. However, there are some general rules that you need to be aware of that apply to defining and building interfaces in both VC++ and VB.

Here, we list the rules Microsoft provides as part of the COM specification that are specific to the design of interfaces. We will dig into these rules as we build our components in VB and in VC++. We will also explore in more detail how each language applies and uses the rules to comply with the COM specification.

Design for Remote Use

COM gives you flexibility. When designing interfaces, you can specify any data type as an argument for a method. One implication of this is that interfaces can be designed so that it is physically impossible for them to be "remote." For example, an interface could take an argument that represents something that makes sense only within the context of the current process, like a Win32 graphics device interface (GDI) device context.

In general, all COM interfaces should be designed so that they support distributed processing. Later in this chapter, we will discuss the design of interfaces with this general rule in mind. However, there are circumstances where an interface will be used only in an in-process case, so you'll see that this rule can be bent for specific situations. For example, you can use the MIDL attribute `local` to enforce local-only calls.

Derive from IUnknown

All COM interfaces must derive directly or indirectly from the `IUnknown` interface. In other words, any interface implemented on a COM object must have `QueryInterface`, `AddRef`, and `Release` as its first three methods, in that order. Thus, when describing any COM interface in IDL, you will use syntax such as this:

```
import "unknwn.idl";

[ object, uuid(6612B9ED-DD34-12DE-5201-070042D13601) ]
interface ISomething : IUnknown
{
    HRESULT SomeMethod(void);
};

[ object, uuid(6612B9FE-DD34-12DE-5201-070042D13601) ]
interface ISomethingElse : ISomething
{
    HRESULT SomeOtherMethod([in] long l);
};
```

The interface `ISomething` contains four methods: `QueryInterface`, `AddRef`, `Release`, and `SomeMethod`. The interface `ISomethingElse` has five methods: all of the methods in `ISomething`, plus `SomeOtherMethod`.

Create Unique Identifiers

Remember that the “real” name of an interface is a 128-bit GUID, not its human-readable name. Thus, for each newly defined interface, a new *interface identifier* (IID) must be generated.

You can use either uuidgen.exe or guidgen.exe to generate a new GUID that will be your interface’s IID. UUIDGEN is a console application that is part of Microsoft RPC, and GUIDGEN is a Microsoft Windows application that is included with Microsoft VC++. These utilities are functionally the same, eventually calling the distributed computing environment (DCE) RPC runtime application-programming interface (API) UuidCreate. (UUIDGEN calls UuidCreate directly; GUIDGEN calls the COM API CoCreateGuid, which is simply a wrapper around UuidCreate.)

Notice that in the IDL example in the previous section, each interface has its own unique IID. By convention, symbolic constants used to identify a specific IID are of the form IID_<interface name>. Thus, for our example, IID_ISomething would represent the IID value 4411B7FE-EE28-11ce-9054-080036F12502.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Interfaces Are Immutable

After it is published, the interface contract associated with a particular IID can never change. As explained earlier, this rule guarantees that software can rely on an interface remaining the same once it has been published.

Functions Should Return HRESULTs

All methods in your interface should return values of type HRESULT. Note that this does not apply to IUnknown::AddRef and IUnknown::Release, which are exceptions.

NOTE:

HRESULT implies “handle to a result,” but this is not accurate; it is neither a handle nor a result. HRESULT is synonymous with SCODE (a term from the 16-bit days of OLE). So when you read HRESULT, just think “status code.”

While it is possible for COM interface functions to return types other than HRESULT, the interfaces you design should not do so. The reason is that the COM remote processes need to return RPC errors to the caller. If you define methods with return types other than HRESULT, COM has no way to tell callers of your methods that the server has crashed (RPC_E_SERVER_DIED) or that the network has gone down (RPC_S_COMM_FAILURE). AddRef and Release are specifically defined such that they *cannot* return errors, which explains why they are excepted from this rule.

Strictly speaking, for interfaces that are never intended to be used remotely, this rule can be ignored. However, even for interfaces that are intended to be implemented by in-process objects only and never passed across a process boundary, we recommend that you follow the rule for programming model consistency.

Defining HRESULT Values

The COM specification provides rules regarding the definition of new status codes

(HRESULT values). COM interface methods and COM Library API functions use a specific convention for error codes in order to pass back to the caller both a useful return value and an indication of status or error information. For example, it is highly useful for a function to return a Boolean result (TRUE or FALSE), as well as to indicate failure or success.

The HRESULT type (a 32-bit integer) is the medium through which these status codes are passed. An HRESULT has an internal structure comprised of four fields with the following format:

Field	Bit Position	Description
S	1 bit	Severity field: 0 Success: The function was successful; it behaved according to its proscribed semantics. 1 Error: The function failed due to an error condition.
R	2 bits	Reserved for future use; must be set to zero by current programs generating HRESULT values; current code should not take action that relies on any particular bits being set or cleared this field.
Facility	13 bits	Indicates which group of status codes this belongs to. New facilities must be allocated by Microsoft because they need to be universally unique. However, the need for new facility codes is very small. In most cases, you can and should use FACILITY_ITF.
Code	16 bits	Describes what actually took place, error or otherwise.

All facility codes, except FACILITY_ITF, are reserved for COM-defined failure and success codes. In other words, if you need to define failure or success codes that are specific to the interfaces you are designing, you must set their facility to FACILITY_ITF.

Status codes in FACILITY_ITF are defined solely by the creator of the interface. That is, in order to avoid conflicting error codes, a human being needs to coordinate the assignment of codes in this facility, and it is he or she who defines the interface that does the coordination.

Likewise, it is possible for designers of suites of interfaces to coordinate the error codes across the interfaces in that suite in order to avoid duplication. The designers of the OLE Documents interface suite, for example, ensured that codes were not duplicated.

All the COM-defined FACILITY_ITF codes have a code value that lies in the region 0x0000 through 0x01FF. Thus, while it is legal for the definer of a new interface to make use of any codes in FACILITY_ITF, we strongly recommend that you use code values only in the range 0x0200 through 0xFFFF, so that you reduce the possibility of accidental confusion with any COM-defined errors. We also strongly recommend that you consider defining as legal that most, if not all, of your functions can return the appropriate status codes defined by COM in facilities other than FACILITY_ITF. For example, E_UNEXPECTED is an error code that you will probably want to make universally legal.

String Parameters Should Be Unicode

All strings passed through all COM interfaces (and, on Microsoft platforms, all COM APIs) are Unicode strings. There simply is no other effective way to implement interoperable objects in the face of an architecture that provides call-location transparency and doesn't in all cases intervene in system-provided code between client and server.

Interface Design

The designer of an interface is responsible for documenting all of the information that describes the contract represented by the interface. The following elements must be included in a COM interface contract:

- Interface identifier
- Interface signature
- Interface semantics

These elements are described in detail in the following sections.

Interface Identifier

Each interface must have a GUID that serves as its programmatic name. It is this interface ID (IID) that uniquely identifies the contract defined by the interface. After an interface design with a particular IID is compiled into a binary form and released, the specifics of all the other elements that make up that interface cannot change.

Newcomers to COM may find the previous statement to be too strong. However, the immutability of an interface contract is fundamental to the power and robustness of COM. If an interface design turns out to be insufficient or faulty in some way after it is published, a completely new contract must be drawn up; that is, a new IID must be generated.

An interface should have a *human-readable* name; that is, it should be given a name that indicates the services the interface exposes. By convention, interface names begin with the capital letter *I*. Assigning human-readable names to interfaces, methods, and method parameters is technically optional; however, including them makes the life of human programmers significantly more pleasant.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Interface Signature

The *interface signature*, also called the *interface syntax*, provides the user of an interface with enough information to push parameters onto the stack and determine which offset into the vtable to use as the method address. Specifically, the interface signature defines the following:

- The number and order of methods in the interface
- The number, order, and type of each parameter for each of the methods
- The return value type for each of the methods

For parameters, the type includes whether the parameter is an *in*, an *out*, or an *in-out* parameter. The interface signature also includes type definitions (for example, structures) used in the interface and calling conventions (cdecl, Pascal, __stdcall, and so on).

The values of any constants associated with the interface (including failure and success codes) are also part of the signature, but the *meanings* of those constants are part of the semantics. Strictly speaking, each method in an interface could legally have a different calling convention. However, calling conventions are very platform-specific, and, in terms of binary interoperability, a calling convention is important only on a per-platform basis. Thus, it is highly recommended that interfaces adopt the default calling convention for the platform being used. Do this by simply not specifying a calling convention.

Interface Semantics

Just looking at the names of methods and method parameters can often give callers clues about what function the methods perform when called, but callers usually can't glean enough information in this manner. The definitions of the interface are the parts of the contract that the signature alone cannot describe.

Interface semantics include descriptions of the behavior of each method, the context and order in which the method can or should be called, the failure codes specific to the method, and the possible success codes. Documenting the semantics of a COM interface is no different than documenting the definitions of a set of related C APIs. You should document the definition of each method that makes up the interface, just as you would document any other API function, and then follow some simple rules to bring all your method definitions together into one interface definition.

Implementation Rules for COM Components

In order to build COM components, the code you write must follow some well-defined rules. As we mentioned earlier in the chapter, COM is a Microsoft standard. You must follow this standard in order to implement COM in your code. Although the standard applies for every language that wants to create COM components, the steps taken to ensure that the requirements are met can be quite different from language to language. Here, we list the rules that apply to COM implementation for VB and VC++. These rules will be discussed in detail in later chapters.

The Implement IUnknown Rule

An object is not a COM object unless it implements at least one interface, and that interface is IUnknown. You will learn how your code can implement IUnknown in VB and in VC++ in later chapters.

Memory Management Rules

Although VB and VC++ manage memory in different ways, some basic concepts apply to both. Managing memory is managing pointers. The lifetime management of pointers to interfaces is always accomplished through the `AddRef` and `Release` methods found in every COM interface. There are some rules that apply to parameters to interface member functions. There are three types of parameters:

In parameter For in parameters, the caller should allocate and free the memory.

Out parameter Out parameters must be allocated by the callee and freed by the caller using the standard COM memory allocator.

In-out parameter In-out parameters are initially allocated by the caller, then freed and reallocated by the callee if necessary. As with out parameters, the caller is responsible for freeing the final returned value.

What if a function returns a failure code? In general, the caller has no way to clean up the out or in-out parameters, which leads us to a few more rules:

- In error returns, out parameters must be set to a value that will be cleaned up without any action on the caller's part.
- All out pointer parameters must explicitly be set to `NULL`.
- In error returns, all in-out parameters must either be left alone by the callee or be explicitly set as in the out parameter error return case.

Reference Counting Rules

Reference counting is how applications keep track of the COM instances they have to ensure compliance with the COM specification. VB handles reference counting internally, thereby hiding it from developers. VC++ explicitly requires developers to maintain this functionality.

NOTE:

When we discuss COM+, you will see how this model changes under VC++ to make it work similarly to the way VB components are implemented.

The following reference counting rules apply:

- In VC++, to ensure proper reference counting, `AddRef` must be called for every new copy of an interface pointer, and `Release` must be called for every destruction of an interface pointer. There are some exceptions to this rule that we will explore when discussing components in VC++.
- From a COM client's perspective, reference counting is always a per-interface concept. Clients should never assume that an object uses the same reference count for all interfaces.
- You should rely on the return values of the `AddRef` and `Release` only while debugging, never in deployment or to make code decisions.

COM Activation

So how do clients activate and call the COM interfaces of other components? Well, in the case of VB, when you create a public class, the VB runtime creates a default interface for that class. Just by declaring that public class, your class is now COM compliant. When clients want to create a COM object in a server component,

you usually just add a reference to the server and call the interfaces. VB shields you from a lot of activation that happens behind the scenes. Let's take a look at what happens when you invoke a COM object.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

COM will cache pointers to objects that are loaded into memory already and, as long as you request the same threading model, it will load another instance from there. If it can't find a cached instance that can accommodate the request, it will try to load an in-proc server (if requested). If this fails, COM will then contact the Service Control Manager (SCM). The SCM is a COM service that handles COM activation, using the following procedure:

1. When clients first request a server COM object, they pass the object ID to SCM.
2. SCM finds and loads the server by getting this information from the Registry.
3. SCM asks the server component to create the object you requested.
4. SCM takes the reference to the interface of the object and returns it to the client.
5. After SCM returns the reference to the client, SCM's job is complete. The client and server are now ready to interact.

There is an additional piece of the activation process that you should be familiar with, even though you don't need to worry about implementing or calling it. It is something that happens behind the scenes every time you declare an object as new. When SCM asks the server component to create a new object, it is actually asking something called a *class factory* to create an uninitialized instance of the object class. Although this might sound interesting, as a VB programmer, you have no direct access to the class factory, and don't need to implement it or work directly with it. However, it is important to know that it is there and how it works. We will discuss the class factory in detail in later chapters.

Types of COM

The three different ways that you can package COM objects are as clients, as servers, and as ActiveX controls. Here, we will briefly discuss each of these methods. Upcoming chapters describe in detail how to build and use the various COM types.

COM Clients

A COM client is an application or programming tool that manipulates one or more COM objects. The objects can exist in the same application or in another application. Clients can use existing objects, create new instances of objects, get and set properties, and invoke methods supported by the object.

VB is a COM client. You can use VB and similar programming tools to create packaged scripts that access Automation objects. You can also create clients by doing the following:

- Writing code within an application that accesses another application's exposed objects through Automation
- Revising an existing programming tool, such as an embedded macro language, to add support for Automation
- Developing a new application, such as a compiler or type information browser, that supports Automation

COM Servers

COM objects can exist in the same process as their controller, or they can reside in a different process.

- *In-process server objects* are implemented in a DLL and are run in the process space of the controller. Because they are contained in a DLL, they cannot be run as stand-alone objects.
- *Out-of-process server objects* are implemented in an executable file and are run in a separate process space.

Access to in-process objects is much faster than to out-of-process server objects because Automation does not need to make remote procedure calls across the process boundary. This is true if you are limited to VB.

NOTE:

In-process server objects actually can run as stand-alone objects if they are running under a surrogate process (either the default surrogate or a custom surrogate). From the client's perspective, they could create the DLL-based object out-of-process without concern. (Surrogate processes are discussed further in *Mastering COM and COM+* from Sybex.)

The access mechanism (IDispatch or the vtable) and the location of an object (in-process or out-of-process server) determine the fixed overhead required for access. The most important factor in performance, however, is the quantity and nature of the work performed by the methods and procedures that are invoked. If a method is time-consuming or requires remote procedure calls, the overhead of the call to IDispatch may make a call to the vtable more appropriate.

ActiveX Controls

An ActiveX control is implemented as an in-process server that can be used in any OLE container, such as VB. The difference between an ActiveX control and a COM in-process server is that ActiveX controls generally have a user interface. This means that the full functionality of an ActiveX control is available only when used within an OLE container designed to be aware of ActiveX controls.

An ActiveX control uses several programmatic elements to interact efficiently with a control container and with the user. ActiveX controls are discussed at some length in *Mastering COM and COM+* from Sybex, including how to build ActiveX controls both in VB and VC++ using the Active Template Library (ATL) and how to take advantage of the container-provided functionality.

COM and Object-Oriented Techniques

By now, you are beginning to understand COM from a high-level perspective. You might even be thinking that you are ready to write some code. However, there is one rather large topic that we need to explore before we actually start to create programs. We need to answer these questions: How do COM and object-oriented programming (OOP) fit together? What does an object have to do with COM? Since you already know how to program objects, how will this be different?

One of the most confusing aspects of COM is how objects fit in the picture and what OOP has to do with COM. These concepts are even more confusing when the words *component* and *object* are used interchangeably. Objects, as we know them today without COM, are instances of a class at runtime. You cannot work with an object at design time; you can work only with classes. An object can be called a COM *object* if it follows some of the COM rules that we mentioned in previous sections. A collection of objects that provides a set of functionality in a compiled form is called a *component*. So, a component is a collection of objects that follow the COM rules and that are compiled to a binary form.

NOTE:

In the case of VB, all the COM rules are handled transparently for you, so every object in VB is a COM object. Unfortunately, this is not the case for VC++ objects; you need to do some work to make them COM objects.

So, what do OOP techniques have to do with COM? Well, simply put, nothing. Using OOP techniques does not make COM happen or function better. However, OOP techniques can help us concentrate on the business problems and come up with solutions in the form of objects that we can implement as COM components.

Most OOP techniques are implementation techniques, meaning that they work on classes and code. The objective is to combine the best of both worlds by using OOP techniques when implementing objects and also making them COM objects by following the COM rules. The challenge is not to break the rules of OOP or COM.

There are four main concepts involved in an OOP system:

- *Encapsulation* is the concept of containing properties and behavior together and exposing only interfaces to use these properties and behaviors.
- *Abstraction* is the concept behind information hiding. Abstraction helps in improved representation of real-world objects.
- *Polymorphism* is the concept of using an interface in different situations to perform different functions.
- *Inheritance* is the concept of reusing functionality without needing to reimplement it.

In the following sections, we will look at how VB specifically implements some OOP techniques and how we can take advantage of these features to build well-designed COM-based applications.

NOTE:

The subject of OOP and VB has long been discussed—and even exhausted at some point. We are not here to argue whether VB is a pure OOP language. There are some areas where VB has no support for OOP features, but there are also other areas where it does provide OOP support.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Encapsulation

Encapsulation is the idea behind the *black box* approach to building objects. The concept is that objects should separate their implementation details from their interfaces. So, when you examine an object, you know what the object can do, but you can't tell how it does it. This means that an object should completely contain any data it needs and it should also contain the code that it needs to manipulate that data. COM interfaces hide this implementation detail—when you examine an object, all you see are interfaces and not implementation details.

As we discussed earlier, programs interact with an object through an interface, using properties and methods. Clients that use your objects and components will never work directly with the data owned by those objects. The interaction between clients and objects is done by sending requests to the object. An object can also receive requests from other objects. The way the object reacts to these requests is through methods and properties.

VB 6 (and some earlier versions) provides full support for encapsulation through class modules. You can then create classes, which entirely hide their internal data and code. The same is also true for VC++ classes. Classes provide a well-established interface of properties and methods to the outside world.

Abstraction

Abstraction is the technique of isolating complexities and focusing on the necessary. Abstraction is a guideline. A map analogy can help clarify this concept. A map gives you direction by pointing out the major areas that you need to know about. It does not give you all the details of the street, the cars parked on it, or the pedestrians. If you need that level of direction, you must get it somewhere else. COM helps by bringing the concept of abstraction to life. You can say that COM made abstraction real.

Encapsulation provides the explicit boundary between an object's interface and its implementation details. Encapsulation puts these details into a "capsule." Encapsulation provides OOP developers the freedom to implement abstraction in any way consistent with the interface. Abstraction provides business value; encapsulation protects abstraction.

If you provide good abstraction, users won't be tempted to peek at the object implementation. There is nothing more frustrating than lousy abstraction that is encapsulated. When a developer encapsulates bad

abstraction, users will continually attempt to violate the abstraction barrier.

Providing good abstraction is the key to software reuse. You simply can't reuse something that is bad.

Polymorphism

Polymorphism is the feature that allows programmers to use the same method name for many different objects. Each object can have its own method of any given name. When a method is called, the method code in the invoked object will be used.

Each method is a part of an object. It is possible to have the same method available in different objects. For example, if the method `GetBalance` is part of the `General Account` class, every checking account and savings account will have this method.

It is possible, and often desirable and necessary, to write a program that uses accounts (checking, savings, loans, IRAs) without knowing which particular class the account belongs to. The ability to write general code like this, calling a method without knowing in advance which object is being called, is what polymorphism is all about.

COM interfaces can be derived from other COM interfaces using single-interface inheritance. In fact, all COM interfaces, directly or indirectly, inherit from `IUnknown`. Besides `IUnknown`, there are only two other important base interfaces that are commonly inherited from: `IDispatch` and `IPersist`. Otherwise, interface inheritance in COM is rarely used. Surprisingly, interface inheritance in COM has nothing to do with the polymorphism COM supports.

For example, assume that a client holds a reference to an interface, `IDispatch`. In reality, the interface the client refers to can be of any subtype of `IDispatch`. In other words, the vtable may contain additional methods over and above those required by `IDispatch`. However, there is no way for the client to find this out. If the client wants a more specific interface, it must use `QueryInterface`, the mechanism that clients use to find out about the features that the object supports.

If you are using VC++, you need to write code to query for object interfaces, such as the following line:

```
HRESULT QueryInterface(REFIID riid, void **ppvObject);
```

In VB, you don't need to write any code to query for object interfaces. VB automatically and transparently does this by calling `QueryInterface` when you do an object assignment, such as the following VB code:

```
Dim IMyNewObject as ITheMotherObject  
Set IMyNewObject = Something  
IMyNewObject.Method
```

Why is it important to know this? Because `QueryInterface` is how versioning happens in COM. When you create interfaces for your components and then make changes to your components by adding or removing interfaces, clients can query for those interfaces.

Polymorphism and Late Binding

We've discussed *binding* when calling a polymorphic method, which is just another way of calling methods that imply the same thing but are implemented on different objects. Let's say we created a class called `CDataManipulation` with methods such as `Save`, `Delete`, and `Update`. Different objects such as `Customer`, `Vendor`, and `Product` would react to these methods in a different way, because the implementation for each of these methods is different. Let's examine the following routine:

```
Public Sub Save(AnyObject As Object)  
    AnyObject.Save Me  
End Sub
```

This code will run whether you pass it the `Customer`, `Vendor`, or `Product` object, simply because it accepts the `Object` parameter, and it will figure out which object's `Save` it should call. This mechanism is referred to as *late binding* or `IDispatch` binding.

With `IDispatch` binding, the client has no preexisting information about the object. During compilation, it assumes that the code you are calling is correct. It then attempts to execute the code at runtime and trap for

runtime errors. This is a step backward from the familiar syntax and type checking we have come to expect from VB.

An IDispatch binding also requires both the client and the server to marshal their arguments into a consistent form that is understood by both. In most cases, this means that arguments are copied into and out of variants. This not only slows down IDispatch, but it also limits the data types that can be passed using this technique.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Polymorphism and Early Binding

Another technique that is much more efficient than late binding is called *interface inheritance using early binding*. As explained earlier, you can access interfaces either through IDispatch or directly through the vtable. If Visual Basic can tell at compile time what object a property or method belongs to, it can look up the vtable address of the member in the type library.

Early binding requires type information provided in the form of a type library. This type information allows VB to perform compile-time syntax and type checking. At runtime, this type of binding is faster because the entry points for the objects are already known and the data types and syntax have already been verified. The key to using this technique is a keyword in VB called Implements. The next section shows an example of early binding.

Inheritance

Inheritance has always been associated with reuse. Most developers who want to reuse some piece of functionality and have worked with pure OOP languages usually think of inheritance in the form of *implementation inheritance*—including the one class implementation into their own class implementation. VB does not support implementation inheritance, but VC++ does. When we discuss VC++ programming in Chapter 22, you will learn how to use implementation inheritance and COM.

Through COM, VB supports a different type of inheritance called *interface inheritance*. The Implements keyword is the key to interface inheritance. The Implements keyword causes the class to accept COM QueryInterface calls for the specified interface ID. VB does not implement derived classes or interfaces. When you implement an interface, you must include all the public procedures involved. A missing member in an implementation of an interface or class causes an error. If you don't place code in one of the procedures in a class you are implementing, you can raise the appropriate error (Const E_NOTIMPL = &H80004001) so a user of the implementation understands that a member is not implemented.

Interface inheritance lets you inherit the interface of one class—not the implementation of the object—into a new class. Although this is a great feature, one of the disadvantages of interface inheritance is that COM does not allow you to extend that interface. However, until VB supports full inheritance and we can inherit implementation, as in other OOP languages, we will have to live with interface inheritance.

As an example, let's see how we can apply the Implements keyword and inherit the interfaces of a component called Account. If we look at the Account class, we'll find an interface that includes elements appropriate for any type of account. So let's design a class with an interface that we can use in different types of accounts, such as a checking account, a savings account, and any other type of accounts we add to the system.

Option Explicit

Public Property Get AccountBalance() As Long

End Property

Public Property Get AccountStatus() As String

End Property

Public Property Get AccountRating() As Integer

End Property

You will notice that we did not add any implementation code inside these properties, meaning that we did not specify what happens when you call the AccountBalance property. We don't need to write implementation code because we are inheriting the interface.

Now we can create a new class, Checking, and inherit the interface from the Account class by using the Implements keyword.

Option Explicit

Implements Account

Private Property Get Account_AccountBalance() As Long

 Account_AccountBalance = 1235

End Property

Private Property Get Account_AccountStatus() As String

 Account_AccountStatus = "Active"

End Property

Private Property Get Account_AccountRating() As Integer

 Account_AccountRating = 3

End Property

Instead of declaring AccountBalance, AccountStatus, and AccountRating directly, we need to make them private in scope and put Account_ in front of each name so VB knows that these routines belong to the Account interface.

We might also create a Savings class, again based on the Account interface:

Option Explicit

Implements Account

Private Property Get Account_AccountBalance () As Long

 Account_AccountBalance = 15986

End Property

Private Property Get Account_AccountStatus () As String

 Account_AccountStatus = "Active"

End Property

Private Property Get Account_AccountRating () As Integer

 Account_AccountRating = 5

End Property

You can now write code that uses these two classes in this elegant way, accepting two types of accounts and summing them up:

```
Public Function GetTotalBalance( Checking As Account,
                               Savings As Account )
GetTotalBalance = Checking.AccountBalance +
                  Savings.AccountBalance
End Function
```

Here, we've created different classes, both having inherited the same interface from Account. This technique allows us to write client code that treats all the objects the same, even though they have different implementations.

COM+

At this point, we want to switch gears and talk about a new addition to COM called COM+. The addition of the plus sign may bring to mind C and C++, but the relationship between COM and COM+ is different from the relationship between C and C++. You can jump in and start learning C++ without any prior knowledge of C. However, to understand COM+, you must know COM and how to use it.

COM+ and Windows DNA

COM is a programming model. COM+ is a set of services that complements the programming model. COM+ provides you with the services necessary to build distributed applications.

To understand what COM+ is, you need to understand Microsoft Windows Distributed interNet Applications Architecture (Windows DNA). Windows DNA is a framework that enables developers to build distributed applications using technologies and services that are part of the Windows operating system. Windows DNA provides design guidelines to help developers make the right choices when creating new software applications.

The Windows DNA model is intended for building different types of applications from a very thin "reach" client, which was designed to be able to reach users connecting over the Web, to very "rich" clients running on a network topology or an intranet. Windows DNA is aimed at allowing developers to focus on solving business problems rather than needing to handle the technology infrastructure.

The Windows DNA model targets the three tiers of applications:

- User interface (UI) tier, addressed by the Forms+ technology
- Middle tier, addressed by the COM+ technology
- Database tier, addressed by the Storage+ technology

NOTE:

At the time of this writing, only the COM+ technology is available. The Windows DNA technologies to address the other tiers are still under development, and the names Forms+ and Storage+ may be replaced by other terms.

Although we are focusing on the middle tier, which is COM+, let's take a quick look at the other tiers to see how they all fit together. Keep in mind that many of the technologies for Windows DNA still need to be developed and embedded into development tools, the Windows operating system, and databases.

[Previous](#) [Table of Contents](#) [Next](#)

HOME

SUBSCRIBE

SEARCH

FAQ

SITEMAP

CONTACT US



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The User Interface Tier Technologies

Developing different clients requires using different types of technologies. For example, if you are developing rich desktop applications, your choices for tools are forms combined with ActiveX controls and whatever flashy components you like. Being bound to a certain technology to achieve your goal can be dangerous. What if you decide to migrate your application to a web client?

The Forms+ initiative promises to deliver a robust solution for building different types of clients without crippling your functionality. Tools to develop Forms+ are not yet available, unless you count the Windows Foundation Classes (WFC) attempt in the latest version of Visual J++, which comes close to fulfilling this promise. Future releases of development tools will probably focus on meeting the Forms+ initiative.

The Middle Tier Technologies

COM+ is what the middle tier is all about. The middle tier has been the primary focus for Microsoft for the past few years. Before Windows NT 4, the primary focus for Microsoft's development tools was the user interface and how to build applications rapidly and effectively. As the use of these tools and adoption of the RAD model increased, more and more concerns arose about the lack of support for the middle tier. Microsoft listened to these cries for help and presented Microsoft Transaction Server (MTS), the first tool to offer assistance in the middle tier. The main focus of MTS was to manage transactions for developers, thereby allowing them to focus more on the business problem and less on the infrastructure needed to support transactions. MTS also assisted developers with managing server resources.

Because MTS was such a success, Microsoft developers decided to look at other areas in the middle tier where developers spend a lot of time and see what other support they could provide. The second tool was Microsoft Message Queue (MSMQ), which supports message-based applications and disconnected users.

Again, with the success of this tool in solving business problems, the trend to provide middle tier support continued. At this point, Microsoft realized that developers were relying on these support features, so why not make them a core part of the operating system? Then these services would not need to be shipped with the applications. That is what COM+ is all about—providing infrastructure support for middle-tier services.

COM+ has been referred to as many things. In fact, we've heard it said that COM + MTS = COM+. The main point is that COM and COM+ are two separate entities, even though you'll hear them referred to as the same

thing. COM+ is the consolidation of MTS and all the other services designed to support building a distributed application.

The Database Tier Technologies

Storage+ is the least-discussed feature of Windows DNA. Information about the database tier technologies under development is not readily available.

There are some who speculate that version 2 of COM+ and Microsoft SQL Server will converge in the area of object persistence. Until these initiatives are clarified, we will need to rely on ActiveX Data Objects (ADO) and OLE DB (see Chapter 18, “ADO Database Access”). These are currently some of the most important technologies for Microsoft’s long-term database strategy.

The Component Services Snap-In

With the Component Services Explorer tool, shown in Figure 21.3, you can configure and administer COM components and applications. The Components Services Snap-in, also referred to as COM+ Explorer, is going to become your new best friend.

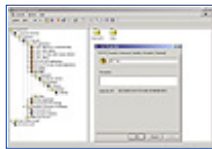


FIGURE 21.3 The Component Services Snap-in

This is where you will perform the majority of all your component configurations, such as the following tasks:

- Configure the system for Component Services. After you install Component Services, there are a few tasks you must perform before you can use the Component Services administrative tool.
- Install COM+ applications. After making settings specific to the services your COM+ applications use, you will deploy client-side and server-side portions across your network.
- Configure load balancing. When your application’s workload is distributed through load balancing, administrative tasks might include defining participating computers and components, monitoring activity, and responding to problems.
- Configure security. Depending on the type of application you are administering, you may need to set security attributes.
- Configure distributed transactions. If your applications use distributed transactions, you can configure your system to use Microsoft Distributed Transaction Coordinator (MS DTC), view transaction statistics, and resolve transaction states.
- Configure Queued Components (QC). You can configure components to take advantage of a Component Services feature that enables COM components to request services from another component, even when that component is unavailable or offline.
- Configure and set up IMDB. You can configure COM+ applications to use a feature that provides fast access to databases stored in main memory.

You also can use the Component Services administrative tool to configure routine component and application behavior, such as security and participation in transactions. We will revisit each of these areas when we discuss these features and component development in more detail in the upcoming chapters.

In addition to the Components Services Snap-in, you will be using the Computer Management Explorer, shown in Figure 21.4. With this tool, you can configure services as well as security for users and groups.

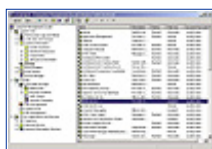


FIGURE 21.4 Computer Management Explorer

Transactions

COM+ components that have transaction capabilities provide a wrapper around MS DTC transactions in the form of COM+ transactions. In your code, you never explicitly start, commit, or abort the physical (MS DTC) transaction (unless you are using manual transactions). Instead, you tell COM+ what it should do to that transaction. When an object is created from a transactional component, COM+ will automatically create a COM+ transaction. When the object is activated (a method is called), COM+ will create a physical transaction (an MS DTC transaction). When the object is deactivated, COM+ calls Commit or Abort on the physical transaction, depending on what your object told it to do. To influence the transaction's outcome, you call SetComplete or SetAbort on the MTSObjectContext within your method execution.

You can control transaction creation through transaction property settings. When you install a component under COM+, you have the option to set its transaction property to one of the following settings:

Requires Transaction	This component must be run within a transaction. If the calling object already has a transaction, this object will inherit the same transaction. Otherwise, MTS or COM+ will create a new transaction.
Requires New Transaction	This component must be run in a transaction by itself. MTS or COM+ will always create a new transaction.
Supports Transactions	This component may be run within or outside a transaction. If the calling object has a transaction, this object will inherit the same transaction. Otherwise, this object will not have a transaction.
Does Not Support Transactions	This component cannot be run inside a transaction. If the calling object has a transaction, MTS or COM+ will run this object outside the context of that transaction.
Disabled	This component becomes a non-COM+ object and a context is not created.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

As an example, suppose that you have three objects, named ObjectA, ObjectB, and ObjectC, with the following settings:

ObjectA: Requires Transaction

ObjectB: Requires New Transaction

ObjectC: Supports Transaction

If the client calls ObjectA, which in turn calls ObjectB and ObjectC, the transaction layout will be as shown in Figure 21.5. As you can see from the figure, COM+ creates a transaction for ObjectA and a separate transaction for ObjectB (because it requires a new transaction). ObjectC is activated within the same transaction as ObjectA. In this scenario, ObjectA is referred to as the *root object* and ObjectB and ObjectC are called *secondary objects*.



FIGURE 21.5 A multiple-object transaction

To understand how multiple-object transactions work, let's walk through an example. When a client calls component ObjectA, COM+ starts a new transaction, with ObjectA as the root object. Next, ObjectA calls ObjectB, which requires that a new transaction be created. If ObjectB succeeds, it calls SetComplete and exits. COM+ deactivates ObjectB and commits its transaction. Next, ObjectA calls ObjectC. If ObjectC encounters some major difficulties while performing its work, it calls SetAbort and exits with a return code indicating that an error occurred. Since ObjectA is still running, MTS or COM+ will simply take note of the fact that ObjectC called SetAbort. However, the physical (MS DTC) transaction is not yet aborted. Execution is returned to ObjectA, which detects that ObjectC failed and decides to return an error to the client and exit. COM+ deactivates ObjectA. Since ObjectA is the transaction's root object, COM+ will now abort the physical transaction. This mechanism (called a *voting mechanism*) means that any secondary object may affect the overall outcome of the transaction.

We will go into more detail about programming COM+ transactions in the following chapters.

Queued Components (QC)

QC is a key feature of COM+. QC provides an easy way to invoke and execute components asynchronously. QC is based on MSMQ. However, MSMQ requires knowledge of marshaling to queue, send, and receive asynchronous messages. QC marshals data automatically in the form of an MSMQ message. QC also offers built-in support for transactions.

QC enhances the COM+ programming model to provide an environment in which components can be invoked either synchronously (real-time) or asynchronously (queued), where the component does not need to be aware of this environment.

Limitations of Real-Time Architectures

There are several limitations to “real-time” architectures. The most significant has to do with the availability of the system with which you are communicating.

A component that you want to communicate with may not be available for various reasons, such as network problems, server overload, or because the component only comes online at a certain time of the day. If a component is not available, the process cannot complete.

The requirements of real-time servers are high and difficult to balance. Often, systems are underutilized most of the time and are overutilized at other times, depending on the peak usage of the system.

Another limitation of real-time systems is the lack of control over the processing of the calls. The real-time systems’ approach is “first come, first served.” There is no notion of priority, where you can control the order of processing.

Transactional Message Queuing

In a single application, you might choose to adopt several architectural approaches to solving problems. In line-of-business applications, transactional integrity is critical for both connected and disconnected systems.

There are many application situations where a business transaction can be split into several smaller transactions that can be separated in time and can be connected via reliable messaging. Using this approach, the client adds a message to a queue and commits that small transaction. A server (running at a later time) gets the client’s message from an input queue, performs updates to server resources, adds a message to a client response queue, and commits those operations in one transaction. The client retrieves its response message in a third transaction. Figure 21.6 illustrates this approach.

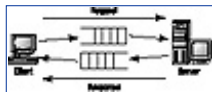


FIGURE 21.6 A transactional request in action

In this example, immediate notification from the server component to the client is not necessary. However, the eventual notification and delivery of the message are required. This model serves mobile clients very effectively.

Queued Components Architecture

To invoke a queued component, you use the new moniker, which creates object instances directly or passes a CLSID to the queue moniker. The queue moniker instantiates the QC Recorder for a given CLSID.

Several other components make up QC:

- The Recorder, for the client or send side
- A Listener, for the server or receive side
- A Player, for the server or receive side
- The Message Mover administrative tool, which moves messages from one queue to another

Figure 21.7 illustrates the QC components.



FIGURE 21.7 The dynamic view of a client and server QC interaction where the Message Mover tool

transfers messages from one queue to another

Runtime Architecture

The overall objective of QC is to provide a remoting mechanism for COM interfaces that permits the remote server object to be executed at a later time than the client invocation. The existing DCOM remoting mechanism (RPC) records a method call on the proxy (client) side and plays back the method call on the stub (server) side immediately. In contrast, the QC remoting mechanism makes a recording of all the method calls for an object and does not play them back until after the client component completes its use of the server object and the transaction that recorded the method calls reaches normal completion (commits).

A representation of the recording is presented to MSMQ as a message to be sent to a server. MSMQ participates in the client-side transaction so that the sending of the message representing the recorded object participates in the same transaction as other resource managers invoked by the client. Therefore, MSMQ accepts the message for delivery only if the transaction commits.

QC requires that its application methods contain only “in” parameters and no significant return values. The server object isn’t necessarily available when the client makes its method calls. Server results may be returned by sending another message, perhaps to the client, that instantiates another object. Thus, two-way communications are performed, when required, by a series of one-way messages resulting in object invocations.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

QC Server Component

To design a queued component, you must follow the rules and guidelines to ensure that your component can be marked as queued. The code that follows shows a simple interface that does not contain any out parameters that return data to a client (although the data will be returned directly to the client).

```
Public Sub GetData(ByVal sSql As String)
    Dim rs As ADODB.Recordset
    Dim cn As ADODB.Connection
    Set cn = New ADODB.Connection
    Set rs = New ADODB.Recordset
    Dim sConnStr As String
    sConnStr = "Provider=SQLOLEDB.1;Persist Security
        'Info=False;User ID=sa;Initial Catalog=Northwind;
        òData Source=MyServer"
    cn.Open sConnStr
    rs.Open sSql, cn, adOpenStatic, adLockReadOnly
End Sub
```

NOTE:

This example doesn't show where we are storing the result of this method so that it can be retrieved later by the client. This is just a code snippet and is not complete. (The complete code example appears in *Mastering COM and COM+*, from Sybex, which covers COM+ services for VB.)

When we register this component under COM+, we will need to mark the component, as well as the interfaces that we want to queue, as queueable.

QC Client Component

The client code for a queued component is quite simple. All you need to do is create this component with the GetObject call using the new queue moniker.

```
Dim oQCS As QCServer.CServer
Set oQCS = GetObject("queue:/new:QCServer.CServer")
oQCS.GetData "Select * from Customers"
```

If the component was marked as queueable, your call will go through and will be queued. If the server component does not meet the QC requirements, you will receive an Automation error.

Queued Components Failure Recovery

Because the communication between the client and server components can occur in a disconnected environment, there must be a mechanism to handle failure and maintain the integrity of the application. The Listener and Player components together deal with “poison messages.” If a transaction being played back aborts, then MSMQ, as part of its transaction-abort process, moves the input message back to the input queue. The Player aborts if it receives a failure HRESULT from the server component being played to or if it catches an exception. If the server object repeatedly aborts, the server might loop continuously, performing these steps:

1. The Player de-queues the message.
2. The Player instantiates the object.
3. The Player suffers a rollback.
4. MSMQ puts the message back on the top of the queue.

Server-Side Retry Queues

QC handles transaction failure by using a series of application-specific “retry” queues. There are five queues for each application:

Normal input queue	This is the first queue into which the message is initially inserted.
First retry queue	Messages are moved here if the transaction aborts when processing messages from the normal input queue.
Second retry queue	Messages are moved here if the transaction repeatedly aborts when attempted on the first retry queue. A message may be retried three times on this queue before being moved to the tail of the next queue.
Third retry queue	Messages are moved here if the transaction repeatedly aborts when attempted in the second retry queue. A message may be retried five times on this queue before being moved to the end of the final queue.
Final resting queue	Messages are moved here if the transaction aborts repeatedly when attempted on the third retry queue. This queue is not serviced by a queue Listener. Messages stay here until they are manually moved (using the QC Message Mover utility) or are purged by the MSMQ Explorer.

The Player issues a *loosely coupled event (LCE)* when each transaction aborts. It issues another event when a message is moved from one queue to another. It issues yet another event when a message is deposited into the final resting queue.

NOTE:

LCEs are part of a component-based eventing system that uses a publish and subscribe model. A publisher publishes events to subscribers. Subscribers can subscribe to any published events. A publisher is any COM+ component calling an event object. A subscriber is any COM+ component that implements the methods of the event interface.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The message may be modified (augmented with diagnostic or original sender information) before being moved from queue to queue. This mechanism might be used to keep track of the number of retries at each level. The security mechanism permits a message to be moved to retry queues and then reinserted into the application's initial input queue.

The application server object may optionally have an associated exception handler that implements the `IPlaybackControl` interface that the Player calls before playing back any method calls. This interface is intended to inform the exception handler object that it is about to be placed on the final queue. The exception handler object may implement an alternative behavior, such as gathering problem-diagnosis information or generating an object or message that informs the client of a consistent and serious problem.

If the application does not implement the `IPlaybackControl` interface, the poison message is placed in the final resting queue when its retries have been exhausted. If the exception handler does implement `IPlaybackControl`, the exception handler's methods are called one last time. If it aborts this final time, the poison message is placed in the final resting queue.

The Client-Side Dead Letter Queue

At the client side, it may be impossible for MSMQ to move a message to its destination queue. This can happen, for example, if the queue access controls do not permit the message to be moved from the client to the server. In this case, the message will be moved to the client-side queue manager's Xact Dead Letter Queue (DLQ).

A QC-provided DLQ monitor instantiates the `ExceptionClass` and issues `QueryInterface` for the `IPlaybackControl` interface. If that succeeds, the DLQ monitor will invoke the `IPlaybackControl::FinalClientRetry` method, and then play back the method calls to the client-side implementation of the server object. That object might preserve diagnostic information or take some action to reverse the effect of a prior transaction. If the playback commits, the message is removed from the Xact Dead Letter queue. If the playback aborts or the required CLSID and interface are not available, the message remains in the Xact Dead Letter queue.

QC Security

QC objects support role-based security semantics as other COM+ objects. However, QC objects do not support impersonation-style security.

Consider a client call to a QC component X. This call is actually made to the Recorder, which packages it as part of a message to the server machine. The Listener dequeues the message and passes it to the Player. The Player then invokes the actual server component X and makes the same method call. The server component X must observe the security call context of the client (*not* the Player's security call context) when the Player makes the method call. The Recorder marshals the client's security call context into the message, and the Player unmarshals it at the server side before making the method call. As far as the server object is concerned, there is no difference in security context between a direct call from the client and an indirect call from the Player.

This mechanism works only if we can trust the process that enqueues the message. There is a remote possibility that a malicious user could generate a message whose format resembles the QC message format. In this case, the marshaled security call context may have been constructed to attempt access to the server object. The Recorder places the client's SID in the message and sets appropriate MSMQ properties so that MSMQ carries the sender's SID as well. The user cannot change the MSMQ SID. The Player compares the MSMQ SID with the SID in the message. If the two SIDs match, then the process that enqueued the message is the same as the process that made the method calls to the queued component. It is also possible to configure the Recorder to run as a different user. However, in this case, the user who is running the Recorder must be registered as a trusted "COM+ enqueuer."

Dynamic Load Balancing

Dynamic load balancing is a mechanism to distribute client calls to multiple servers for processing that is transparent to the client. In dynamic load balancing, a target group (typically the server group) is established for a given application. The application is automatically replicated between targets in the group. Data is centrally shared or is replicated between the servers (for read-only applications). Clients or client applications are configured to use a load-balancing router. A load balancing analyzer monitors managed resource entities. A load-balancing engine uses analyzer data to dynamically make decisions about which target to use. Various analyzers can cooperate to make the routing decision.

Windows 2000 contains two load-balancing technologies:

- COM+ load balancing allows applications built using COM components to scale across multiple servers. It is intended to provide fine-grained, dynamic load balancing for application servers in a client-transparent manner.
- The Windows Load Balancing (WLB) service provides IP-based balancing and multiple-server scalability for system-level services. These services take different approaches to address multi-node scalability, but they can be used in a highly complementary manner.

The COM+ Approach

In the typical COM remote server scenario, the client application is configured to create a server object on a specific server. When the server application (and objects) is configured to be load balanced across a group of servers, known as an *application cluster*, the client is configured to create server object(s) on a special server, called the *component router* or *load-balancing router*.

The component router is simply a service that may be installed on a server in an application cluster. The client programming model remains the same. The component router has knowledge of the servers that are capable of creating instances of the server objects, and it forwards creation requests to the appropriate server. The choice of server depends on the load-balancing algorithm implemented on the component router.

NOTE:

The load-balancing algorithms are not extensible in COM+ v1. The only option available is the response-time-analysis algorithm. It is possible that user extensibility will be available for COM+ v2.

This approach allows fine-grained control, at the component level, of load distribution. The server administrator can choose which components can be load balanced on which set of servers. Furthermore, fine-grained statistics can be incorporated into making routing decisions when picking the appropriate server.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The WLB Approach

The WLB technology is targeted toward distributing a consistent load across identical servers providing homogenous services. In this configuration, a set of servers is configured identically to provide the same system services. They are configured to respond to the same IP address.

The Windows 2000 technology is implemented at the driver level. Every server sees every client request at the network level, but only one server services this request. This is based on an agreed-upon algorithm and is typically based on the client's IP address.

Object Pooling

Object pooling was designed to offset the cost of creating expensive objects. Performance can be improved by pooling the object. When you request a new object, `CoCreateInstance` pulls from the pool, and `Release` returns to the pool. Central to the high performance of this mechanism is that the pool is homogeneous per CLSID.

While a consumer creates and releases objects, the consumer is unaware whether objects are really being created and released or if, instead, pooled objects are being activated and deactivated. To pool or not to pool is the private business of the component developer and application administrator.

In MTS 1, object pooling requires cooperation of the component developer, who must implement the `IObjectControl` interface to coordinate object instance activation and deactivation. But this interface is not visible to the object's consumer. The consumer sees only the *real* interfaces that the object exposes. In previous versions of MTS, the `IObjectControl` interface had a method called `CanBePooled`. Although that method was not used, you needed to return a Boolean value. For VB components, the recommended value was `False`, which was a good choice, since VB objects would not be pooling (due to the threading model they implement).

Summary

In this chapter, we discussed what makes COM special and why it is a critical technology to building distributed applications. We also explored what makes a piece of code a COM component. You learned that COM is a Microsoft specification for the construction of binary-compatible software components. We

discussed the main benefits of using COM, and explored the concept of COM interfaces and the rules for interfaces. You learned that components depend on interfaces to communicate and that, once published, interfaces cannot be altered. You also learned that a COM component must implement the IUnknown interface. We also reviewed the rules for implementing COM components and discussed COM activation.

Next, you learned about the three main ways of using COM: COM clients, COM servers, and ActiveX controls. Which type you use depends on your application. We discussed some object-oriented programming techniques that can be used in conjunction with COM. The techniques are related to the concepts of encapsulation, abstraction, polymorphism, and inheritance. The rules discussed in this chapter are the foundation on which we will be building. It is critical to understand how COM and interfaces work.

Next, COM+ was introduced as an extension of COM, beginning with a brief summary of Windows DNA, which has evolved from being just a marketing strategy into a set of tangible services (although some parts of it, such as the user interface tier and database tier technologies are still undefined).

Finally, we covered several of the new services in COM+: transactions, QC (Queued Components), load balancing, and object pooling.

Next, in Chapter 22, we'll look at building and using COM Servers in VC++.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



▶ **KEYWORD SEARCH**

▶ [Search Tips](#)

▶ [Advanced Search](#)

▶ **PUBLICATION LOOKUP**

Title

▶ **JUMP TO TOPIC**



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#) [Table of Contents](#) [Next](#)

CHAPTER 22

Building and Using COM Servers in VC++

- An introduction to IDL
- A COM server created with VC++
- A COM server created with ATL
- In-process versus out-of-process servers
- Threads and COM servers
- Automation and IDispatch

In Chapter 21, “COM/COM+ Overview,” you learned that you can create your own custom interfaces. Now the question is, how do you define your custom interfaces? The answer is, you use the Microsoft Interface Definition Language (MIDL).

MIDL is a declarative language used to define COM interfaces. It is based on the Open Software Foundation (OSF) Distributed Computing Environment (DCE) Remote Procedure Call (RPC) IDL. Its objective is to allow interfaces to be defined in a language-neutral manner while being used to generate RPC code that handles communicating with a COM server implementing the defined interfaces. The language itself borrows a lot from the C++ syntax. For the remainder of this book, we will use *IDL* to refer to the Interface Definition Language used to define COM interfaces, and *MIDL* to refer to the Microsoft IDL compiler itself.

In this chapter, we focus on building COM servers in VC++ where you'll get some hands-on experience as we create and use COM servers. At the same time, you'll learn about the underlying theory, which will help you to apply the concepts introduced here.

A Look at an IDL File

Before examining the structure of an IDL file, we need to review the three key elements of a COM server:

Interface The interface is the contract between the server and its clients. Clients use the interface to communicate with the server.

CoClass This is the component class that provides the implementation of the defined interface.

Type library This is a compiled version of the IDL file used to convey information about the interface to environments that support COM (such as VB).

The following shows an IDL file from the <Visual Studio path>\VC98\Include directory. This particular file defines the IWPObj interface (part of the Microsoft Web Publishing API).

```
/*-----  
 *  
 *   Copyright 1997 - 1998 Microsoft Corporation  
 *  
 *   Module Name:  
 *  
 *       wpobj.idl  
 *  
 *   Abstract:  
 *  
 *       Declaration of the IWPObj interface  
 *       and type library  
 *  
 \*-----  
  
[  
    object,  
    uuid(EDD8BBC0-9240-11CF-9ED3-00AA004C120C),  
    dual,  
    helpstring("IWPObj Interface"),  
    pointer_default(unique)  
]  
interface IWPObj : IDispatch  
{  
    import "oaidl.idl";  
  
    HRESULT WpPostFile(  
        [in]                LONG hwnd,  
        [in, string]        BSTR bsLocalPath,  
        [in, out]           LONG *plSiteNameBufLen,  
        [in, out, string]   BSTR *pbsSiteName,  
        [in, out]           LONG *plDestURLBufLen,  
        [in, out, string]   BSTR *pbsDestURL,  
        [in]                LONG lFlag,  
        [out]               LONG *plRetCode );  
}
```

```

};
[
    uuid(336c8c70-a62b-11d0-ad5f-00aa00a219aa),
    version(1.0),
    helpstring("WPObj 1.0 Type Library")
]
library WPObjLib
{
    importlib("stdole2.tlb");

    [
        uuid(53DEFDE0-9222-11CF-9ED3-00AA004C120C),
        helpstring("WPObj Class")
    ]
    coclass WPObj
    {
        [default] interface IWPObj;
    };
};

```

This file begins with a comment section documenting its name and purpose. The next section begins with a square bracket (I), which means there are a set of attributes to follow.

IDL allows the definition of attributes by listing them within square brackets before the element to which they relate. So the attribute `helpstring("IWPObj Interface")` relates to the interface `IWPObj`. The following attributes are listed in the previous code listing:

- The first attribute is `object`, which tells the IDL compiler that this is a COM interface definition rather than an RPC interface definition.
- The GUID, defined by `uuid(EDD8BBC0-9240-11CF-9ED3-00AA004C120C)`, is the unique identifier of the `IWPObj` interface (IID).
- The dual attribute says that clients have two different ways of accessing properties and methods exposed by this interface (we will discuss dual interfaces in more detail later in this chapter, in the “Automation and IDispatch” section).
- The `helpstring` attribute adds the quoted string to the type library as a hint to developers using this COM server.
- Finally, the `pointer_default` attribute specifies the default characteristics for all pointers except those included in parameter lists; unique means pointers may be NULL but do not support aliasing.

After the list of attributes comes the interface definition itself. Here, we define the interface `IWPObj`, which inherits from the `IDispatch` interface. Next, we use the `import` statement to bring in the interface definitions found in the file `oidl.idl`, which includes `IDispatch`. The `import` statement is similar to the C `#include` directive in that it brings in definitions contained in another IDL file and makes them available within our file. The interface exposes the method `WpPostFile`, which has three input parameters (designated by `[in]`), four input/output parameters (designated by `[in,out]`) and one output parameter (designated by `[out]`). We will discuss `IDispatch` and parameter data types in more detail in “Automation and IDispatch,” later in this chapter.

So far, we’ve defined the interface. If you remember, there are two more key elements: the component class and the type library. We take care of these in the next section. The library statement defines the type library `WPObjLib`, which has its own unique identifier that is defined in the preceding attributes section as `uuid(336c8c70-a62b-11d0-ad5f-00aa00a219aa)`. The `importlib` directive is similar to `import`, except for using a binary (compiled) type library. All type libraries are required to `importlib` the base type library defined in `Stdole32.tlb`. When you import a type library using the `importlib` directive, you need to make sure that the library will be available to your clients (you must install the library with your COM server if it is not already there).

Finally, we define our `CoClass` (component class) called `WPObj` and we specify that it implements the previously defined interface `IWPObj`.

That’s it! We have defined the basic elements needed for a COM server. This was just the definition; we have not actually implemented any

interfaces.

Building Our First COM Server

Now that you've learned how to define interfaces using IDL, the next step is to understand more of the concepts behind COM. To do this, we will build a simple in-process COM server using VC++. The process we'll follow goes like this:

- Define the custom interfaces we wish to implement.
- Implement the required interface IUnknown.
- Implement our custom interfaces.
- Add some necessary code to enable clients to create and use instances of our COM server.

So, if you're ready to get your hands dirty, fire up VC++ and let's get started.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Defining Custom Interfaces

To define our custom interface, we need to create a new text file and call it FirstSrv.idl, as shown in the following sample code. As you may have already guessed, we will use this file to write our IDL definition. We want to define a component called FirstComponent, which implements the custom interface IFirstInterface (I suppose by now you see the "First..." pattern we've got going here). The interface will have a single method: DoSomething().

```
//FirstSrv.idl
import "oaidl.idl";

[
    object,
    uuid(Place GUID1 Here)
]
interface IFirstInterface : IUnknown
{
    HRESULT DoSomething();
};

[
    uuid(Place GUID2 Here),
    version(1.0)
]
library FirstTypeLib
{
    importlib("stdole32.tlb");
    [
        uuid(Place GUID3 Here)
    ]
    coclass FirstComponent
    {
        [default] interface IFirstInterface;
```

```
};
};
```

Type the above code into the text file we just created. You will need to generate three GUIDs, one each for the interface ID (IID), type library ID (LIBID), and CoClass ID (CLSID). To generate GUIDs, you can run GUIDGEN.EXE, which is a Visual Studio tool, and copy the GUID in Registry format. After you paste the GUID into the IDL file, remember to remove the extra braces around it.

Save the file and open a command prompt. Change to the directory where the file is saved and issue the following command to compile the IDL file:

```
midl FirstSrv.idl
```

MIDL.EXE is the MIDL compiler. This compiler reads the IDL file and outputs the following key files:

- The interface proxy file, firstsrv_p.c, contains marshaling code for the interface IFirstInterface defined in the IDL.
- The header file, firstsrv.h, contains C++ interface and type definitions. It also declares symbolic constants for the interface ID (IID_IFirstInterface) and the CLSID (CLSID_FirstComponent).
- The interface UUID file, firstsrv_i.c, contains the GUID *definitions* for the IID, CLSID, and LIBID declared in the header file.
- The type library, firstsrv.tlb, is a binary version of the IDL file.

Implementing IUnknown and Our Custom Interface

Now that we have our interface definition, the next step is to implement the required interface IUnknown. Start Visual Studio and create a new Win32 DLL project called FirstSrvDll. Choose the option to create an empty DLL project and click Finish. Add a new C++ header file to the project and call it FirstComponent.h. This is where we will put our component definition and the implementation. The following code listing shows the CFirstComponent class implementing the IFirstInterface interface (FirstComponent.h).

```
#include <windows.h>
#include "firstsrv.h"

class CFirstComponent : public IFirstInterface
{
public:
    CFirstComponent() : m_ulRefCnt(0)
    {
    }

    ~CFirstComponent()
    {
    }

    //methods of the IUnknown interface
    STDMETHOD (QueryInterface)(REFIID riid, void** ppv)
    {
        if( riid == IID_Iunknown )
            *ppv = static_cast<IUnknown *>(this);
        else
            if( riid == IID_IFirstInterface )
                *ppv = static_cast<IFirstInterface *>(this);
            else
                *ppv = NULL;

        if(*ppv!=NULL)
        {
            static_cast<IUnknown *>(*ppv)->AddRef();
            return S_OK;
        }
        else
            return E_NOINTERFACE;
    }
};
```

```

    }
    STDMETHOD_ (ULONG, AddRef)()
    {
        InterlockedIncrement( reinterpret_cast<LPLONG> 0
                               (&m_ulRefCnt) );
        return m_ulRefCnt;
    }

    STDMETHOD_ (ULONG, Release)()
    {
        if( ! InterlockedDecrement( reinterpret_cast 0
                                     <LPLONG> (&m_ulRefCnt)) )
        {
            delete this;
            return 0;
        }
        return m_ulRefCnt;
    }

    //the IFirstInterface methods
    //(there's only one method)
    STDMETHOD (DoSomething)()
    {
        MessageBox( NULL, "We did it!", "First COM Server",
                    MB_OK + MB_ICONINFORMATION);
    }

private:
    ULONG m_ulRefCnt;
}

```

NOTE:

Unlike most data types starting with an *H* in VC++, HRESULT is not a handle to anything. As explained in Chapter 21, most COM method calls and COM APIs return HRESULT values indicating whether the call succeeded or failed and providing information on the cause of failure. When working with HRESULT values, you should not attempt to interpret the results manually; instead, use the COM error-handling macros. The most commonly used macros are `FAILED`, which returns `TRUE` if the function call failed, and its complement macro `SUCCEEDED`, which returns `TRUE` if the function call succeeded.

Let's take a look at this code to see what it does. First, we include the `windows.h` file, which contains the declaration of some APIs we use later on. Next, we include the file `firstsrv.h`, which was generated by the IDL compiler. As pointed out earlier, this is the file with our interface definition. Then we declare a class called `CFirstComponent` that derives from `IFirstInterface` (the interface definition comes from `firstsrv.h`).

The class constructor simply initializes the member variable `m_ulRefCnt` to 0. After the destructor, we start implementing the methods required for the `IUnknown` interface. The macros `STDMETHOD` and `STDMETHOD_` are used to describe the return type and calling convention of functions. `STDMETHOD` assumes that the function return type is `HRESULT`, which is valid for almost all COM interface methods. `STDMETHOD_` allows you to specify the function return type as the first parameter. Since `AddRef` and `Release` return a reference count, we use `STDMETHOD_` to define these methods. `QueryInterface` is defined using `STDMETHOD` because it returns an `HRESULT`. We could have just as easily declared the methods as follows:

```

virtual HRESULT _stdcall QueryInterface( REFIID riid,
                                         void **ppv );
virtual ULONG _stdcall AddRef();
virtual ULONG _stdcall Release();

```

Although declaring methods this way is perfectly valid, it is strongly dependent on Win32 systems. Using the macros enables us to compile our code on other systems such as a Macintosh (or maybe even Win64!). `STDMETHODIMP` is similar to `STDMETHOD`, except that it does not define the method as virtual.

The QueryInterface method examines the requested interface to see whether it is supported by this component. To do this, it compares riid to the interface ID for IUnknown (IID_IUnknown) and to IID_IFirstInterface, then returns the appropriate pointer by casting this to the appropriate base class. Note that before returning a pointer to the requested interface, we must AddRef it. This is a COM rule to ensure that the component doesn't suddenly die before the client is done using the interface.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

AddRef and Release are similar in their functionality. They both use Win32 APIs to get exclusive control of the variable `m_ulRefCnt` while incrementing or decrementing it. We do this to make our component “thread-friendly.” It would be a problem if AddRef used something like `m_ulRefCnt++` while multiple threads are executing this statement concurrently. In addition to decrementing the reference count, Release also deletes the object instance if the reference count is 0. We are assuming that the object was created using new; therefore, we delete it using delete.

Now we are finished with the IUnknown interface, and we can implement our IFirstInterface. The method DoSomething is the only method in this interface. Its implementation is rather simple—it pops up a message box signaling our success.

Completing the COM Server

If you have been following along, you’ll notice that we have completed the implementation for the IUnknown interface as well as our own custom interface, IFirstInterface. The next step involves writing all the code that enables clients to create our COM component and get access to the interfaces it supports. Before we start doing this, we need to talk about class factories.

Understanding the Class Factory

Imagine you’re a client and you want to create an instance of our component and get a hold of the IFirstInterface interface. How do you go about that? The logical steps would be as follows:

- Find the binary file that contains the COM server.
- Load that file into memory.
- Ask that file to create an instance of the COM server and give you a pointer to IUnknown implemented by that server.
- Proceed with QueryInterface to get a pointer to IFirstInterface.

Obviously, your client is not going to do all this work on its own, it will need some help from system-level services. Typically, to create an instance of a COM server, the client calls the API CoCreateInstance, which is defined as follows:

```

WINOLEAPI CoCreateInstance( REFCLSID rclsid,
                             LPUNKNOWN pUnkOuter,
                             DWORD dwClsContext,
                             REFIID riid,
                             LPVOID * ppv );

```

The parameter `rclsid` is the CLSID of the desired component. The parameter `riid` is the desired interface, and `*ppv` is used to return a pointer to that interface. We will be look at the other parameters later in this chapter.

In general, creating instances of a COM component may require certain steps that are specific to that COM component, such as coordination with other objects. Such specifics cannot be addressed directly by `CoCreateInstance` because it is a generic API that has no prior knowledge of all the available COM components that it will need to create. To solve this problem, `CoCreateInstance` does not directly create an instance of the COM component. Instead, it gets a pointer to the specific implementation of the interface `IClassFactory`, which is responsible for creating instances of the desired COM component. It then delegates instance creation to this class factory.

To get a pointer to `IClassFactory`, `CoCreateInstance` calls the function `CoGetClassObject`, which handles loading the DLL and getting a pointer to the appropriate `IClassFactory` interface. This leads us to our next task: building a class factory for our component.

A class factory implements the `IClassFactory` interface, which has the following two methods:

```

HRESULT CreateInstance( IUnknown * pUnkOuter,
                       REFIID riid,
                       void ** ppvObject );
HRESULT LockServer( BOOL fLock );

```

`CreateInstance` is used to create an object instance and return a pointer to the requested interface. `LockServer` is used to keep the server (for example, the DLL file) in memory. You may want to do this if you are frequently creating and destroying objects using this COM server. By keeping the server in memory, future object creations are performed faster.

Add a new C++ header file to the project and call it `FirstClassFactory.h`. This is where we will put the following class factory implementation.

```

//bring in our component declaration
#include "FirstComponent.h"
class CFirstClassFactory : public IClassFactory
{
public:
    CFirstClassFactory()
    {
    }

    ~CFirstClassFactory()
    {
    }

    //methods of the IUnknown interface
    STDMETHOD (QueryInterface)(REFIID riid, void** ppv)
    {
        if( riid == IID_Iunknown )
            *ppv = static_cast<IUnknown *>(this);
        else
            if( riid == IID_IClassFactory )
                *ppv = static_cast<IClassFactory *>(this);
            else
                *ppv = NULL;
        if( *ppv != NULL )
        {
            static_cast<IUnknown *>(*ppv)->AddRef();
            return S_OK;
        }
    }
}

```

```

    }
    else
        return E_NOINTERFACE;
}
STDMETHOD_ (ULONG, AddRef)()
{
    InterlockedIncrement( reinterpret_cast<LPLONG>
                          (& g_ulRefCnt) );
    return g_ulRefCnt;
}

STDMETHOD_ (ULONG, Release)()
{
    //we are not deleting the object because we
    //won't be creating it on the heap
    InterlockedDecrement( reinterpret_cast<LPLONG>
                          (& g_ulRefCnt) );
    return g_ulRefCnt;
}

//IClassFactory Methods
STDMETHOD( CreateInstance)( IUnknown* pUnkOuter,
                           REFIID riid,
                           void** ppvObject )
{
    *ppvObject = NULL;
    //create component
    CFirstComponent * pfc = new CFirstComponent;
    if( pfc == NULL )
        return E_OUTOFMEMORY;
    HRESULT hr = pfc->QueryInterface( riid, ppvObject );
    if( FAILED(hr) )
        delete pfc;
    return hr;
}

STDMETHOD( LockServer )( BOOL fLock )
{
    if( fLock )
        InterlockedIncrement( &g_lLocks );
    else
        InterlockedDecrement(&g_lLocks);
    return S_OK;
}
};

```

We are creating the class CFirstClassFactory, which inherits from IClassFactory and implements the methods CreateInstance and LockServer (since this is a COM server, we must also implement the IUnknown methods). CreateInstance simply creates a new object of type CFirstComponent. Remember how the Release method of the CFirstComponent class uses delete to delete the object? Well, this is where we create the object; we must use new to create it so that it can be deleted using delete.

After creating the object, we call QueryInterface, asking for a pointer to the requested interface riid. If the object does not support the requested interface, then the call will fail and we return to the client the HRESULT we get from QueryInterface. LockServer increments or decrements the number of locks held on the COM server. If fLock is TRUE, we increment the number of locks using the InterlockedIncrement API; otherwise, we decrement the number of locks using the InterlockedDecrement API. This is similar to AddRef and Release implemented in one function, except here we are managing the server lock count rather than an interface reference count.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Creating the DLL File

Now that we have the component and its class factory, we need to add a few more functions to make this a legitimate DLL COM server. Here is a description of what we need to implement (note that these are all API style functions that are called directly rather than through some COM interface):

DllMain This is a DLL entry point, which is called by the system when a thread or process begins using the DLL and when it is done with the DLL. This allows you to do any initialization work needed and to clean up afterwards. We will simply return TRUE from this function.

DllGetClassObject This function is called by the CoGetClassObject function to retrieve a pointer to the IClassFactory interface.

DllCanUnloadNow As the name implies, this function returns S_OK if the number of locks on the server is 0; otherwise, it returns S_FALSE.

DllRegisterServer This function is used to register the COM server with the system. On Win32 systems, this means adding the appropriate entries to the Windows Registry.

DllUnregisterServer This function is used to unregister the COM server. On Win32 systems, this is done by removing the relevant entries from the Windows Registry.

The following code implements the DLL functions required to complete the COM server (FirstSrvDll.cpp):

```
// FirstSrvDll.cpp

//lock count
static long g_lLocks=0;

//reference count to class factory
//we can do this because it is a singleton
static long g_ulRefCnt;

//class factory header file
#include "FirstClassFactory.h"
```

```

    //file containing GUID definitions generated by MIDL
#include "firstsrv_i.c"

    //header file containing error constants
#include "Olectl.h"

    //the global Class Factory singleton
static CFirstClassFactory g_FirstClassFactory;

    //this DLL's HINSTNACE. Used to get the DLL file path
static HINSTANCE Myhinstance;

BOOL OpenKey( const char* szKey, HKEY& hkey )
{
    //opens the key specified in szKey
    //and returns its handle in hkey
}

BOOL CreateKey( const char* szParentKeyName,
                const char* szSubKeyName,
                const char* szDefaultVal )
{
    //creates a sub key under the given parent key
}

BOOL SetValue( const char* szKeyName,
                const char* szValName,
                const char* szValData )
{
    //sets a value under the specified key name
}

BOOL DeleteKey( const char* szKey )
{
    //deletes the specified key
}

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason,
                    LPVOID lpvReserved )
{
    Myhinstance = hinstDLL;
    return TRUE;
}

STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid,
                        LPVOID* ppv )
{
    if( rclsid == CLSID_FirstComponent )
    {
        return g_FirstClassFactory.QueryInterface( riid,
                                                    ppv );
    }
    else
    {
        *ppv=NULL;
        return CLASS_E_CLASSNOTAVAILABLE;
    }
}

STDAPI DllCanUnloadNow(void)
{

```

```

        if( g_lLocks==0 && g_ulRefCnt==0 )
            return S_OK;
        else
            return S_FALSE;
    }

STDAPI DllRegisterServer(void)
{
    char szMyPath[MAX_PATH];

    if( ! GetModuleFileName( Myhinstance, (LPSTR) szMyPath,
                            MAX_PATH ) )
        return SELFREG_E_CLASS;

    //Create a new component key under CLSID
    if( ! CreateKey( "CLSID",
                    "{57140483-CB78-11d2-91A6-0008C7FE9130}",
                    "FirstComponent" ) )
        return S_FALSE;

    //Check whether we were successful
    if( ! CreateKey( "CLSID\\{57140483-CB78-11d2-91A6-0008C7FE9130}",
                    "InprocServer32", (const char *)szMyPath ) )
        return SELFREG_E_CLASS;

    if( SetValue( "CLSID\\{57140483-CB78-11d2-91A6-0008C7FE9130}
                  ~CA\\InprocServer32",
                  "ThreadingModel", "Apartment" ) )
        return S_OK;
    else
        return SELFREG_E_CLASS;
}

STDAPI DllUnregisterServer(void)
{
    BOOL bSuccess = FALSE;

    //Delete the Inproc32 subkey
    if( ! DeleteKey( "CLSID\\{57140483-CB78-11d2-91A6-0008C7FE9130}
                    ~CA\\InProcServer32" ) )
        return SELFREG_E_CLASS;

    //Delete the component's key
    if( ! DeleteKey( "CLSID\\{57140483-CB78-11d2-91A6-0008C7FE9130}" ) )
        return SELFREG_E_CLASS;
    else
        return S_OK;
}

```

Notice that the variable `g_lLocks` is defined as a global static variable. As we've seen before, this is incremented and decremented by the `LockServer` method of the `CFirstClassFactory` class. `DllCanUnloadNow` returns `S_OK` if both `g_lLocks` and `g_ulRefCnt` are 0. There is one other global variable: `g_FirstClassFactory`. This is a single instance of the `CFirstClassFactory` class, which is used to create objects of the COM component.

We created the functions `OpenKey`, `CreateKey`, `SetValue`, and `DeleteKey` to handle registering and unregistering our COM server (they do not handle the type library registration, which is okay because we don't need it

now). You will need to replace the GUID used in DllRegisterServer with the one you used for CoClass in your IDL file. The complete code for these functions is available on the CD-ROM.

The DllGetClassObject function checks the requested CLSID. If it is CLSID_FirstComponent, then it uses the global instance of CFirstClassFactory to query for the requested interface (riid) and return the result. Note that CFirstClassFactory::QueryInterface knows only about IUnknown and IClassFactory, so if riid is referring to any other interface, the call to g_FirstClassFactory.QueryInterface will fail.

DllRegisterServer and DllUnregisterServer simply manage adding or removing the necessary Registry keys and values. DllRegisterServer creates a Registry key under HKEY_CLASSES_ROOT\CLSID. The name of this key is simply our component's CLSID enclosed in braces (the same one used in the IDL file). Inside this key, we set the (Default) value to the name of our CoClass: FirstComponent. Under this new key, we create another key called InprocServer32. Inside this key, we set the (Default) value to the path of our DLL file (the path is obtained from GetModuleFileName()). We also add a value called ThreadingModel and we set it to Apartment (we will discuss threading models in more detail in the section "Threads and COM Servers," later in this chapter).

Finally, to export the above functions, we create a text file called FirstSrv.def to define our DLL exports:

```
LIBRARY "FirstSrvDll.dll"
EXPORTS
    DllCanUnloadNow           @1      PRIVATE
    DllGetClassObject         @2      PRIVATE
    DllRegisterServer         @3      PRIVATE
    DllUnregisterServer       @4      PRIVATE
```

This won't be new to you if you've created DLL files that exposed an API before. Note that these are COM-specific APIs intended to be used only by the COM system; therefore, they must be exported as PRIVATE.

That's it; we have completed our first COM server! You can build your project now to get the DLL file FirstSrvDll.dll. Once you've done that, register the COM server by issuing the following command from the directory where the DLL file resides:

```
regsvr32 FirstSrvDll.dll
```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

REGSVR32.EXE will load our DLL and call the DllRegisterServer function to create the necessary Registry entries. Now we are ready to test our server.

Creating a Test Client

Just like our first server, our test client will be simple. We will create a Win32 console application called FirstSrvClient (use AppWizard to create an empty application), then add a C++ file called main.cpp. In this file, we will code the single function main(), which will create an instance of our server and call the DoSomething method on it.

To access IFirstInterface, we need to include the file firstsrv.h, which was generated a while back by the MIDL compiler. To access our CLSID and IID, we include the file firstsrv_i.c, which was also created by the MIDL compiler. Then we initialize COM by a call to CoInitialize (which is matched at the end by a call to CoUninitialize). Next, call CoCreateInstance using CLSID_FirstComponent as the CLSID of the desired component and IID_IFirstInterface as the IID of the desired interface. We expect the interface pointer to come back in the variable pFI.

Then we check the returned HRESULT. If successful, we simply call the DoSomething method, then release the reference to the interface (remember that CoCreateInstance returns an interface pointer that has been AddRef'd, so we must provide a matching Release). Finally, we uninitialize COM by calling CoUninitialize. The following shows the test client for IFirstInterface from the main.cpp source code.

```
//main.cpp
#include <windows.h>
#include "..\FirstSrvDll\firstsrv.h"
#include "..\FirstSrvDll\firstsrv_i.c"
int main()
{
    //Initialize COM
    CoInitialize(NULL);

    //create an instance of the FirstComponent
    //and get a pointer to IFirstInterface
```

```

IFirstInterface* pFI = NULL;
HRESULT hr=CoCreateInstance( CLSID_FirstComponent, NULL,
                             CLSCTX_INPROC_SERVER,
                             IID_IFirstInterface,
                             reinterpret_cast <void**>
                             (&pFI) );

if( FAILED(hr) )
{
    MessageBox( NULL, "Could not create instance",
                "Error in CoCreateInstance",
                MB_OK + MB_ICONSTOP );
}
else
{
    //Use the interface
    pFI->DoSomething();
    //release the reference to the interface
    pFI->Release();
}
CoUninitialize();
return 0;
}

```

Go ahead and build and run our test client. You should get a message box similar to this one:



To gain a better understanding of how all the pieces fit together, step through the code line by line, following the execution from the client to the COM server and back to the client. Follow this procedure to step through the code:

1. Make a debug build of the server and client.
2. Open the FirstSrv project.
3. Select Project Ø Settings. On the Debug tab, set the Executable for Debug Session option to the path of the client executable: FirstSrvClient.exe.
4. Place breakpoints in the functions DllGetClassObject and DllCanUnloadNow.
5. Open the client file main.cpp and place a breakpoint on the CoCreateInstance call.
6. Press the F5 key to start debugging.
7. Step through the code using the F10 key to step over a function call (such as CoCreateInstance) and the F11 key to step into a function call (such as g_FirstClassFactory.QueryInterface).

While stepping through the code, examine the reference counts and the lock counts to see how they behave, as shown in the example in Figure 22.1.



FIGURE 22.1 Stepping through our COM server and test client

So far, we have used IDL to define COM interfaces and then implemented those interfaces. We also created a class factory and added the necessary functions to our COM server DLL. Next, we will take a look at the Active Template Library and see how it can make it easier to create COM servers.

Building COM Servers with ATL

You probably noticed that when we created our first COM server, we spent the bulk of our effort doing COM-related work, such as implementing interfaces and creating a class factory. This effort may seem especially disproportionate because we implemented only one method (DoSomething()). Building COM components with raw C++ is a lot of work, and much of it may be repetitive. Well, as it turns out, computers are especially good at automating repetitive work, and that's exactly why Active Template Library (ATL) was designed.

ATL helps create the code necessary to do a lot of the COM-related work, such as registering and unregistering servers, creating class factories, and managing server locking. Notice that we said it “helps create the code.” That is because ATL is based on C++ templates, which are easily customized to generate the code you want. Because it generates code for us, we get only what we need.

As even more help, ATL allows you to build COM servers with very few required support files. Compare this to building COM servers in VB or Microsoft Foundation Classes (MFC). These techniques also automate much of the COM work, but they require runtime libraries that can reach several megabytes in size. If size and speed are issues, you should definitely consider ATL.

A Look at ATL

To build a COM component using ATL, you create a class that derives from at least two other classes: CComObjectRoot and CComCoClass. In addition, a class called CComObject derives from your class.

Figure 22.2 illustrates the basic relationship between the most common classes in an ATL COM server. The diagram shows some of the methods implemented by each class. As you can see, CComObject exposes the IUnknown interface. The AddRef and Release methods delegate the actual work to CComObjectRoot. Since CComObject is derived from your class, it needs to know the name of that class. That is why it is a template, and the template parameter is the name of your class, which becomes the base class for CComObject.

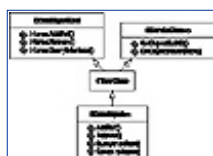


FIGURE 22.2 Inheritance hierarchy of an ATL COM object. CyourClass is the class we add to provide CoClassimplementation.

CComCoClass defines the class factory model by using the DECLARE_CLASSFACTORY macro (we'll take a closer look at this macro in a few minutes). This class also provides two standard methods to get an object's CLSID and description. Any COM object that can be created by a client must be derived from CComCoClass. This class is provided as a template with two parameters: your class name and the component's CLSID.

CComObjectRoot provides the actual implementation for managing the object's reference count. A class that implements a COM server must derive from CComObjectRoot or CComObjectRootEx. CComObjectRootEx is a class template that takes one parameter: the threading model. CComObjectRoot is defined as a typedef using CComObjectRootEx and the default threading model of the server.

In addition to the class templates mentioned here, there are many other class templates, global functions, and macros provided by ATL.

Building an In-Process COM Server with ATL

We will start building our first ATL COM server and use it to learn more about ATL as we proceed. This time, we are building a piece of the COMTrader application, the Access Control Manager. This component will be responsible for checking whether the user has access to a requested function. In the completed application, the component will read access control information from a database table; but to keep things simple for now, we will just allow everyone access to everything!

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

We'll start by creating a new VC++ project called SecurityMgr. We'll use the ATL COM AppWizard (by selecting it from the New Projects dialog box). Figure 22.3 shows the ATL COM AppWizard dialog box.



FIGURE 22.3 The ATL COM AppWizard dialog box

This wizard is simple yet powerful. It has only one dialog box, which allows you to choose the type of server as well as options for MFC and Microsoft Transaction Server (MTS), which is the predecessor to COM+ support. MFC support allows you to use MFC in your ATL project. One reason you may want to do this is if you are reusing some legacy code that uses MFC. For our current purposes, we want a DLL and we do not want any of the other options (we will be making other kinds of COM servers, including those with MTS/COM+ support, in later chapters).

Select the Dynamic Link Library (DLL) radio button in the ATL COM AppWizard dialog box, and then click the Finish button. At the New Project Information dialog box, click OK to let the wizard create the necessary files.

Examining Project Files

Figure 22.4 shows a list of the files currently in the project. Let's take a look at each of the wizard-generated files to understand what they give us. (If you are anxious to start coding, you can skip to the section "Adding a CoClass.")

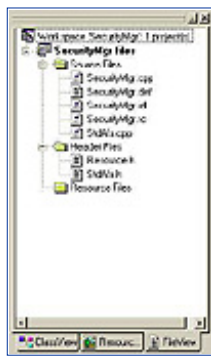


FIGURE 22.4 The files added to our project by the ATL COM AppWizard

Source Files The first file is the project's .CPP file, `SecurityMgr.cpp`. This file begins with the #includes. First it includes `stdafx.h`, which is a precompiled header file. `Resource.h` is the file containing project resource definitions. `Initguid.h` is a system header file that defines the `DEFINE_GUID` macro. Next, it includes two files generated by the MIDL compiler when you build the project:

- `SecurityMgr.h` contains C++ interface and type definitions. It also declares symbolic constants for the IID and the CLSID.
- `SecurityMgr_i.c` contains the GUID definitions for the IID, CLSID, and LIBID.

Note that neither of these files is created until you compile the IDL file.

The wizard automatically adds a custom-build step to ensure the IDL file is compiled as part of the build process. Therefore, when you build the project (by pressing the F7 key), these files are created and made available to the VC++ compiler when it kicks in.

The `SecurityMgr.cpp` file continues with the declaration `CComModule _Module`, which represents a shell of a COM server. It provides support for locking and unlocking the server, adding and removing Registry entries, and registering and revoking class factories (this is needed for out-of-process COM servers, as we will see later in this chapter). `_Module` is the one global instance of the `CComModule` class used throughout the project.

The macros `BEGIN_OBJECT_MAP` and `END_OBJECT_MAP` delimit a list of all COM objects in the server. The parameter `ObjectMap` is an array holding `_ATL_OBJMAP_ENTRY` structures, each describing an object that may be created independently (an object that may be created by calling `CoCreateInstance`). An `_ATL_OBJMAP_ENTRY` structure contains information related to creation or registration of the class it defines. Elements of the `ObjectMap` array are defined by using the `OBJECT_ENTRY()` macro for each element. We will see how this macro is used as we add objects to our project.

Next comes the `DllMain` function. As you know, we need to implement this ourselves when we build a COM server without using ATL. In this case, the ATL COM AppWizard takes care of creating this function for us with an appropriate implementation, which calls `_Module.Init` to initialize the global instance of `CComModule`. The call to `DisableThreadLibraryCalls` disables the `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` notifications for our DLL. This is done as a way to optimize performance in an environment where threads are being frequently created and destroyed. Our DLL will not receive notification with each such creation or destruction of a thread, which is fine since we do not need such notifications.

The remaining code in the `SecurityMgr.cpp` file implements the functions `DllCanUnloadNow`, `DllGetClassObject`, `DllRegisterServer`, and `DllUnregisterServer`. These are all functions we need to implement ourselves when we are not using ATL. With ATL, these are created for us by the ATL COM AppWizard.

There are four other source files currently in our project:

SecurityMgr.def This file is similar to the .def file we created manually while building our first COM server earlier in this chapter. It contains the DLL exported function names and their ordinal numbers. Remember that functions for use by the COM system must be private functions.

SecurityMgr.idl The IDL file is automatically generated for us by the wizard. It includes a GUID for the type library. Note however, that it does not contain any interface or CoClass definitions because we don't have any yet.

SecurityMgr.rc This is the project's resource file. It contains a string table with the string `IDS_PROJNAME` defined as "SecurityMgr." It also contains version information.

StdAfx.cpp This source file includes `stdafx.h`, which results in a precompiled header with the common system includes.

Header FilesThe Resource.h file contains the resource ID definitions. Initially, it contains the resource ID for the project name resource.

In the StdAfx.h file, the first constant definition is _WIN32_WINNT 0x0400, which means that this application is intended to run on Microsoft Windows NT 4 (or later) systems. The main distinction of Windows NT 4 systems is that they have built-in DCOM support. If you intend to run this application on Windows 98 or Windows 95 systems with DCOM support installed, replace this with _WIN32_DCOM. If your application must run on an older version of Windows that does not have DCOM support installed, you need to remove the #define.

The next definition in the StdAfx.h file is for _ATL_APARTMENT_THREADED, which is the default threading model of our DLL. Next, the #include for atlbase.h brings in many of the ATL class declarations, including CComModule, which is used in the next line. _Module is declared as an extern CComModule. This is the global instance of CComModule that we saw in the SecurityMgr.cpp file.

The system header file atlcom.h contains many of the ATL templates, such as CComClassFactory, CComRootObjectEx, and CComObject.

So now that we know what files the ATL COM AppWizard created for us, let's see what we can do with them. We will start by adding a component to our COM server.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Adding a CoClass

So far, we have the COM server shell. We need to add a CoClass to it. To do this, follow these steps:

1. Select Insert Ø New ATL Object. This starts the ATL Object Wizard, which lets you add different kinds of ATL objects.
2. For the simple component we are creating, choose Simple Object (how appropriate!) and click Next. The ATL Object Wizard Properties dialog box appears, as shown in Figure 22.5. This is where you specify the name of your component.



FIGURE 22.5 Using the ATL Object-Wizard to add a CoClass

3. Type **AccessControl** in the Short Name field. You will see the other fields default to show the appropriate names.
4. Click OK to accept the defaults.

The following are the other fields on the Names tab of the ATL Object Wizard Properties dialog box:

- Class is the name of the C++ class providing the implementation.
- The .H File and the .CPP File are the header and source files for the C++ class.
- CoClass is the component's name (as defined in the IDL file).
- Interface shows the interface name.
- Type is used as a CoClass description when adding the necessary key to the Registry.
- ProgID is the component's Programmatic ID, which allows a CoClass to be created using this ID rather than its CLSID (which is usually much longer).

The Attributes tab, shown in Figure 22.6, gives you control over many component attributes.



FIGURE 22.6 The Attributes tab of the ATL Object Wizard Properties dialog box allows you to specify attributes such as threading model and dual or custom interface.

Adding a Property

The ATL Object Wizard will add some new files and modify some existing ones. We will examine the code it generated later.

Let's continue building our component.

1. In the ClassView, you will see a newly added interface called `IAccessControl` (this is what we called our interface when we added the object). Right-click this interface and select **Add Property**. The **Add Property to Interface** dialog box appears (Figure 22.7).



FIGURE 22.7 Selecting **Add Property** from the pop-up menu

2. Select **BSTR** from the **Property Type** drop-down list. Then type **UserName** in the **Property Name** field, as shown in Figure 22.8.



FIGURE 22.8 Specifying the property data type and name in the **Add Property to Interface** dialog box

3. Click the **Attributes** button, change the helpstring attribute to read “Sets or retrieves UserName,” and then click **OK**.
4. Click **OK** to close the **Add Property to Interface** dialog box.

The `UserName` property is a string variable, which is used to set the name of the current user. This property is used in conjunction with the method `IsAllowed`, which we will add shortly.

You'll notice that we've defined the property type to be `BSTR`. A `BSTR` is a special kind of string used by COM. `BSTR` values have one key feature that regular null-terminated strings lack: The `DWORD` preceding a `BSTR` in memory carries the size of the `BSTR`. This feature is used by COM when marshaling `BSTR` values between clients and servers. The way you use `BSTR` values is also a little different: You allocate a `BSTR` using the function `SysAllocString`, with its parameter being a null-terminated string that is copied to the newly allocated string. To free a `BSTR`, you use `SysFreeString`.

Implementing a Property

Now, let's implement the `UserName` property.

1. From the **FileView**, open the file `AccessControl.h`. This is where our C++ implementation class is declared.
2. At the end of the class definition, add a private member variable of type `BSTR`:

```
private:
    BSTR m_bstrUserName;
```

We will use this member variable to hold the value of the `UserName` property.

3. In the class constructor, add code to initialize this variable to 0:

```
CAccessControl() : m_bstrUserName(NULL)
{
}
```

4. Open the file AccessControl.cpp. In this file, you will find the following two functions:

```
STDMETHODIMP CAccessControl::get_UserName(BSTR *pVal)
{
    return S_OK;
}
```

```
STDMETHODIMP CAccessControl::put_UserName(
                                   BSTR newVal )
{
    return S_OK;
}
```

These functions were added by the ATL Object Wizard. Note that to be implemented as a read/write property, UserName requires two functions: get_UserName (to read the property) and put_UserName (to write the property).

5. Type the following code in the function implementations (in AccessControl.cpp):

```
STDMETHODIMP CAccessControl::get_UserName(BSTR *pVal)
{
    // allocate and return a new BSTR.
    // Remember that COM requires clients to
    // free [out] params
    *pVal = SysAllocString( m_bstrUserName );
    return S_OK;
}

STDMETHODIMP CAccessControl::put_UserName(BSTR newVal)
{
    // free the private BSTR member variable
    if( m_bstrUserName )
        SysFreeString( m_bstrUserName );

    // allocate the private BSTR member variable
    // using the new value
    m_bstrUserName = SysAllocString( newVal );
    return S_OK;
}
```

The get_UserName function simply allocates a new BSTR and copies to it the private member variable m_bstrUserName. This new BSTR is returned via the out parameter pVal. The put_UserName function frees the existing member variable m_bstrUserName and allocates a new BSTR, copying to it the in parameter newVal. Both functions return S_OK.

NOTE:

In general, you should allocate out parameters in the server (since this is where you know how large they need to be) and free them in the client after you are finished using them. The client cannot send in a pointer to an allocated BSTR, because it has no way of knowing how long the returned string will be.

Now let's see how to add methods.

Adding Methods

Adding methods is just as easy as adding properties. We will add the method IsAllowed, which returns TRUE if the user (as determined by the UserName property) has access to the requested function. A client must set the

UserName property first, then call this method specifying a requested function. This allows a client to create an instance of our AccessControl component, set the UserName property once, then use the IsAllowed method to check access to different application functions.

1. In ClassView, right-click the IAccessControl interface and select Add Method to open the Add Method to Interface dialog box.
2. Type **IsAllowed** in the Method Name field.
3. In the Parameters field, we need to type a list of method parameters, as shown in Figure 22.9. These are the parameters for the IsAllowed method:

```
[in] short ReqFunction, 0  
[out,retval] VARIANT_BOOL *pbIsAllowed
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

NOTE:

Notice that we defined `pisAllowed` as `[out,retval]`. Tools such as VB interpret this as the function return value, which makes the semantics of calling this function a little easier. For example, a VB developer would call this as `if IsAllowed(iReqFunc) Then...` Although all our COM methods/properties return `HRESULT` values, VB developers have no way of getting at this return value. Also notice that we are using the data type `VARIANT_BOOL` as the return value instead of `BOOL`. This is because we would like our server to be usable by VB, which sees the type `BOOL` as a `LONG` (a Win32 `DWORD`). `VARIANT_BOOL` is recognized by VB as a Boolean data type.



FIGURE 22.9 Specifying the method name and parameters in the Add Method to Interface dialog box

4. Click the Attributes button and set the helpstring attribute to "Returns TRUE if the user has access to the requested function, false otherwise." Then click OK.
5. Click OK to close the Add Method to Interface dialog box.
6. Before we implement this method, let's add another method called `IsUserAllowed`. This is similar to the `IsAllowed` method, except it takes the username as the first parameter. Right-click the interface again and select Add Method.
7. Type **`IsUserAllowed`** for the Method Name. Enter the following in the Parameters field:

```
[in] BSTR UserName, [in] short ReqFunction, 0
[out, retval] VARIANT_BOOL *pbIsAllowed
```

Note the extra first parameter `UserName`. This method can be used to check whether a user has access to a particular function without first setting the `UserName` property. Having this method allows our component to be stateless, which comes in handy when we use COM+.

8. Click the Attributes button and set the helpstring attribute to "Returns TRUE if the user specified by

UserName has access to the requested function, false otherwise.” Then click OK.

9. Click OK to close the Add Method to Interface dialog box.

Implementing the Methods

To implement our new methods, go to the `AccessControl.cpp` file and look for the newly added class methods.

Type in the following code for the `IsUserAllowed` method:

```
STDMETHODIMP CAccessControl::IsUserAllowed( BSTR UserName,
      short ReqFunction, VARIANT_BOOL *pbIsAllowed )
{
    // We will allow everyone access to everything for now
    *pbIsAllowed=VARIANT_TRUE;
    return S_OK;
}
```

As you can see, the method implementation is trivial. It simply sets the `pbIsAllowed` parameter to `VARIANT_TRUE`, which means everyone will be allowed access to all application functions—not very secure is it? Of course, this is just our current implementation. We will update this method when we learn more about COM+ security in Chapter 23, “New COM Features in Windows 2000.”

The `IsAllowed` implementation has a little more code, but it is still simple:

```
STDMETHODIMP CAccessControl::IsAllowed(
      short ReqFunction,
      VARIANT_BOOL *pbIsAllowed )
{
    // make sure the username member variable
    // is not a null pointer
    if( ! m_bstrUserName )
        return E_INVALIDARG;
    else
        // call IsUserAllowed passing in m_bstrUserName
        // as the first parameter
        return IsUserAllowed( m_bstrUserName, ReqFunction,
                              pbIsAllowed );
}
```

The method first checks if the member variable `m_bstrUserName` is `NULL`. If it is `NULL`, we don’t have a username to check access for, so we return `E_INVALIDARG` (this is not strictly an invalid argument type of error, but it’s close enough for now).

Building the DLL

Now we are done with the server. You can go ahead and build the DLL. At the end of the build process, `REGSVR32.EXE` is invoked automatically to register the newly built DLL.

This completes building our COM server. Now that we’ve seen how easy it is to create a simple COM server using ATL, let’s examine what the wizards were doing behind the scenes and how all the parts fit together. (If you are happy with just building COM servers and don’t care about the theory behind it, feel free to skip to the section “Testing Our ATL Server.”)

Going Behind the Scenes

As you may have guessed, each part of the process (adding a `CoClass`, adding a property, and adding methods) results in the addition of files to the project and/or modification to existing files. Let’s look at each of these processes and the resulting changes to our project.

Adding a `CoClass` This step causes the most changes to the project and with good reason—we are adding a whole new COM component to our server.

Do you remember what we needed to do when we created a COM server with raw C++? Let me refresh your

memory:

- We defined our interface and our CoClass in the IDL file.
- We created a C++ class (which was inherited from our custom interface) to implement our interface methods and properties.
- We wrote the necessary code to register our server in the Windows Registry.

Since we didn't do any of this work here, someone must have done it for us! In fact, the ATL Object Wizard added the necessary code to accomplish all of this. The only thing we needed to type in was the C++ short name.

After we add the simple ATL object, our IDL file contains an interface definition for the `IAccessControl` interface, which is initially empty. Because we accepted Dual, which is the default in the Interface section of the Attributes tab, our `IAccessControl` interface inherits from the `IDispatch` interface. (We will talk more about `IDispatch` later in this chapter, in the "Automation and `IDispatch`" section.) In addition to the `IAccessControl` interface, the wizard also adds the `AccessControl` CoClass definition to the IDL file. Before we add any methods or properties, the IDL file looks like this:

```
import "oaidl.idl";
import "ocidl.idl";
[
    object,
    uuid(6497C034-CF10-11D2-91AE-0008C7FE9130),
    dual,
    helpstring("IAccessControl Interface"),
    pointer_default(unique)
]
interface IAccessControl : IDispatch
{
};

[
    uuid(63098966-CEA5-11D2-91AD-0008C7FE9130),
    version(1.0),
    helpstring("SecurityMgr 1.0 Type Library")
]
library SECURITYMGRLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(6497C035-CF10-11D2-91AE-0008C7FE9130),
        helpstring("AccessControl Class")
    ]
    coclass AccessControl
    {
        [default] interface IAccessControl;
    };
};
```

As part of adding the simple ATL object, the wizard creates C++ header and source files, called `AccessControl.h` and `AccessControl.cpp`, respectively. These contain the C++ class `CAccessControl`, which provides implementation for the interface `IAccessControl`. The file `AccessControl.h` contains the following code:

```
class ATL_NO_VTABLE CAccessControl :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccessControl,
        &amp;CLSID_AccessControl>,
    public IDispatchImpl<IAccessControl,
        &amp;IID_IAccessControl, &amp;LIBID_SECURITYMGRLib>
{
```

```
public:
    CAccessControl()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_ACCESSCONTROL)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CAccessContrsol)
    COM_INTERFACE_ENTRY(IAccessControl)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IAccessControl
public:
};
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The ATL_NO_VTABLE macro tells the compiler not to create a vtable for this class, thereby reducing the DLL size. We don't need a vtable for this class because it is never instantiated directly. What is instantiated is the template CComObject class.

As you can see, our implementation class inherits from three different classes:

- CComObjectRootEx is another class template, which takes in a class representing the desired component's threading model. CComObjectRootEx provides the IUnknown implementation and the component's threading model, including managing the reference count.
- CComCoClass provides the default aggregation behavior and defines the class factory.
- IDispatchImpl is a class template, which provides the IDispatch interface implementation.

The DECLARE_REGISTRY_RESOURCEID macro enables registration using a Registry script, which is created by the wizard as a project resource.

The DECLARE_PROTECT_FINAL_CONSTRUCT macro protects the object from causing itself to be deleted. To understand how this can happen, you need to understand FinalConstruct, which we will talk about as part of aggregation.

The BEGIN_COM_MAP and END_COM_MAP macros delimit the list of COM_INTERFACE_ENTRY macros. The COM map is the mechanism that allows CComRootObjectEx::InternalQueryInterface to return pointers to your interfaces. The wizard adds an entry for each of our interfaces (we have two: IAccessControl and IDispatch) in the COM map.

The wizard also created a Registry script file called AccessControl.rgs, with the contents shown here:

```
HKCR
{
    SecurityMgr.AccessControl.1 = s 'AccessControl Class'
    {
        CLSID = s '{6497C035-CF10-11D2-91AE-0008C7FE9130}'
    }
    SecurityMgr.AccessControl = s 'AccessControl Class'
    {
```

```

        CLSID = s '{6497C035-CF10-11D2-91AE-0008C7FE9130}'
        CurVer = s 'SecurityMgr.AccessControl.1'
    }
    NoRemove CLSID
    {
        ForceRemove {6497C035-CF10-11D2-91AE-0008C7FE9130}
            = s 'AccessControl Class'
        {
            ProgID = s 'SecurityMgr.AccessControl.1'
            VersionIndependentProgID =
                s 'SecurityMgr.AccessControl'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
            'TypeLib'
                = s '{63098966-CEA5-11D2-91AD-0008C7FE9130}'
        }
    }
}

```

This script is used to add or remove the Registry keys and values when registering or unregistering the component. It contains the CLSID, ProgID, and file path information (%MODULE%) needed to create appropriate Registry keys and values.

WARNING:

You should avoid manually editing this Registry script file. However, if you must, be careful not to delete the line `NoRemove CLSID`. Otherwise, when you unregister the component, it will remove the `HKEY_CLASSES_ROOT\CLSID` key from your Registry. The `HKEY_CLASSES_ROOT\CLSID` contains registration information for all the registered components on your system, so deleting it would be a disaster! One would hope that Microsoft changes this so that the `CLSID` key is better protected from inadvertent deletion

The `SecurityMgr.cpp` file is updated to include the new file `AccessControl.h` to bring in the definition of the `CAccessControl` class. The wizard also changed the object map in the `SecurityMgr.cpp` file. It added a new entry for our new `CoClass` and its implementation class `CAccessControl`.

```

BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_AccessControl, CAccessControl)
END_OBJECT_MAP( )

```

The object map enables the call to `_Module.RegisterServer` (called from `DllRegisterServer`) to register each COM component added to the map. That's why we must add an object map entry for each COM component provided by our server.

Adding a Property The extent of the changes is not so dramatic when adding a property. The changes begin with the interface definition in the IDL file. It is modified to reflect two new functions: one for the property put and one for the property get, as follows:

```

interface IAccessControl : IDispatch
{
    [propget, id(1),
        helpstring("Sets or retrieves UserName")]
        HRESULT UserName([out, retval] BSTR *pVal);
    [propput, id(1),
        helpstring("Sets or retrieves UserName")]
        HRESULT UserName([in] BSTR newVal);
};

```

The next change is to the AccessControl.h file, where we find the declaration of the two member functions, which implement this property:

```
STDMETHOD(get_UserName)(/*[out, retval]*/ BSTR *pVal);
STDMETHOD(put_UserName)(/*[in]*/ BSTR newVal);
```

Finally, the AccessControl.cpp file is also modified to include the implementation of the two member functions declared above:

```
STDMETHODIMP CAccessControl::get_UserName(BSTR *pVal)
{
    // TODO: Add your implementation code here
    return S_OK;
}

STDMETHODIMP CAccessControl::put_UserName(BSTR newVal)
{
    // TODO: Add your implementation code here
    return S_OK;
}
```

That's all you need to add the property. Of course, the actual implementation code is still up to you.

Adding a Method The modifications for adding a method are similar to those for adding a property. The wizard adds the method to the interface in the IDL file. The resulting interface definition looks like this:

```
interface IAccessControl : IDispatch
{
    [propget, id(1),
     helpstring("Sets or retrieves UserName")]
     HRESULT UserName([out, retval] BSTR *pVal);
    [propput, id(1),
     helpstring("Sets or retrieves UserName")]
     HRESULT UserName([in] BSTR newVal);
    [id(2),
     helpstring("Returns TRUE if the user has access 0
     to the requested function, false otherwise.")]
     HRESULT IsAllowed([in] short ReqFunction,
     [out, retval] VARIANT_BOOL *pbIsAllowed);
    [id(3),
     helpstring("Returns TRUE if the user specified 0
     by UserName has access to the requested 0
     function, false otherwise.")]
     HRESULT IsUserAllowed([in] BSTR UserName,
     [in] short ReqFunction,
     [out, retval] VARIANT_BOOL *pbIsAllowed);
};
```

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The header file AccessControl.h is also modified to show the new member functions, which provide implementation for the new interface methods:

```
STDMETHOD(IsUserAllowed)(/*[in]*/ BSTR UserName,  
                        /*[in]*/ short ReqFunction,  
                        /*[out, retval]*/ VARIANT_BOOL *pbIsAllowed);  
STDMETHOD(IsAllowed)(/*[in]*/ short ReqFunction,  
                    /*[out, retval]*/ VARIANT_BOOL *pbIsAllowed);
```

And the AccessControl.cpp file has the implementation of the same two new member functions:

```
STDMETHODIMP CAccessControl::IsAllowed( short ReqFunction,  
                                       VARIANT_BOOL *pbIsAllowed )  
{  
    // TODO: Add your implementation code here  
  
    return S_OK;  
}  
  
STDMETHODIMP CAccessControl::IsUserAllowed( BSTR UserName,  
                                             short ReqFunction,  
                                             VARIANT_BOOL *pbIsAllowed )  
{  
    // TODO: Add your implementation code here  
  
    return S_OK;  
}
```

Similar to the way it handles properties, the wizard creates all the code we need for our interface method, except the method implementation itself.

Testing Our ATL Server

Now it's time to put our work to the test. We can create a test client in many ways (using any language that can call COM components). But since this chapter is for VC++ developers, and since we want to learn how to use as well as create COM components, we'll create a test client using MFC.

Creating an MFC Client We will use the MFC AppWizard to create our test client.

1. Start by creating a new MFC AppWizard project. Call it MFCTestClient. Select the Dialog Based radio button, as shown in Figure 22.10.

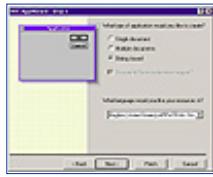


FIGURE 22.10 Creating a dialog-based MFC project

2. Click Finish, and then click OK in the New Project Information dialog box.
3. Go to the Resources tab and double-click the dialog called IDD_MFCTESTCLIENT_DIALOG.
4. Select the Button control in the Controls toolbox and draw a button in the dialog.
5. Right-click the button and select Properties. Change its caption to **&Test Server**, as shown in Figure 22.11.

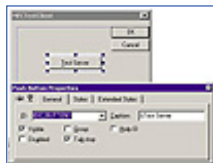


FIGURE 22.11 Changing the button's caption

6. Double-click the button to add a message handler to the BN_CLICKED message. Call the handler OnButton1Click. Add the code shown in the following listing to the handler function.

```
// Create an instance of the SecurityMgr.AccessControl
// and get a pointer to IAccessControl
IAccessControl* pAccessControl=NULL;
HRESULT hr = CoCreateInstance( CLSID_AccessControl,
                                NULL,
                                CLSCTX_INPROC_SERVER,
                                IID_IAccessControl,
                                reinterpret_cast<void*> (&pAccessControl));
//Check return code
if( FAILED( hr ) )
{
    MessageBox( "Could not create instance",
                "Error in CoCreateInstance",
                MB_OK + MB_ICONSTOP );
}
else
{
    // Use the interface
    // Allocate and initialize the parameters
    BSTR bstrUserName =
        SysAllocString( L"Any user name" );
    long lReqFunction = 3;
    VARIANT_BOOL Result = VARIANT_FALSE;

    // Make the call
    pAccessControl->IsUserAllowed( bstrUserName,
                                   lReqFunction,
                                   &Result );

    // Check result
    if( Result == VARIANT_TRUE )
```

```

        MessageBox( "User is allowed access",
                    "Access Check",
                    MB_OK | MB_ICONINFORMATION );
    else
        MessageBox( "User is denied access",
                    "Access Check",
                    MB_OK | MB_ICONSTOP );

    // Release the reference to the interface
    pAccessControl->Release();

    //Free the BSTR
    SysFreeString( bstrUserName );
}

```

Let's stop for a moment and examine this code. First, we use `CoCreateInstance` to create a new instance of our `AccessControl` component and get a pointer to the `IAccessControl` interface it exposes. Next, we declare and allocate a `BSTR` and declare a `LONG` and a `VARIANT_BOOL` to hold the result. We then use the returned interface pointer to call `IsUserAllowed`. After checking and reporting the result, we release the interface (we are no longer using it). Finally, we free the `BSTR` parameter. If we wanted, we could keep the interface pointer and use it to make other calls within our program. We just need to make sure that we release the reference to the interface when we no longer need it and before our program exits.

Editing the .cpp Files Add the following two includes in the `MFCTestClientDlg.cpp` file:

```

#include "..\SecurityMgr\SecurityMgr.h"
#include "..\SecurityMgr\SecurityMgr_i.c"

```

These includes bring in the `CLSID`, `IID`, and interface method declarations needed to compile our client. Open the `MFCTestClient.cpp` file and find the `CMFCTestClientApp::InitInstance()` method. Add a call to `CoInitialize()` before creating the dialog and a call to `CoUninitialize()` before returning from the function, as shown in the following fragment.

```

.
.
.
    CoInitialize(NULL);
    CMFCTestClientDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if( nResponse == IDOK )
    {
        // TODO: Place code here to handle when the dialog
        // is dismissed with OK
    }
    else
    if( nResponse == IDCANCEL )
    {
        // TODO: Place code here to handle when the dialog
        // is dismissed with Cancel
    }

    CoUninitialize();
.
.
.

```

Building and Running the Test Client Now we are ready to build and run our test client. When the test client's dialog box appears, click the Test Server button. You should get a message box (as shown) confirming that the user is allowed access to the requested function (remember that our current version of `IsUserAllowed()` always returns `TRUE`).



Building Out-of-Process COM Servers with ATL

So far, you've learned how to build COM servers that reside in DLL files. When a client calls such a server, it is loaded into the same process as the client. Sometimes you may want to have a COM server that runs in its own process and serves many clients. This can be especially useful if that server handles access to some kind of back-end resource, such as a legacy mainframe application. Such a server is referred to as an *out-of-process* server or a *local* server and is implemented (on Win32 systems) as an EXE file.

ATL helps with building out-of-process servers, just as it does with in-process servers. To see this in action, we will build the same server as we did earlier in this chapter, but this time we will build it in an EXE file instead of a DLL file.

To build an out-of-process server, create a new ATL COM AppWizard project and call it **SecurityMgrExe**. Select Executable (EXE) as the Server Type in the ATL COM AppWizard dialog box, as shown in Figure 22.12. Then click Finish.



FIGURE 22.12 Using the ATL COM AppWizard to create an EXE-based server

Examining the Out-Of-Process Server Differences

Let's look at the differences between what the wizard creates for out-of-process and in-process servers.

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

File Creation Differences For our EXE-based server, the wizard creates the same files as it did for the DLL-based server, with one exception: no .def file. This is because EXE-based COM servers do not export standard APIs as do DLL-based COM servers. Here's why:

- DLLRegisterServer and DLLUnregisterServer are not needed because the server registers itself if you run it with the /RegServer command-line parameter and it unregisters itself with the /UnregServer parameter.
- The DllGetClassObject API is not needed because the executable creates and registers its objects with the COM system as soon as it is loaded. Therefore, there is no need to get a class factory by calling DllGetClassObject.
- DllCanUnloadNow is not needed because the executable is supposed to handle its own unloading. So, if the executable thinks it's okay for it to unload, it should go ahead and unload itself.

As you can see, the standard APIs exported by DLL-based COM servers are not needed for EXE-based ones.

StdAfx.h File Differences A second difference can be found in the StdAfx.h file, which contains the following code after the line #include atlbase.h:

```
class CExeModule : public CComModule
{
public:
    LONG Unlock();
    DWORD dwThreadId;
    HANDLE hEventShutdown;
    void MonitorShutdown();
    bool StartMonitor();
    bool bActivity;
};
extern CExeModule _Module;
```

Here, _Module is an instance of CExeModule, rather than of CComModule as in the DLL version of the server. CExeModule inherits from CComModule and overrides the Unlock() method. In addition, it defines two methods: MonitorShutdown and StartMonitor.

This new derived class handles monitoring the locks on the module and determining when the EXE can

unload itself. We will see the CExeModule implementation in SecurityMgrExe.cpp.

SecurityMgrExe.cpp Differences Open the module implementation file, SecurityMgrExe.cpp. Scroll down until you find the function `_tWinMain`. This function is called when the executable is run and proceeds as follows:

- `_tWinMain` initializes COM by a call to `CoInitializeEx` or `CoInitialize`. This is necessary because the executable is running in a process of its own, so no one has initialized COM for us.
- The function calls `_Module::Init`, similar to what the DLL version did.
- It processes the command line, looking for a `/` or a `-`, followed by `UnregServer` or `RegServer`. If it finds either of these symbols, it will register or unregister the server as needed. If neither parameter is specified, then `bRun` will be `TRUE` after the while loop exits.
- It calls `_Module.StartMonitor`, which is a CExeModule method implemented just a few lines previously in the same source file.
- `StartMonitor` creates an event called `hEventShutdown`, then spawns a thread running `MonitorProc`, which simply calls `_Module.MonitorShutdown` to monitor this event. The event is set in the `CExeModule::Unlock` method if the lock count goes down to 0. When the event is set, the server waits in idle state for `dwTimeOut` (defined to be 5000 milliseconds) before shutting down.
- After starting the call to `_Module.StartMonitor`, `_tWinMain` continues by registering the objects with the class table by calling `_Module.RegisterClassObjects`. Clients use the class table (indirectly through the COM system) to get access to interface pointers exposed by these objects.
- The function goes into a loop, receiving and processing messages.
- When it is time to shut down, the `MonitorShutdown` method posts a `WM_QUIT` message, which is caught by this message-processing loop and causes the server to quit.
- Objects are revoked from the class table using `_Module.RevokeClassObjects`.
- Finally, there is a call to `_Module.Term` to free up data members and to `CoUninitialize` to uninitialize COM.

Adding CoClasses, Properties, and Methods

The steps for adding CoClasses, properties, and methods for EXE-based servers are identical to those for DLL-based servers. Follow the steps in the section “Building an In-Process COM Server with ATL” earlier in this chapter to add a simple ATL object called `AccessControl`, the `UserName` property, and the `IsAllowed` and `IsUserAllowed` methods. When you are finished, build your project and get ready to test it.

Testing Our Out-of-Process Server

We can test our out-of-process server by simply making the following two changes to our `MFCTestClient`:

- Change the include statements in `MFCTestClientDlg.cpp` to point to the `SecurityMgrExe.h` and `SecurityMgrExe_i.c` files.
- Change the third `CoCreateInstance()` parameter from `CLSCTX_INPROC_SERVER` to `CLSCTX_LOCAL_SERVER`.

Now we are ready to test our out-of-process server using the `MFCTestClient`. To make things more interesting, let’s see how to use our `SecurityMgrExe` server from a VB client.

1. Start VB and create a new Standard Exe project. Save the project as `VBTestClient.vbp` and the form as `frmMain.frm`.
2. Select Project Ø References. Scroll down until you find the `SecurityMgrExe 1.0 Type Library`, as shown in Figure 22.13. Check this item and click OK to add a reference to our server’s type library (we will talk more about type libraries later in this chapter).



FIGURE 22.13 Adding a reference to the `SecurityMgrExe` type library in our VB project

3. Select the Button control in the toolbox and draw a button on the form.
4. Double-click the button, and in the `Command1_Click()` event, add the code shown in the listing that

follows this procedure.

5. Start your project by pressing F5.

6. When the form appears, click the button to load and test our server. You should get a message box saying that the user is allowed access. Our out-of-process server works and can even be called from VB!

The Command1_Click() event handler (from VBTestClient\frmMain.frm) is shown in the following listing:

```
Private Sub Command1_Click()  
    ' declare an object of type AccessControl  
    ' ( this points to the IAccessControl interface )  
    Dim objAC As SECURITYMGREXELib.AccessControl  
    ' Declare and initialize parameters  
    Dim bSuccess As Boolean  
    Dim sUserName As String  
    Dim lReqFunc As Long  
    sUserName = "Any User Name"  
    lReqFunc = 3  
  
    ' Create the object  
    Set objAC = New SECURITYMGREXELib.AccessControl  
  
    ' Call the method  
    bSuccess = objAC.IsUserAllowed(sUserName, lReqFunc)  
  
    ' check and report results  
    If bSuccess Then  
        MsgBox "User is allowed access", vbInformation, 0  
        "Access Check"  
    Else  
        MsgBox "User is denied access", vbCritical, 0  
        "Access Check"  
    End If  
  
    ' Free the object  
    ' ( call Release on the IAccessControl interface )  
    Set objAC = Nothing  
  
End Sub
```

Threads and COM Servers

NOTE:

Multithreaded applications were introduced in Chapter 6, "Creating and Synchronizing Multiple Threads."

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Now that you know the basics of building COM components, it's time to explore the more advanced concepts of multithreaded components. If you have no interest in multithreaded applications, you can skip this section. However, you should be aware that multithreaded applications can be very useful for user interface and server (business logic) components. We will not go into gory details about multithreaded application development; instead, we will focus on the concepts and capabilities of multithreading and COM components.

NOTE:

Multithreading is one of those things some people dread to the point where they don't even want to talk about it. Most of this fear stems from lack of practice and/or understanding. Effective multithreaded application programming is possible, even for mere mortals, provided they are armed with the proper knowledge and understanding

We will start our discussion with a brief review of multithreaded Win32 applications, then talk about multithreaded COM components and clients.

Multithreaded Win32 Applications

In a typical Win32 application, you may create several threads, each responsible for a specific task. For example, in a server application, you may create a number of threads and have each thread responsible for serving client requests that come in. This allows a certain amount of parallel processing and can boost application throughput, especially on multiprocessor machines.

As you create threads that go happily about their business, you soon realize that you have some data that needs to be shared. For example, a count of the number of client requests coming in would need to be incremented by each thread as it starts processing a new client request. You now must handle concurrency and reentrancy. To accomplish this, you need to perform thread synchronization, basically serializing access to certain parts of your application as well as communicating events between threads. There are several APIs and kernel objects that let you do this, including mutexes, semaphores, events, and waitable timers. The basic idea is to ensure that shared data (such as global variables) are accessed in a controlled manner by one thread at a time.

When you write a COM component, you have no idea what kind of threading models your clients will be using, so how do you decide on a threading model for your component? Read on to find the answer to this and

many more questions.

Threading COM Components

COM is about binary reuse. So you get this cool COM component, and you start developing around it. Your application is multithreaded, and you want to be able to call instances of this component from multiple threads. Can you do it? The answer is yes, but how it behaves depends on many things. To understand the various possible behaviors, we need to talk about COM threading models.

COM abstracts threads by introducing the concept of *apartments* (previously known as *execution contexts*). An apartment is a concept rather than an actual kernel object, such as a thread or a process, which makes it hard to grasp sometimes. Think of an apartment as a logical context of execution that contains one or more objects. If that is too abstract, think about it as a space confined by four walls just like a real-life apartment. The inhabitants of this apartment are one or more COM objects (people). Just like a person, each COM object can live in only one apartment, but one apartment can accommodate one or more COM objects.

Like people living in an apartment, COM objects sharing an apartment have some common rules. In the case of COM objects, the rules do not relate to using the kitchen or playing loud music—they relate to concurrency and reentrancy. All COM objects living in an apartment share the same concurrency and reentrancy constraints.

Unlike real apartments, there are only two types of COM apartments: a single-threaded apartment (STA) and a multithreaded apartment (MTA), which is sometimes referred to as a free-threaded apartment. COM controls which threads have “apartment keys” and can enter apartments. An STA has one set of keys and can therefore be entered by only one *particular thread*. An MTA has an unlimited number of keys and therefore allows *any number of threads* to enter and coexist in the apartment. Figure 22.14 illustrates an STA with its single thread executing and an MTA with any two threads executing.

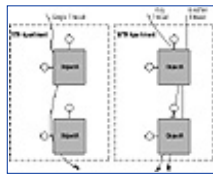


FIGURE 22.14 Single-threaded and multithreaded apartments

Now that we know the two kinds of apartments, let’s explore the behavior of each kind.

Single-Threaded Apartments

An object in an STA is safe from concurrent thread access. Furthermore, it is safe from serialized access by different threads, since only one particular thread is allowed to run within the apartment. This means that if multiple objects live in this apartment, only one object will ever be executing at any point in time. In addition, if other threads wish to communicate with objects within this apartment, they must go through COM. COM handles this situation by setting up a message loop with a message queue. If a thread needs to call a method of an object in the apartment, a message is posted to the apartment’s message queue. Messages in this queue are sequentially fetched and processed by the apartment’s message loop.

Multithreaded Apartments

An object in an MTA is accessible concurrently by any thread in the MTA. It is up to the object’s developer to protect the object’s state or data members from concurrent access through thread synchronization. The payback you get for going through the trouble of writing a thread-safe component is the ability to create and use worker threads to improve throughput or increase your component’s responsiveness to user input. MTAs do not need serialized access and therefore do not use message queuing as do STAs.

Object Threading Models and Calling Threads

As you already know, each Win32 process has one or more threads. A thread must enter an apartment before it can call COM components. To enter an apartment, the thread calls `CoInitialize()` or `CoInitializeEx()`. `CoInitialize` automatically places the thread in an STA. `CoInitializeEx` lets you specify which type of apartment you want via its second parameter (the possible values are `COINIT_APARTMENTTHREADED` or `COINIT_MULTITHREADED`). To exit from an apartment, the thread calls `CoUninitialize()`.

The first STA created within a process is referred to as the *primary STA*; subsequently created STAs are

referred to as *secondary STAs*. After entering an apartment, the thread can create COM objects by calling `CoCreateInstance()`, which, as we discussed before, takes care of instantiating an object and returning an interface pointer to the requested interface. That new object may be in the same apartment as the thread that created it, or it may be in a different apartment. If the object is in a different process (for example, if the COM server providing this object is an out-of-process, or EXE-based, server), then the object is automatically in a different apartment. If the COM object is in the same process as the creating thread (an in-process, or DLL-based, server), then COM decides whether or not to place it in the same apartment as the calling thread, depending on the object's threading model.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

When an out-of-process COM server starts, it calls `CoInitialize()` or `CoInitializeEx()` to enter an apartment. This means that an EXE-based COM server can decide the kind of apartment it wants to be in by calling `CoCreateInitializeEx` and passing the appropriate parameters. DLL-based servers, on the other hand, are created after COM has been initialized and, therefore, do not call `CoInitialize()`. So how does such a server control the kind of apartment it wants to be in? The answer lies in the Registry. Under the `InprocServer32` key, there is an optional value called `ThreadingModel`. This value, if present, can be set to one of the following:

- Single (or no value), which means this object must be placed in the primary STA of the calling process
- Apartment, which tells COM to place the object in an STA (not necessarily the primary one)
- Free, which means the object needs to be placed in an MTA
- Both, which means the object supports both threading models and can be placed in the same apartment as the calling thread

If the object's threading model is not the same as the calling thread's apartment, COM will place the object in a new apartment compatible with its threading model. To allow the calling thread and object to communicate, COM provides a layer between the two called *marshaling*. This is essentially the same mechanism used to communicate between a COM object and its client on a different machine. For now, remember that if the object and its calling thread are in different apartments, this involves an additional layer of COM between the two that adds overhead and may affect performance.

Table 22.1 shows the different combinations of object threading model and its calling thread's apartment and the type of communication between the two. *Direct* means the object and its calling thread communicate directly (because they are in the same apartment); *remoting* means they communicate through COM marshaling services (because they are in different apartments).

TABLE 22.1: Effect of the Object Threading Model and the Client's Apartment on How the Call Is Performed

Object Threading Model	Primary STA	Calling Thread Secondary STA	MTA
Single	Direct	Remoting	Remoting
Apartment	Direct	Direct	Remoting

Free	Remoting	Remoting	Direct
Both	Direct	Direct	Direct

ATL and Apartments

Now that we have reviewed the concepts, let's go back to creating COM components with ATL. Earlier in the chapter, you saw that the Attributes tab of the ATL Object Wizard Properties dialog box (Figure 22.6) offers the Threading Model choices of Single, Apartment, Both, and Free.

When you choose a threading model here, the generated code uses the appropriate template parameter for the `CComObjectRootEx` class template. In addition, if your COM server is a DLL, your selection here will determine the setting of the `ThreadingModel` value under your component's `Inproc32` Registry key. Recall that `CComObjectRootEx` provides the implementation for the `IUnknown` interface, which includes `AddRef` and `Release`.

`CComObjectRootEx<CComMultiThreadModel>` causes `AddRef` and `Release` to use `InterlockedIncrement()` and `InterlockedDecrement()`, respectively, to protect the reference count from concurrent access.

`CComObjectRootEx<CComSingleThreadModel>` causes `AddRef` and `Release` to use the `++` and `-` operators in C++, since the object will not be accessed by multiple threads.

At this point, you've specified your component's threading model. If you've chosen anything other than Single or Apartment, you must code your component to be thread-safe. As we mentioned earlier, thread-safe coding requires two things: thread synchronization and reentrancy. These are the same requirements as for thread-safe Win32 applications.

Automation and IDispatch

It's now time to explore how we can make our components accessible from a wide variety of clients, including those written in VBScript and other scripting languages.

The VC++ clients we've implemented so far included a header file containing the server's interface descriptions as an abstract class. This header file is needed to allow the compiler to figure out the vtable index for each of the interface methods. Languages like VB and VBScript (which are used heavily to automate applications like Microsoft Excel and Word) do not make use of header files. So how can they call interfaces exposed by COM components? The problem is solved by having applications expose a special interface that allows VB to automate those applications (VB5 and later can call custom interfaces directly, but VBScript still goes through the automation interface). This interface has two key characteristics:

- No type checking is performed at development time. At runtime, the interpreter gets the necessary type information and performs type checking.
- By limiting parameter and return value data types to a well-known set of predefined types, we make it easier for scripting languages such as VB to call these interfaces. This set of predefined types is known as Automation data types.

The interface is called `IDispatch`. Its purpose is to allow all calls to a COM component to go through a standard, well-known interface, regardless of the actual methods and properties exposed by the component. Let's explore how this works.

IDispatch Implementation with VC++

As you learned in Chapter 21, `IDispatch` is a standard COM interface enabling Automation. Automation clients use the `IDispatch` interface to get type information about the methods and properties implemented by the object, translate a method or property name to a unique dispatch ID (`DISPID`), and finally use that `DISPID` to invoke the required method or property. The listing following shows the `IDispatch` interface description in the file `oaidl.idl`.

```
[
    object,
    uuid(00020400-0000-0000-C000-000000000046),
    pointer_default(unique)
]
interface IDispatch : IUnknown
{
```

```

typedef [unique] IDispatch * LPDISPATCH;

HRESULT GetTypeInfoCount( [out] UINT * pctinfo );

HRESULT GetTypeInfo( [in] UINT iTInfo,
                    [in] LCID lcid,
                    [out] ITypeInfo ** ppTInfo );

HRESULT GetIDsOfNames( [in] REFIID riid,
                      [in, size_is(cNames)] LPOLESTR * rgpszNames,
                      [in] UINT cNames,
                      [in] LCID lcid,
                      [out, size_is(cNames)] DISPID * rgDispId );

[local]
HRESULT Invoke( [in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS * pDispParams,
                [out] VARIANT * pVarResult,
                [out] EXCEPINFO * pExcepInfo,
                [out] UINT * puArgErr );

```

[Previous](#)
[Table of Contents](#)
[Next](#)

[HOME](#)
[SUBSCRIBE](#)
[SEARCH](#)
[FAQ](#)
[SITEMAP](#)
[CONTACT US](#)

[Products](#) |
 [Contact Us](#) |
 [About Us](#) |
 [Privacy](#) |
 [Ad Info](#) |
 [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The first method of interest is `GetIDsOfNames`, which is used to translate method and property names to their DISPID. The method takes in an array of names (`rgszNames`) and returns the corresponding DISPID in the array `rgDispId`.

Once the client has the DISPID, it will want to call methods and properties on the COM component. Imagine yourself designing a flexible function that allows you to specify which method or property you want to call and then calls it for you. Obviously, you would have to pass this function something to identify the method or property you would like to call; in this case, the identifier would be the DISPID. You would also need to pass it arguments expected by the method or property. To allow maximum flexibility, you would want to pass these arguments as some form of self-describing array of values. In addition, you would want to get back return values as well as error information. Again, these must be communicated in a generic form to allow maximum flexibility in calling methods and properties. That's exactly what the `IDispatch::Invoke` method does.

Due to its flexibility, the `IDispatch::Invoke` method is a bit more involved. It has the following parameters:

dispIdMember The DISPID of the method or property

pDispParams A structure of type `DISPPARAMS`, containing an array of arguments and the number of arguments

pVarResult A pointer to the location where the return value is to be stored (more on the `VARIANT` data type later in this chapter, in the "Automation Data Types" section)

pExcepInfo Used to communicate errors (exceptions) back from the server

puArgErr Used to specify the first argument within `pDispParams->rgvarg` that has an error (if any)

TIP:

When creating the argument array in `pDispParams`, remember that the arguments go into this array in reverse order. So the last method argument would be the first array element and the first method argument would be the last array element.

In Chapter 21, we pointed out that `IDispatch` was designed for use by automation languages such as VB and VBScript. Although it can be used by VC++, it takes much more work than accessing custom interfaces. Custom interfaces offer ease of use and speed for the VC++ developer. `IDispatch` interfaces offer ease of use for the script developer. Can we keep both developers happy? Is it possible to get the best of both worlds?

So far, we haven't looked at objects that implement the IDispatch interface. It turns out there are a couple of different ways an object can implement IDispatch. To understand the difference between the two, we need to talk about dispinterfaces.

Dispinterfaces

A *dispinterface* (derived from dispatch interface) is an interface that supports a set of methods and properties callable through IDispatch. So, any object that allows access to its methods and properties through IDispatch is exposing a dispinterface.

As an example, consider the following sample code, which creates an instance of the fictitious object SpaceCrafts.XWing, then calls the FireMissile method on it. The FireMissile method takes no parameters and returns a Boolean value indicating whether the firing was successful:

```
//first initialize COM
CoInitialize(NULL);
    // get the component's clsid from the progid
    // we could use the clsid directly if we know it
    CLSID clsid;
    CLSIDFromProgID( L"SpaceCrafts.XWing", &clsid );

    // create the component and
    // get an IDispatch interface on it
    IDispatch* pIDispatch = NULL;
    CoCreateInstance( clsid, NULL,
                      CLSCTX_INPROC_SERVER,
                      IID_IDispatch,
                      (void**) &pIDispatch );
    // get the DISPID of the FireMissile method
    DISPID dispid;
    OLECHAR FAR* szMemberName = L"FireMissile";
    pIDispatch->GetIDsOfNames( IID_NULL,
                              &szMemberName,
                              1,
                              LOCALE_SYSTEM_DEFAULT,
                              &dispid );

    // the method takes no arguments
    DISPPARAMS dispparams= { NULL, NULL, 0, 0 };

    // prepare the result (return value)
    VARIANT Result;
    Result.vt = VT_BOOL;
    Result.boolVal = VARIANT_FALSE;

    //exception information
    EXCEPINFO excep;
    //argument error information
    UINT uArgErr;
    //Make the call
    pIDispatch->Invoke( dispid,
                       IID_NULL,
                       LOCALE_SYSTEM_DEFAULT,
                       DISPATCH_METHOD,
                       &dispparams,
                       &Result,
                       &excep,
                       &uArgErr );

    //Uninitialize COM
    CoUninitialize();
```

Figure 22.15 shows the XWing object exposing three methods (FireMissile, FireThrust, and Brake) through

Dual Interfaces

The diagram illustrates the system architecture. On the left, a 'Data Multi-Reg. Memory' block contains a vertical list of registers: 'id register', 'name', 'phone', 'Mailing address', 'last name', 'A/C Number', 'E-mail', 'M/credit', 'password', and 'status'. An 'id' input is connected to the 'id register'. This memory block is connected to a 'Register' block on the right, which contains a 'data' register. The 'data' register is connected to an 'M/Processor' block.

So what do we gain from dual interfaces? Dual interfaces give the VC++ programmer direct access to the object's custom interface. This custom interface can be called directly using `pInterface->Method()`. By exposing an `IDispatch` interface, the same object is equally accessible from scripting and Automation clients such as VBScript. This is indeed the best of both worlds!

Automation with ATL

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

When we created our SecurityMgr.AccessControl object, we chose the dual interface option (on the Attributes tab of the ATL Object Wizard dialog box). The resulting IDL specifies that our custom interface IAccessControl inherits from IDispatch, as is shown in the following sample code:

```
[
    object,
    uuid(6497C034-CF10-11D2-91AE-0008C7FE9130),
    dual,
    helpstring("IAccessControl Interface"),
    pointer_default(unique)
]
interface IAccessControl : IDispatch
{
    [propget, id(1),
        helpstring( "Sets or retrieves UserName" ) ]
    HRESULT UserName([out, retval] BSTR *pVal);
    [propput, id(1),
        helpstring( "Sets or retrieves UserName" ) ]
    HRESULT UserName([in] BSTR newVal);
    [id(2),
        helpstring( "Returns TRUE if the user has "0
                    "access to the requested function, "0
                    "false otherwise." ) ]
    HRESULT IsAllowed( [in] short ReqFunction,
        [out, retval] VARIANT_BOOL *pbIsAllowed);
    [id(3), helpstring( "Returns TRUE if the user "0
                        "specified by UserName has "0
                        "access to the requested "0
                        "function, false otherwise." ) ]
    HRESULT IsUserAllowed( [in] BSTR UserName,
        [in] short ReqFunction,
        [out, retval] VARIANT_BOOL *pbIsAllowed );
}
```

```
};
```

The header file `AccessControl.h` is also modified to show the new member functions, which provide implementation for the new interface methods:

```
STDMETHOD(IsUserAllowed)( /*[in]*/ BSTR UserName,
                          /*[in]*/ short ReqFunction,
                          /*[out, retval]*/ VARIANT_BOOL *pbIsAllowed );
STDMETHOD(IsAllowed)( /*[in]*/ short ReqFunction,
                      /*[out, retval]*/ VARIANT_BOOL *pbIsAllowed );
```

And the `AccessControl.cpp` file has the implementation of the same two new member functions:

```
STDMETHODIMP CAccessControl::IsAllowed( short ReqFunction,
                                         VARIANT_BOOL *pbIsAllowed )
{
    // TODO: Add your implementation code here
    return S_OK;
}

STDMETHODIMP CAccessControl::IsUserAllowed( BSTR UserName,
                                              short ReqFunction,
                                              VARIANT_BOOL *pbIsAllowed )
{
    // TODO: Add your implementation code here
    return S_OK;
}
```

In addition, the `CAccessControl` class inherits from `IDispatchImpl<>`, as follows:

```
class ATL_NO_VTABLE CAccessControl :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CAccessControl,
                    &amp;CLSID_AccessControl>,
    public IDispatchImpl<IAccessControl,
                    &amp;IID_IAccessControl,
                    &amp;LIBID_SECURITYMGRLib>
```

The `IDispatchImpl<>` ATL template provides a default `IDispatch` implementation for use with any dual interface (no, we don't need to code our own `IDispatch` interface).

The only other thing a dual-interface developer must do is make sure to use only Automation-compatible data types (the subject of the next section). If you recall, we used `BSTR`, `VARIANT_BOOL`, and `LONG` values, all of which are Automation-compatible data types. What does this mean? It means our `AccessControl` object already supports `IDispatch` and we didn't even know about it! We will create an Automation client for our `AccessControl` object soon. But first, let's find out more about Automation data types.

Automation Data Types

Automation defines a few data types associated with the `IDispatch` interface. To make your COM object accessible from Automation clients such as VBScript, you will need to use these Automation data types as the parameters for your object's methods and properties.

BSTR

`BSTR` stands for Basic String. We talked about `BSTR` briefly when we created our first ATL COM server. The motivation behind creating `BSTR` values is to facilitate moving strings back and forth between COM clients and servers running in different processes (and possibly on different machines). The regular null-terminated string does not know its length and must therefore be counted one character at a time until a null (`\0`) character is reached. Although this works, it is certainly inefficient. Therefore, the `BSTR` was created with the count of

characters directly preceding the string in memory. But wait a minute—if we have the count of characters, that means we know exactly how long the BSTR string is. This has two interesting consequences.

The first consequence is that you can't assign a BSTR like this:

```
BSTR Mybstr = L"This will not work";
```

The count needs to be set first before the characters can be assigned. That's why we needed to use the SysAllocString API when we were assigning BSTRs in our AccessControl object.

The second interesting consequence is that we no longer need to terminate the string with a null. That's because we already know where the string ends, thanks to the count preceding the string. This means BSTR values may include embedded nulls. So be careful when reading a BSTR and guard against it having embedded nulls. Of course, if it makes sense for your application, you can still null-terminate it.

When working with BSTR values, remember to allocate them using SysAllocString and free them using SysFreeString. Also note that, unlike pointers to null-terminated strings, a null BSTR is valid; it simply means an uninitialized string. For example, SysFreeString will not complain if you pass it a null BSTR.

WARNING:

Since a BSTR points to the first character and not to the length prefix, to the compiler it looks like an OLECHAR*. So the compiler will let you send an OLECHAR* in place of a BSTR, which can cause all sorts of problems—especially when you try to free the parameter. Always use real BSTR values when dealing with BSTR parameters.

To make life easier for the COM programmer, ATL includes a CComBSTR class, which handles allocating and freeing BSTR values as well as giving you a few useful operators and string-manipulation functions, such as Append(), Copy(), and Length(). We recommend that you use CComBSTR instead of BSTR whenever you can.

CURRENCY

The CURRENCY data type is simply an 8-byte integer scaled by 10,000. The CURRENCY type definition is:

```
typedef CY CURRENCY;  
  
typedef struct tagCY  
{  
    LONGLONG int64;  
} CY;
```

CURRENCY gives you numbers with fifteen digits to the left of the decimal point and four digits to the right. As its name implies, it is mostly used for monetary amounts.

SAFEARRAY

The SAFEARRAY data type allows passing of arrays of data to and from Automation objects. A SAFEARRAY carries an array of data, as well as information about that array, which makes it possible to access elements of the array without prior knowledge of the array. The following declaration of SAFEARRAY can be found in the Automation IDL file oaidl.idl:

```
typedef struct tagSAFEARRAY  
{  
    USHORT cDims;  
    USHORT fFeatures;  
    ULONG cbElements;  
    ULONG cLocks;  
    PVOID pvData;  
    SAFEARRAYBOUND rgsabound[ 1 ];  
} SAFEARRAY;
```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The parameters are as follows:

cDims Defines the number of dimensions in the array.

ffFeatures Describes the data type contained in the array and how it is allocated. This information is used when freeing the array. Allowed types are BSTR, VARIANT, IUnknown*, and IDispatch*.

cbElements Specifies the size of elements in the array.

cLocks Specifies the number of locks on the array (which is used to avoid deleting the array while others are using it).

pvData Acts as a pointer to the actual data array.

rgsabound[1] Defines the dimension's bounds (for each dimension, the rgsabound array has an element defining that dimension's lower and upper bounds).

There are several APIs provided to help you create and manipulate SAFEARRAY values, such as SafeArrayCreate, SafeArrayDestroy, SafeArrayAccessData, and SafeArrayUnaccessData. Be sure to use these functions to create and access SAFEARRAYs.

Finally, to declare parameters of type SAFEARRAY in your IDL file, use the SAFEARRAY(*type*) IDL data type. For example, an array of strings would be defined as SAFEARRAY(BSTR).

VARIANT

VARIANT is the most flexible Automation type. We've used VARIANT types, but we haven't really talked about what they are or why they are needed.

When calling an Automation server, the client needs to be able to pass different types of parameters through the rgvarg array passed to the IDispatch::Invoke method. The VARIANT data type accommodates this requirement. VARIANT is basically a union of the most common data types including pointers to IUnknown and IDispatch. Here is the definition of the VARIANT data type:

```
typedef struct FARSTRUCT tagVARIANT VARIANT;  
typedef struct FARSTRUCT tagVARIANT VARIANTARG;  
typedef struct tagVARIANT  
{
```



```

VARTYPE vt;
unsigned short wReserved1;
unsigned short wReserved2;
unsigned short wReserved3;
union
{
    unsigned char    bVal;        // VT_UI1
    short            iVal;        // VT_I2
    long             lVal;        // VT_I4
    float            fltVal;      // VT_R4
    double           dblVal;      // VT_R8
    VARIANT_BOOL     boolVal;     // VT_BOOL
    SCODE            scode;       // VT_ERROR
    CY               cyVal;       // VT_CY
    DATE             date;        // VT_DATE
    BSTR             bstrVal;     // VT_BSTR
    IUnknown         FAR* punkVal; // VT_UNKNOWN
    IDispatch        FAR* pdispVal; // VT_DISPATCH
    SAFEARRAY        FAR* parray;  // VT_ARRAY | *
    unsigned char    FAR* pbVal;   // VT_BYREF | VT_UI1
    short            FAR* piVal;   // VT_BYREF | VT_I2
    long             FAR* plVal;   // VT_BYREF | VT_I4
    float            FAR* pfltVal; // VT_BYREF | VT_R4
    double           FAR* pdblVal; // VT_BYREF | VT_R8
    VARIANT_BOOL     FAR* pboolVal; // VT_BYREF | VT_BOOL
    SCODE            FAR* pscode;  // VT_BYREF | VT_ERROR
    CY               FAR* pcyVal;  // VT_BYREF | VT_CY
    DATE             FAR* pdate;   // VT_BYREF | VT_DATE
    BSTR             FAR* pbstrVal; // VT_BYREF | VT_BSTR
    IUnknown         FAR* FAR* ppunkVal; // VT_BYREF | VT_UNKNOWN
    IDispatch        FAR* FAR* ppdispVal; // VT_BYREF | VT_DISPATCH
    SAFEARRAY        FAR* FAR* pparray; // VT_ARRAY | *
    VARIANT          FAR* pvarVal; // VT_BYREF | VT_VARIANT
    void             FAR* byref;    // Generic ByRef
};
};

```

As you can see, the VARIANT (or the VARIANTARG) consists of the discriminator, `vt`, which indicates the type of data carried in the VARIANT, and a union of data types. Most of the supported data types can be passed by reference (BYREF) or by value (BYVAL). “By reference” means you are passing a pointer to the variable, thereby allowing the server to change its value. “By value” means you are passing a copy of the variable. To specify the type of data carried in the VARIANT, set the `vt` discriminator to one of the supported data types, such as `VT_BOOL`. If the parameter is passed by reference, you OR the parameter type with the constant `VT_BYREF`, like this:

```
MyVariant.vt = VT_BOOL | VT_BYREF;
```

NOTE:

If you are calling a method with optional parameters, you still must have an element in the `pDispParams.rgvarg` array for each optional parameter. Set the parameters’ discriminator (`vt`) to `VT_ERROR`, and set the `SCODE` union member to `DISP_E_PARAM_NOTFOUND`.

More on Type Libraries

We saw how the MIDL compiler generates a type library as one of its output files. Type libraries are used for a couple of things: Automation objects register their type libraries to make use of the Universal Marshaler, and languages such as VB use type libraries to provide compiler-time type checking (equivalent to a header file in VC++).

As you know, a type library is a binary file. It can exist as a separate file (usually with a .tlb extension) or it may exist as a resource inside a DLL or EXE file. To programmatically examine the contents of a type library, you need an ITypeLib interface. You get this interface by calling the IDispatch::GetTypeInfo method. To know whether an object provides an ITypeLib interface, call the IDispatch::GetTypeInfoCount method. This method name is deceiving, because the method actually returns 1 if the object supports the ITypeLib interface or 0 if it doesn't.

To view your type library, you can use the OLE View utility, which comes with Visual Studio. The ITypeLib Viewer uses the ITypeLib interface to get information from the type library about the supported interfaces and their methods and properties, as well as the parameters of each method and property. In addition to being a useful tool for viewing type libraries, OLE View allows you to save type library information in an IDL file. Let's use the viewer now to look at our SecurityMgr type library:

1. Run the OLE View utility from the Microsoft Visual Studio Tools program folder. From the tree view on the left, open the Type Libraries folder.
2. Scroll down until you find the SecurityMgr type library. Right-click this library and choose View (see Figure 22.17.)

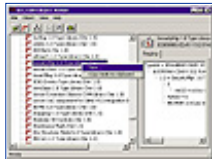


FIGURE 22.17 Opening the SecurityMgr type library

This launches the type library viewer, as shown in Figure 22.18.

3. Select File Ø Save As to save the SecurityMgr type library to an IDL file. Compare that file to the one we originally created when implementing the SecurityMgr. You'll find they are functionally equivalent.



FIGURE 22.18 Viewing type library information using ITypeLib Viewer

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

NOTE:

VB provides an Object Browser that uses the `TypeInfo` interface to visually represent interfaces, methods, and properties supported by a COM server. VB programmers make extensive use of this feature as a handy quick reference tool

A C++ Automation Client

We've been talking about Automation for a while now. Along the way, we found out that the `AccessControl` object we implemented already supports Automation. Let's create an Automation client for it to see what exactly is involved.

Start by creating a Win32 console application called `AutomationClient`. Add a .cpp file called `main.cpp` and add the code shown in the following sample:

```
#include <windows.h>

int main()
{
    //first initialize COM
    CoInitialize(NULL);

    //get the component's clsid from the progid
    //we could use the clsid directly if we know it
    CLSID clsid;
    CLSIDFromProgID( L"SecurityMgr.AccessControl", &clsid );
    //create the component and //get an IDispatch interface on it
    IDispatch* pIDispatch = NULL;
    CoCreateInstance( clsid, NULL, CLSCTX_INPROC_SERVER,
                     IID_IDispatch, (void**) &pIDispatch );
    //get the DISPID of the IsUserAllowed method
    DISPID dispid;
```

```

OLECHAR FAR* szMemberName = L"IsUserAllowed";
pIDispatch->GetIDsOfNames( IID_NULL,
                           &szMemberName,
                           1,
                           LOCALE_SYSTEM_DEFAULT,
                           &dispid );

    //prepare the arguments
DISPPARAMS dispparams;

dispparams.cArgs = 2;           //two arguments
dispparams.cNamedArgs = 0;      //no named arguments
dispparams.rgdispidNamedArgs = NULL;
dispparams.rgvarg = new VARIANTARG[2];
                           // the arg array contains two variants
// REMEMBER: arguments in the rgvarg array are reversed
// in order, so the last method argument is rgvarg[0] !
dispparams.rgvarg[0].vt = VT_I4;    //a 4-byte integer
dispparams.rgvarg[0].lVal = 3;
dispparams.rgvarg[1].vt = VT_BSTR;  //a BSTR
dispparams.rgvarg[1].bstrVal =
    SysAllocString( L"AnyUserName" ); //allocate the BSTR

    //prepare the result (return value)
VARIANTARG Result;

Result.vt = VT_BOOL;
Result.boolVal = VARIANT_FALSE;
    // we will set it to false
    // to see our method set it back to true
EXCEPINFO excep;    // exception information
UINT uArgErr;      // argument error information

    //Make the call
pIDispatch->Invoke( dispid,
                   IID_NULL,
                   LOCALE_SYSTEM_DEFAULT,
                   DISPATCH_METHOD,
                   &dispparams,
                   &Result,
                   &excep,
                   &uArgErr );
if( Result.boolVal == VARIANT_TRUE )
    MessageBox( NULL, "User is allowed access",
               "Access Check",
               MB_OK | MB_ICONINFORMATION );
else
    MessageBox( NULL, "User is denied access",
               "Access Check",
               MB_OK | MB_ICONSTOP );

    // Free the BSTR
SysFreeString( dispparams.rgvarg[1].bstrVal );
    // delete the argument array
delete[] dispparams.rgvarg;
    // Uninitialize COM
CoUninitialize();
return 0;
}

```

The first interesting point is that we are not including the server's header file. That's because we are using the IDispatch interface, which is defined in oaidl.h.

We start by initializing COM. Next, we get the component's CLSID using its ProgID. Here we are pretending we don't know the component's CLSID, which may be the case if you are reusing someone else's component. We then call CoCreateInstance to instantiate the component and get a pointer to its IDispatch interface. We use GetIDsOfNames to get the DISPID of the IsUserAllowed method. If we were interested in other methods or properties, we would ask for their DISPIDs all in the same call by sending in an array of names. This allows for efficient caching of the DISPIDs on the client. Then we prepare the arguments. This method takes in two arguments: A user name (BSTR) and a requested function (an int). As you can see, much of the work involved lies in preparing the arguments to send. We mentioned before that IDispatch access is not an ideal method for C++ clients because of the work involved. Before making the call, we declare the variable Result used to hold the return value (it was a VARIANT_BOOL, remember?).

Finally, we are at the point where we can make the method call. We call IDispatch::Invoke, passing it the appropriate parameters. After that, we check the return value. If it is TRUE, we report that the user is allowed access to the requested function. Otherwise, we inform the user that access was denied. Before uninitializing COM, we free the BSTR and the argument array that we previously allocated. That's all there is to it.

Build your executable and run it. You will get a message box saying, "User is allowed access."

Place a breakpoint on CoInitialize and step through the code while examining the value of each variable to gain more insight into how IDispatch::Invoke is called.

If you compare the code in the previous listing to our other test client, which uses the custom interface, you'll see how much more work IDispatch C++ clients need to perform. That's exactly why our AccessControl component supports dual interfaces!

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

A VB Automation Client

Now let's see how much effort it takes to create a VB Automation client. (If you do not have access to VB, you can skip this section.)

Start by creating a Standard Exe project. Save the form as frmMain.frm, and save the project as VBAutomationClient.vbp. Click the Button control in the toolbox and draw a button on the form. Double-click the button, and in the Button1_Click procedure, type the following code:

```
Private Sub Command1_Click()  
    ' declare our object as Object to use IDispatch  
    Dim objAccessControl As Object  
    Dim sUserName As String  
    Dim lReqFunction As Long  
    Dim bSuccess As Boolean  
  
    ' create the object  
    Set objAccessControl = 0  
    CreateObject("SecurityMgr.AccessControl")  
    ' Prepare the parameters  
    sUserName = "AnyUserName"  
    lReqFunction = 3  
    ' call the method passing the parameters  
    bSuccess = objAccessControl.IsUserAllowed( sUserName,0  
                                                lReqFunction )  
  
    'check the return value  
    If bSuccess Then  
        MsgBox "User is allowed access", vbInformation,0  
        "Access Check"  
    Else  
        MsgBox "User is denied access", vbCritical,0  
        "Access Check"
```

```
End If
' call pIDispatch->release
Set objAccessControl = Nothing
End Sub
```

Press F5 to run the project. When the form appears, click the button to invoke the Button1_Click() procedure. You should get a message box saying, "User is allowed access."

Clearly, IDispatch is easy and natural to use with VB. This makes sense, since that's what it was designed for.

In this section, we learned about Automation, the IDispatch interface, and dual interfaces. We've seen how you can easily tell the ATL Object Wizard to implement a dual interface. We also looked at some Automation data types and saw how we can write an Automation client in VC++ and in VB.

Summary

We've come a long way in this chapter. We started out by explaining how to define interfaces, CoClasses, and type libraries in IDL. We then created our first COM server using raw C++, which allowed us to see COM in action with no wizards or helpers.

Next, we looked at ATL and why it was invented. We saw how, by using VC++ templates and wizards, ATL helps automate much of the drudgery involved in building COM components while maintaining a small binary size and few dependencies. We used the ATL Object Wizard to build the SecurityMgr in-process COM server, which will be used in our COMTrader project. We then talked about out-of-process servers and we recreated SecurityMgr as an out-of-process server. We saw that the key differences between out-of-process and in-process servers are how the module is locked and how the components are created.

A discussion of threading and COM apartments came next. You learned about the different types of COM apartments and how the COM object's threading model affects how it communicates with the client. We saw how the ATL Object Wizard lets you easily pick your component's threading model.

Our last topic was about Automation and IDispatch. You learned what Automation is and why there is a need for it. We talked about IDispatch, dispinterfaces, and dual interfaces. We saw how to create Automation clients and how the ATL Object Wizard lets you choose whether you want to expose dual or custom Interfaces. We then compared writing Automation clients in VC++ and VB and saw that Automation is more suited for VB and VBScript.

Next, in Chapter 23, we'll look at some of the new COM features offered by Windows 2000 including asynchronous COM, pipes, synchronization mechanisms, lightweight handlers and call objects, and call cancellation.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 23

New COM Features in Windows 2000

- COM synchronizatio mechanisms
- Asynchronous COM
- COM pipes
- Call objects and call cancellation
- Lightweight handlers (LWHs)

Perhaps the most fundamental change to COM in Windows 2000 is the whole activation/contexts/synchronization architecture that resulted from integrating MTS into standard COM and adding numerous enhancements. However, Windows 2000 is packed with more new COM features. Although not as profound as the new architecture, some of these features offer interesting new programming models and allow for more powerful applications.

In this chapter, we'll discuss some of the new COM features in Windows 2000 and describe how to take advantage of them in your applications. We'll start by examining the new COM synchronization mechanisms. Then we will describe Asynchronous COM, COM pipes, call cancellation mechanisms, and lightweight handlers (LWHs).

Synchronization Mechanisms

In versions prior to Windows 2000, using Win32 synchronization primitives, such as critical sections and mutexes, is especially tricky with STA-based components. This is because a system call to a Win32 synchronization function such as `WaitForMultipleObjects` does not service the window message pump, which is needed to dispatch calls on the STA. Therefore, if you call a synchronization API that is waiting for a critical section, but that section has been entered by another object in the same apartment, there is a deadlock situation. The STA's thread is blocked waiting for the critical section, which will never be freed because it was entered on the same STA thread.

The COM Synchronization API

To provide an STA-safe synchronization mechanism, Windows 2000 has a new COM API, `CoWaitForMultipleHandles`. This API is defined in `ObjBase.h` as follows:

```
WINOLEAPI CoWaitForMultipleHandles( IN DWORD dwFlags,
                                     IN DWORD dwTimeout,
                                     IN ULONG cHandles,
                                     IN LPHANDLE pHandles,
                                     OUT LPDWORD lpdwindex );

/* Flags for Synchronization API and Classes */
typedef enum tagCOWAIT_FLAGS
{
    COWAIT_WAITALL = 1,
    COWAIT_ALERTABLE = 2
} COWAIT_FLAGS;
```

The `CoWaitForMultipleHandles` API parameters are similar to the Win32 `WaitForMultipleObjectsEx` API. The `dwFlags` can be any combination of `COWAIT_FLAGS`.

`CoWaitForMultipleHandles` detects the caller's threading model and behaves accordingly. If the caller resides in an STA, the API will enter the COM modal loop, thereby allowing calls to be dispatched on the STA. If the caller is in an MTA, the API simply calls `WaitForMultipleObjectsEx`.

The COM Synchronization Interfaces

Windows 2000 also introduces a few COM synchronization interfaces built around this new API. The new `ISynchronize` interface has three methods:

```
[
    object,
    uuid(00000030-0000-0000-C000-000000000046)
]
interface ISynchronize : IUnknown
{
    HRESULT Wait( [in] DWORD dwFlags,
                  [in] DWORD dwMilliseconds );
    HRESULT Signal();
    HRESULT Reset();
}
```

where:

- `Wait` is used to wait for the signaled state until the specified timeout value. `dwFlags` is a flag that can be used to indicate whether the client wants to wait for all outstanding calls to complete or for any of them to complete. The second parameter is a timeout period in milliseconds that the call to `Wait` will wait before returning. You can specify `INFINITE` to make it wait indefinitely.
- `Signal` sets the synchronization object to the signaled state. Any pending calls to `Wait` return `S_OK`.
- `Reset` resets the synchronization object state to the nonsignaled state.

Two system implementations of `ISynchronize` are provided by `CLSID_StdEvent` and `CLSID_ManualResetEvent`.

To wait for multiple synchronization objects, you use the `ISynchronizeContainer` interface, which is defined as follows:

```
[
    local,
    object,
    uuid(00000033-0000-0000-C000-000000000046)
]
interface ISynchronizeContainer : IUnknown
{
```

```

        HRESULT AddSynchronize( [in] ISynchronize *pSync );
        HRESULT WaitMultiple( [in] DWORD dwFlags,
                               [in] DWORD dwTimeOut,
                               [out] ISynchronize **ppSync );
    }

```

You call AddSynchronize to add the synchronization objects for which you want to wait, then you call WaitMultiple. If WaitMultiple returns S_OK, ppSync receives the synchronization object that was signaled. A system implementation of ISynchronizeContainer is provided by CLSID_SynchronizeContainer.

Another new interface, ISynchronizeHandle, lets you obtain the Win32 handle associated with a synchronization object:

```

[
    local,
    object,
    uuid(00000031-0000-0000-C000-000000000046)
]
interface ISynchronizeHandle : IUnknown
{
    HRESULT GetHandle( [out] HANDLE *ph );
}

```

Inheriting from ISynchronizeHandle, ISynchronizeEvent lets you assign a Win32 event handle to a synchronization object:

```

[
    local,
    object,
    uuid(00000032-0000-0000-C000-000000000046)
]
interface ISynchronizeEvent : ISynchronizeHandle
{
    HRESULT SetEventHandle( [in] HANDLE *ph );
}

```

These interfaces promise to make thread synchronization easier to implement and more COM-oriented than in earlier versions of Windows. In the next few sections, we will be using these new synchronization interfaces to wait for calls to complete.

Asynchronous COM

While the QC programming model provides an asynchronous calling mechanism for COM clients and servers, it is designed for specific environments and requirements. For example, both the client and server must be running MSMQ.

NOTE: The QC programming model is discussed in *Mastering COM and COM+* from Sybex.

By contrast, Windows 2000 introduces a new asynchronous calling mechanism called Asynchronous COM, which focuses on satisfying somewhat different requirements than QC fulfills. Perhaps the most important design objective of Asynchronous COM is to provide a mechanism whereby a client can continue operation while it is waiting for a method call to complete. This is possible to implement without Asynchronous COM, but it requires multithreaded programming (something most people are perfectly happy to do without once they are past the adolescent programming years).

Another objective of Asynchronous COM is to make the client and server independent in their execution models. This means that a server may be implemented asynchronously, but the client can call it synchronously. Conversely, a server may be implemented synchronously, yet allow the client to call it asynchronously. To meet this design objective, there must be a synchronous version and an asynchronous version of each COM interface that supports Asynchronous COM.

In the COM tradition of location transparency, it must be possible for the client to call the server's methods

asynchronously, whether the server is in the same apartment as the client or on a different machine. Otherwise, the server location would no longer be transparent, and that would violate one of the laws of COM.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Asynchronous Interface Construction

To demonstrate how Asynchronous COM satisfies its design objectives, we will build an asynchronous server to test making calls synchronously and asynchronously.

Building a basic asynchronous interface is relatively straightforward. You start by defining a traditional COM custom interface. Note that it cannot be a dual interface. It *must* be a custom interface. Interfaces derived from IDispatch cannot be asynchronous!

Let's create a new project called AsyncTest and add to it a simple ATL object called AsyncComp. Choose to create an EXE-based server and select Custom on the Attributes tab of the ATL wizard when adding the ATL object.

To make this new interface asynchronous, we need to generate a new GUID using the GUIDGEN.EXE utility. Next, open the AsyncTest.idl file and add the async_uuid attribute to the interface definition, as shown in the following listing:

```
[
    object,
    uuid(B7F43120-2F4F-11D3-8665-0008C7FE9130),
    async_uuid(D42327D2-2F56-11d3-8665-0008C7FE9130),
    helpstring("IAsyncComp Interface"),
    pointer_default(unique)
]
interface IAsyncComp : IUnknown
{
    [helpstring("method ProcessData")]
    HRESULT ProcessData( [in] long lInData,
                        [out] BSTR * pbstrOutData );
};
```

We added a method that has both in and out parameters. Place some implementation code behind this method. You may want to do something that takes time, which will serve to demonstrate the asynchronous behavior.

For example, you can add a message box to ProcessData. Be sure to return something in the out parameter.

Build your project and the proxy/stub DLL by running `nmake -f AsyncTestps.mk`. To successfully build this DLL, your `INCLUDE` environment variable must point to the platform SDK include directory before the VC++ include directory. Register the resulting proxy/stub DLL by running `regsvr32 AsyncTestps.dll`.

Examining the GUID definition file `AsyncComp_i.c` reveals that we now have an interface called `AsyncIAsyncComp`. The IDL compiler creates this for us because we added the `async_uuid` attribute. MIDL uses the synchronous interface name prefixed by `async`, and it uses the GUID we defined with `async_uuid`. If you look in `AsyncComp.h`, you will see the asynchronous interface definition shown here:

```
AsyncIAsyncComp : public IUnknown
{
    public:
        virtual /* [helpstring] */
            HRESULT STDMETHODCALLTYPE Begin_DoSomething(
                /* [in] */ long lData ) = 0;
        virtual /* [helpstring] */
            HRESULT STDMETHODCALLTYPE Finish_DoSomething(
                void ) = 0;
        virtual /* [helpstring] */
            HRESULT STDMETHODCALLTYPE Begin_ProcessData(
                /* [in] */ long lInData ) = 0;
        virtual /* [helpstring] */
            HRESULT STDMETHODCALLTYPE Finish_ProcessData(
                /* [out] */ BSTR __RPC_FAR *pbstrOutData ) = 0;
};
```

The asynchronous version of our interface breaks each method into two methods prefixed with `Begin` and `Finish`. The `Begin` method is used to initiate the call, and it receives all in and in, out parameters. The `Finish` method is used to complete the call and receive any output and takes all out and in, out parameters.

So which of these methods returns the original `HRESULT` from the server? The answer is neither. When you are designing an asynchronous interface, you can't use any application-specific `HRESULT` values as return values from the interface methods. Return values from both `Begin_` and `Finish_` indicate what happens when the call is initiated and completed respectively; they do not indicate the actual return value from the server.

NOTE: At the time of this writing, there is no special support for asynchronous interface definition in VC++ 6, Service Pack 3. You must manually replace the target Windows NT version in the proxy/stub make file `AsyncTestps.mk`. By default, this is set to `0x0400` (NT 4), which is not accepted by the MIDL compiler as a valid target for asynchronous interfaces. You must replace it with `0x0500` (`/D_WIN32_WINNT=0x0500`).

Asynchronous Interface Calls

Now let's build a client to work with our asynchronous server. Create a new console application called `AsyncClient`. This client is shown in the following listing from `AsynchClient.cpp`:

```
int main( int argc, char* argv[] )
{
    // Initialize COM
    HRESULT hr = CoInitialize(NULL);
    _ASSERT(SUCCEEDED(hr));
    // Create server instance
    // Obtain a pointer to the Synchronous interface
    IAsyncComp* pServer = NULL;
    hr = CoCreateInstance( CLSID_AsyncComp, NULL,
                          CLSCTX_SERVER,
                          IID_IAsyncComp,
                          reinterpret_cast<void**>( &pServer ) );
    _ASSERT( SUCCEEDED(hr) && pServer );
    BSTR bstr = NULL;
    bstr = SysAllocString(
```

```

        L"Test in-data, will be overwritten" );
long lData = 7;
printf( "Calling Synchronous Interface\n" );
    // Call the synchronous version of ProcessData
hr = pServer->ProcessData( lData, &pbstr );
printf( "Server returned: %S\n", bstr );
_ASSERT( SUCCEEDED(hr) );
SysFreeString( bstr );
    // Calling Asynchronous version
ICallFactory* pCallF = NULL;
AsyncIAsyncComp* pAsyncTest = NULL;
ISynchronize* pSync = NULL;
    // Get call factory interface
hr = pServer->QueryInterface( IID_ICallFactory,
                             (PVOID*) &pCallF );
_ASSERT( SUCCEEDED(hr) && pCallF );
    // Create a call object with the Async Interface
hr = pCallF->CreateCall( IID_AsyncIAsyncComp, NULL,
                        IID_AsyncIAsyncComp,
                        (LPUNKNOWN*) &pAsyncTest );
_ASSERT( SUCCEEDED(hr) && pAsyncTest );
    // Call the async version of ProcessData
printf( "Starting Async Call\n" );
hr = pAsyncTest->Begin_ProcessData( lData );
_ASSERT( SUCCEEDED(hr) );
SysFreeString( bstr );
    // To synchronize, we need an ISynchronize interface
hr = pAsyncTest->QueryInterface( IID_ISynchronize,
                                reinterpret_cast<PVOID*>( &pSync ) );
_ASSERT( SUCCEEDED( hr ) && pSync );
    // Wait as long as the status is RPC_CALLPENDING
while( ( hr = pSync->Wait( 0, 100 ) ) ==
        RPC_S_CALLPENDING )
{
    printf(".");
}
    // Release ISynchronize interface
pSync->Release();
bstr = SysAllocString(
    L"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" );
    // Make the Finish call to get out data
hr = pAsyncTest->Finish_ProcessData( &pbstr );
_ASSERT( SUCCEEDED(hr) );
printf( "\nCompleted Async Call\n" );
printf( "Server returned: %S\n", bstr );
SysFreeString( bstr );
    //Now try calling Finish immediately after Begin
bstr = SysAllocString(
    L"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" );
printf( "Begin Async Call\n" );
hr = pAsyncTest->Begin_ProcessData( lData );
_ASSERT( SUCCEEDED(hr) );
printf( "Finishing Async Call immediately\n" );
hr = pAsyncTest->Finish_ProcessData( &pbstr );
_ASSERT( SUCCEEDED(hr) );
printf( "Server returned: %S\n", bstr );
SysFreeString( bstr );
pAsyncTest->Release();
pCallF->Release();
pServer->Release();
CoUninitialize();

```



```
}    return 0;
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

This client is intended to demonstrate how to call the simple AsyncTest server that we created in the previous section. To do this, we make it call the regular (synchronous) interface IAsyncComp, as well as the asynchronous version AsyncIAsyncComp. Notice that because the MIDL compiler automatically adds the Async prefix, the interface name no longer begins with an *I*.

The client code starts by calling CoInitialize to initialize COM, followed by a call to CoCreateInstance to create a server instance and get an interface pointer to IAsyncComp. We then call ProcessData.

So far, there is nothing new in this code. These are the same routines we've been using throughout this book. The call to ProcessData is a blocking call, and it will not return until you click OK in the message box displayed by ProcessData. Next comes the good stuff.

We can't just QI the proxy for the asynchronous interface AsyncIAsyncComp because it is not implemented by the server. Instead, we must QI the proxy for ICallFactory, which is implemented by the proxy managers.

Then we use the returned interface pointer to create a call object by calling ICallFactory::CreateCall. This is the only method exposed by ICallFactory and is declared in ObjIDL.h as follows:

```
virtual HRESULT STDMETHODCALLTYPE CreateCall(
    /* [in] */ REFIID riid,
    /* [in] */ IUnknown __RPC_FAR *pCtrlUnk,
    /* [in] */ REFIID riid2,
    /* [iid_is][out] */
        IUnknown __RPC_FAR *__RPC_FAR *ppv ) = 0;
```

where:

- riid is the interface ID of the asynchronous interface for which you want the call object.
- pCtrlUnk is a pointer to the controlling unknown if the called object is to be aggregated. This should be NULL if you don't want the called object aggregated.
- riid2 is the interface you want to receive on the call object. Typically, you would use the asynchronous interface's IID or IID_IUnknown.

The call to CreateCall returns the interface pointer of the asynchronous interface. You then use this interface to call the asynchronous version of ProcessData, specifically Begin_ProcessData and Finish_ProcessData. As explained

earlier in this chapter, `Begin_ProcessData` takes the in parameter, and `Finish_ProcessData` takes the out parameter. If we had any in, out parameters, they would be passed to both methods.

Once it makes a call to `Begin_ProcessData`, the client may continue doing other work while the server processes the data. At some point, the client will want to call `Finish_ProcessData`, but how does the client know that the server has completed its data processing? To get this information, we can QI the call object for `ISynchronize`. By calling `ISynchronize::Wait`, the client waits until the asynchronous call is complete or the specified timeout expires. If the asynchronous call completes before the timeout, the return value is `S_OK`; otherwise, the return value is `RPC_S_CALLPENDING`.

Our test client loops while the call is still pending and displays a dot (.) every 100 milliseconds to show that it can be doing other work while it's waiting. Once `ISynchronize::Wait` returns `S_OK`, the client calls `Finish_ProcessData` to get the out parameter and display it. As you can see, the client is straightforward, but it must be programmed to use the asynchronous version of the interface.

Notes on the Asynchronous Server and Client

There are a few things to note from testing our asynchronous client and server. First, since we obtained `ICallFactory` from the proxy manager, this means the client and server must be in different apartments for this to work. If the client and server were in the same apartment (it would need to be an in-process server with the same threading model as the client), then we wouldn't have a proxy and we wouldn't be able to get an `ICallFactory` interface pointer. There is actually a way around this that we will get to soon.

Second, the asynchronous nature of the call is really just an illusion performed by the call object and proxy manager. The call object receives `Begin_ProcessData`, then turns around and calls the real (synchronous) `ProcessData` on the server (via the proxy), and then returns immediately to the client. (Note that `Begin_ProcessData` succeeds if the call to the server is initiated, so there is no guarantee that the call will reach the server or that it will complete successfully.) When the synchronous call completes and the client calls `Finish_ProcessData`, the call object returns to the client any out and in, out parameters it got from the synchronous call. The point is that *the server does all processing synchronously*. You can take it a step further and make the server do its processing asynchronously. We'll see how to do this next.

Asynchronous Server Processing

If you want the server to asynchronously process requests, you must do a little more work. There are three modes of operation when using asynchronous interfaces:

- The first mode is when the client gets a call object from the proxy manager and uses it to make asynchronous calls. The call object turns around and makes synchronous calls on the server. This mode is the one that we've already implemented in this chapter.
- The second mode is the exact opposite. The client makes a synchronous call to the proxy. The proxy then turns around and makes a call to the server. That call is processed asynchronously.
- In the third mode, the client makes an asynchronous call, and that call is processed asynchronously, just as the client wanted.

Figure 23.1 shows a simplified depiction of the three modes.

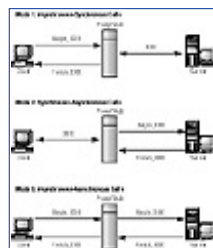


FIGURE 23.1 Modes of operation with asynchronous interfaces

NOTE: There is actually a fourth mode, in which the client calls the server synchronously and the server processes that call synchronously. This is the regular (synchronous) mode of operation.

Asynchronous Processing Runtime Events

To implement the second and third modes, the server must be able to process calls asynchronously. This

means it must have a custom call object that implements the asynchronous version of the interface and allows the server to process the calls asynchronously when and how it wants to. The server must also implement the ICallFactory interface and return the custom call object from ICallFactory::Create. The general sequence of events at runtime is as follows:

1. The client instantiates the server.
2. The client QIs the server for ICallFactory.
3. The client calls ICallFactory::Create.
4. The server returns an instance of the custom call object, which implements the asynchronous version of the interface (in our case, this interface is AsyncIAsyncComp).
5. The client calls Begin_XXX on the call object.
6. The call object manages to call the synchronous interface method XXX on the server *without blocking* and returns immediately to the client. This can be done by spawning a new thread for objects in an MTA or by setting a timer for objects in an STA. Begin_XXX takes only in and in, out parameters, so the call object must create its own version of the out parameters and pass them to the synchronous interface call.
7. The client calls Finish_XXX on the call object.
8. The call object passes on any out and in, out parameters that it got from the synchronous call (assuming it has completed).

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

This is a simplified sequence of events because it ignores an important fact: The client and server communicate via a remoting layer (call object/proxy/channel/stub and associated managers), which can execute calls a little differently from how they are requested. That is how we get the three modes discussed at the beginning of this section.

If you have been reading from the beginning of this chapter, you'll immediately notice a gap in the above steps. What happened to ISynchronize? Who implements this interface the call object or the server? The answer is that it depends. In our example, the server is an out-of-process server. Therefore, the client will always have a proxy manager that creates a call object, which implements ISynchronize. Because ISynchronize makes it possible to use synchronization semantics across machines, we do not need to implement our own ISynchronize. However, we do need to signal the synchronization object for which the client obtains an ISynchronize interface, telling it when we are done processing; otherwise, it will never know that we are finished!

Call Object Implementation

Let's walk through the steps of implementing a call object. Start by adding a new ATL object to your AsyncTest project. Call the object SimpleCallObject. Add the following method to the new ISimpleCallObjectInterface:

```
HRESULT SetServer([in] IAsyncComp * pServer);
```

This method is used to set a pointer to the server. You will see shortly how this pointer is used as a callback.

NOTE: We are creating a simple call object implementation, ignoring some of the real-world issues, such as concurrent access and call serialization. Going into these details would add more complexity and ultimately get in your way of understanding the fundamentals. We'll discuss call-serialization issues a bit later in the chapter.

Now add AsyncIAsyncComp to the inheritance list of CSimpleCallObject and add an interface map for it. The interface method's implementation is shown in the following listing from SimpleCallObject.cpp:

```
CSimpleCallObject * g_CallObject;  
void __stdcall TimerProc( HWND hwnd, UINT uMsg,
```

```

        UINT idEvent, DWORD dwTime )
{
    // Tell the CallObject to invoke process data
    g_CallObject->CallProcessData();
}
////////////////////////////////////
// CSimpleCallObject

STDMETHODIMP CSimpleCallObject::Signal(void)
{
    // Signal the sync object telling it we're done
    ISynchronize* pSync = NULL;
    // QI the outer unknown
    // for an ISynchronize iface on the sync object
    HRESULT hr =
        m_pOuterUnknown->QueryInterface( IID_ISynchronize,
            reinterpret_cast<void*>(&pSync) );
    _ASSERT(SUCCEEDED(hr));
    // Call signal
    hr = pSync->Signal();
    _ASSERT( SUCCEEDED( hr ) );
    pSync->Release();
    return S_OK;
}

STDMETHODIMP CSimpleCallObject::Begin_ProcessData(
    long lData )
{
    // Store [in] data for later use
    m_lData = lData;
    // Set the global variable that will be
    // used by the TimerProc
    g_CallObject = this;
    // Set timer
    m_TimerID = SetTimer( NULL, 0, 100, TimerProc );
    return S_OK;
}

STDMETHODIMP CSimpleCallObject::Finish_ProcessData(
    BSTR * pbstrOutData )
{
    //Pass on [out] params
    SysReAllocString( pbstrOutData, m_bstr );
    return S_OK;
}

STDMETHODIMP CSimpleCallObject::SetServer(
    IAsyncComp *pServer )
{
    // Set the server that will
    // be used later for ProcessData
    m_pServer = pServer;
    return S_OK;
}

void CSimpleCallObject::CallProcessData()
{
    //Stop the timer
    KillTimer( NULL, m_TimerID );
    //Call the synchronous interface
    //Store out param in member variable

```

```

    m_pServer->ProcessData( m_lData, &amp;m_bstr );
    //We're done
    g_CallObject->Signal();
}

```

Begin_ProcessData stores the in parameter into a member variable (m_lData). It then sets up a Win32 timer to fire off after 100 milliseconds and returns immediately.

Because our server lives in an STA, we cannot create another thread to call the real (synchronous) ProcessData. Instead, we create a timer and return immediately. When the timer procedure is called, it will call the synchronous process data. TimerProc is called when the timer's timeout value expires (we are setting it to 100 milliseconds). This function simply calls CSimpleCallObject::CallProcessData.

The first thing we do in CallProcessData is kill the timer to avoid repetitive firing. Then we make a blocking call to the synchronous server interface. Specifically, we call ProcessData. The parameters passed in to this call come from the call object itself. The value of m_lData will have been set by Begin_ProcessData. The value returned in the out parameter pbstrOutData is stored in the member variable m_bstr for use in Finish_ProcessData. Once this synchronous call completes, we call CSimpleCallObject::Signal.

CSimpleCallObject::Signal obtains an ISynchronize interface pointer using the aggregator's IUnknown pointer. It then calls ISynchronize::Signal(), telling it that the call has been completed. Note that we must do this whether ProcessData succeeds or fails, because the client may be using ISynchronize::Wait to wait indefinitely for the call to finish

When Finish_ProcessData is called, it copies the synchronous call's out parameters stored in member variables (in this case just a BSTR) to its own out parameters, thereby making them available to the caller.

That's all we need to do for the call object, but the server still needs a little work.

Asynchronous Server Modifications

As discussed earlier in this chapter, the server must implement ICallFactory. Add this interface to the server's CoClass definition in AsyncTest.idl:

```

[
    uuid(CE366350-2F4F-11D3-8665-0008C7FE9130),
    helpstring("AsyncComp Class")
]
coclass AsyncComp
{
    [default] interface IAsyncComp;
    interface ICallFactory;
};

```

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Also, add ICallFactory to the server's inheritance list and add an interface map for it. Finally, add the ICallFactory::CreateCall implementation to AsyncComp.cpp as shown in the following listing:

```
STDMETHODIMP CAsyncComp::CreateCall( REFIID riid,
                                     LPUNKNOWN pUnkCtrl,
                                     REFIID riid2,
                                     LPUNKNOWN *ppv )
{
    if( riid != IID_IAsyncIAsyncComp )
        return E_NOINTERFACE;
    else
    {
        ISimpleCallObject *pCallObject = NULL;
        HRESULT hr =
            CoCreateInstance( CLSID_SimpleCallObject,
                             pUnkCtrl,
                             CLSCTX_ALL,
                             riid2,
                             reinterpret_cast<void*>(ppv) );
        _ASSERT(SUCCEEDED(hr));
        hr = (*ppv)->QueryInterface( IID_ISimpleCallObject,
                                     reinterpret_cast<void*>(&pCallObject) );
        IAsyncComp * pServer = NULL;
        hr = this->QueryInterface( IID_IAsyncComp,
                                   reinterpret_cast<void*>(&pServer) );
        hr = pCallObject->SetServer( pServer );
        pCallObject->Release();
        return hr;
    }
}
```

This implementation doesn't do that much. It instantiates a call object and returns the requested interface

pointer. It also calls SetServer on the call object, passing it a pointer to the server's own interface. This pointer is what the call object uses when it calls ProcessData on the synchronous interface.

Now the server is ready to test. The best way to understand how all this works is to place breakpoints in each function and to run the server and the client in two separate IDEs. Stepping through the client's code reveals that the synchronous call made by the client is actually executed asynchronously (the second mode shown in Figure 23.1). Step through the rest of the client code and note how the server and call object are being called.

Call Serialization and Auto Complete

Now that you understand how to use Asynchronous COM, you must consider its effect on how you and others write code. If you create a server that has methods and provide an asynchronous version of the interface, you can't make any assumptions about the sequence of your method calls. For example, suppose a client was coded like this:

```
pServer->method1()  
pServer->method2()  
pServer->method3()  
do work  
.  
.  
.
```

That client may now be coded like this (or any other combination of these):

```
pServer->Begin_method1()  
pServer->Begin_method2()  
pServer->Begin_method3()  
do work  
.  
.  
.  
pServer->Finish_method1()  
pServer->Finish_method2()  
pServer->Finish_method3()
```

This can lead to unexpected results unless the server was designed and tested with the call sequence in mind. As a guideline when developing asynchronous servers, consider all possible call sequences and protect against undesired ones. Similarly, when calling an asynchronous server, try to complete the outstanding method call before beginning another call on the same server.

It is legal for a client to begin an asynchronous call on a call object, then release that object without calling Finish. This is called *auto complete*, and it allows a client to begin the call and not worry about synchronization (ISynchronize::Wait) or finishing that call.

COM Pipes

The Distributed Computing Environment (DCE) specification defines a pipe as a mechanism that allows transfer of an open-ended stream of data between a client and a server. The MIDL compiler provides a pipe-type constructor that can be used to define an RPC type. However, RPC pipes are awkward to use and have several limitations, including not being truly COM-oriented. Windows 2000 provides a new COM feature known as COM pipes.

COM Pipe Interfaces

There are three pipe interfaces defined in Windows 2000: IPipeByte, IPipeLong, and IPipeDouble, which provide pipes for bytes, longs, and doubles, respectively. (A future version of the MIDL compiler may provide a keyword to produce a pipe interface for any valid type.)

IPipeByte is declared in ObjIDL.h as follows:

```
IPipeByte : public IUnknown
```

```

{
    public:
        virtual HRESULT STDMETHODCALLTYPE Pull(
            /* [length_is][size_is][out] */ BYTE __RPC_FAR *buf,
            /* [in] */ ULONG cRequest,
            /* [out] */ ULONG __RPC_FAR *pcReturned ) = 0;

        virtual HRESULT STDMETHODCALLTYPE Push(
            /* [size_is][in] */ BYTE __RPC_FAR *buf,
            /* [in] */ ULONG cSent ) = 0;
};

```

Each of the pipe interfaces provides Pull and Push methods. As the name implies, Pull is used to pull or receive data from the pipe implementor. Push is used to push or send data to the pipe implementor.

To take advantage of COM pipes, you must write a component that implements one or more of the IPipeXXX interfaces. We will write a component that implements IPipeByte to send and receive files. Our implementation of IPipeByte::Pull will read from a text file and send data out. IPipeByte::Push will take the supplied data and store it in a new text file.

Start by creating an ATL project named PipeServer and make it an EXE-based server. Add a simple ATL object called PipeByteServer and choose Custom Interface on the wizard's Attributes tab. Since we really care only about the IPipeByte interface, we can remove the default interface called IPipeByteServer that was added by the wizard. The modified PipeServer IDL file is shown here:

```

import "oaidl.idl";
import "ocidl.idl";

[
    uuid(23A3A6A0-320C-11D3-866F-0008C7FE9130),
    version(1.0),
    helpstring( "PipeServer 1.0 Type Library" )
]
library PIPESERVERLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
        uuid(47F785C0-320C-11D3-866F-0008C7FE9130),
        helpstring("PipeByteServer Class")
    ]
    coclass PipeByteServer
    {
        [default] interface IPipeByte;
    };
};

```

Notice that we've made IPipeByte the default interface for our server. You must also modify PipeByteServer.h to remove the default interface IPipeByteServer added by the wizard and add IPipeByte to the inheritance list and to the interface map.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Next, you need to add IPipeByte methods to CPipeServer. While you're at it, you may also add two member variables, one of type ifstream and the other of type ofstream, for the files we will use (include fstream.h). Finally, implement IPipeByte::Push and IPipeByte::Pull in PipeByteServer.cpp as shown in the following listing:

```
STDMETHODIMP CPipeByteServer::Pull( BYTE *buf,
                                     ULONG cRequest,
                                     ULONG *pcReturned )
{
    *pcReturned = 0;
    try // Open file if not already open
    {
        if( ! m_infile.is_open() )
            m_infile.open( "PipeByteInData.txt" );
        // Read data from file
        if( ! m_infile.eof() )
        { // Simulate a lengthy operation
            Sleep(2000); // sleep 2 secs
            m_infile.read( buf, cRequest );
            // Return # of bytes read
            int nCount = m_infile.gcount();
            if( nCount > -1 )
                *pcReturned = nCount;
            else // Close file if bad count
                m_infile.close();
        }
        else //Close file if at EOF
            m_infile.close();
    }
    catch(...)
    { //Generic exception catcher
        return E_FAIL;
    }
}
```

```

        return S_OK;
    }

STDMETHODIMP CPipeByteServer::Push( BYTE *buf,
                                     ULONG cSent )
{
    try
    {
        // Open file if not already opened
        if( ! m_outfile.is_open() )
            m_outfile.open( "PipeByteOutData.txt" );
        // Write out send data
        if( cSent > 0 )
            m_outfile.write( buf, cSent );
        else // Close file if sent is 0
            m_outfile.close();
    }
    catch(...)
    {
        //Catch-all exception catcher
        return E_FAIL;
    }
    return S_OK;
}

```

Our implementation of IPipeByte is simple, with just enough substance to illustrate the point. Pull opens the text file for reading if it's not already opened (you can use the text file included with the source code or your own file). It then reads from it the specified number of bytes and returns the number of bytes that were actually read. We've added a call to Sleep() to simulate a long-running process. This will prove interesting shortly when we talk about asynchronous pipes. If the end of file is reached, we return 0 (zero) as the number of bytes returned and close the file. Returning a count of 0 when there is no more data is part of the COM pipe specification, and you must adhere to it.

Push is just as easy to follow. It opens the file for writing if it's not already open, then writes out the supplied data. If the count of supplied data is 0, there won't be any more data sent as part of this stream, so it closes the file. That's it for our IPipeByte implementation.

Writing a test client is just as straightforward. By now, you should be able to write a test client for this server. A sample test client is shown in the following listing (from PipeClient.cpp) which calls Pull until there is no more data (cReturned=0).

```

#define OPTIMUM_SIZE 10000
int main( int argc, char* argv[] )
{
    // Initialize COM
    HRESULT hr = CoInitialize(NULL);
    _ASSERT( SUCCEEDED( hr ) );
    // Create server instance
    // Obtain a pointer to the Synchronous interface
    IPipeByte* pPipeByte = NULL;
    hr = CoCreateInstance( CLSID_PipeByteServer, NULL,
                          CLSCTX_SERVER,
                          IID_IPipeByte,
                          reinterpret_cast<void**>( &pPipeByte ) );
    _ASSERT( SUCCEEDED( hr ) && pPipeByte );
    BYTE pBuffer[ OPTIMUM_SIZE ];
    ULONG cReturned = 0;
    printf( "Calling IPipeByte::Pull\n" );
    hr = pPipeByte->Pull( (BYTE*) pBuffer, 0
                        OPTIMUM_SIZE, &cReturned );
    printf( "Server returned: %d bytes\n", cReturned );
    // Call until no more data
    while( cReturned > 0 )

```

```

    {
        printf( "Calling IPipeByte::Pull\n" );
        hr = pPipeByte->Pull( (BYTE*) pBuffer,
                             OPTIMUM_SIZE, &pcReturned );
        _ASSERT( SUCCEEDED( hr ) );
        printf( "Server returned: %d bytes\n", cReturned );
    }
.
.
.

```

Again, the best way to understand what is going on is to step through the client and the server. You can do this by running each in a separate IDE and placing breakpoints in the server's IPipeByte implementation.

Asynchronous Pipes and Read-Ahead

Stepping through the client and server described in the previous section reveals an interesting fact. Although the client calls Pull synchronously, the server's Pull method is called asynchronously, and the client's call is blocked until there is data to return. This is equivalent to the second mode of asynchronous operation (described earlier in the chapter and illustrated in Figure 23.1).

All three IPipeXXX interfaces are defined as asynchronous interfaces, so there are also the asynchronous versions, called AsyncIPipeXXX. Furthermore, the IPipeXXX proxies (implemented in Ole32.dll) optimize data transfer through a read-ahead mechanism for Pull and a write-behind mechanism for Push.

If you step through the code carefully, you'll notice that the first Pull call that the client makes is blocked until the call to the server's Pull returns. As soon as the server returns from this call, the client is free to continue working. These are all just normal method calls. However, after the server's Pull method returns, it is immediately called again, but this time it is being called by the proxy, not the client itself. The proxy reads ahead one buffer and maintains this data in memory. The next time the client calls Pull, it gets the data from the proxy instead of from the server, which is usually much faster, especially if the client and server are on two separate machines. At this point, the proxy's buffer is empty because it gave the data to the client. The proxy makes another call to the server's Pull to fill up its data buffer. As you can see, except for the first time Pull is called, the proxy is always one buffer ahead of the client.

You also can call the asynchronous pipe interfaces directly. The rest of the sample client code (from PipeClient.cpp) is shown in the following listing:

```

.
.
.
    // Async calls
    ICallFactory* pCallF = NULL;
    AsyncIPipeByte* pAsyncPipe = NULL;
    ISynchronize* pSync = NULL;

    // Get call factory interface
    hr = pPipeByte->QueryInterface( IID_ICallFactory,
                                   (PVOID*) &ppCallF );
    _ASSERT( SUCCEEDED( hr ) && pCallF );

    // Create a call object with the Async Interface
    hr = pCallF->CreateCall( IID_AsyncIPipeByte, NULL,
                           IID_AsyncIPipeByte,
                           reinterpret_cast<IUnknown*>( &ppAsyncPipe ) );

    // To synchronize, we need an ISynchronize interface
    hr = pAsyncPipe->QueryInterface( IID_ISynchronize,
                                   reinterpret_cast<void*>( &ppSync ) );
    _ASSERT( SUCCEEDED( hr ) && pSync );
    do
    {

```



```

printf( "Calling Async Pull\nDoing other work " );
pAsyncPipe->Begin_Pull(OPTIMUM_SIZE);
    //Wait as long as the status is RPC_CALLPENDING
while( ( hr = pSync->Wait( 0, 100 ) ) ==
        RPC_S_CALLPENDING )
{
    printf(".");
}
pAsyncPipe->Finish_Pull( (BYTE*) pBuffer,
                        &cReturned );
printf( "\nAsync call completed. "
        "Server returned %d bytes \n", cReturned );
}
while( cReturned > 0 );
    // Release ISynchronize interface
pSync->Release();
pPipeByte->Release();
return 0;

```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

This example creates a call object and uses it to call AsyncPipeByte. The behavior is different in this case. Since we are calling the server asynchronously, the proxy does not attempt to read ahead or write behind. Each call to Begin_Pull results in a call on the server's Pull to get the pipe data. When the server finally returns, the client calls Finish_Pull, and it gets the data returned from the server. There is no read-ahead buffer in this case.

Call Objects and Call Cancellation

The concept of call objects was introduced in Windows NT 4 with the CoGetCallContext function. You can call this function to retrieve a call object interface such as IID_ISeurity. This concept is extended in Windows 2000 so that each method invocation has a COM object associated with it. By attaching a call object to each method call, the client can have more control over the method call. Call cancellation is a feature made possible by having a call object for each method call.

Call Cancellation Requests

Now that you are making asynchronous COM calls, you may need to abort or cancel a pending asynchronous call. For example, you may make an asynchronous call based on the end user's request. While the call is still outstanding, the user may ask your program to cancel. It would be beneficial if you could cancel the outstanding asynchronous call at this time. ICancelMethodCalls provides this facility. The interface is defined in ObjIDL.h as follows:

```
ICancelMethodCalls : public IUnknown
{
    public:
        virtual HRESULT STDMETHODCALLTYPE Cancel(
            /* [in] */ ULONG ulSeconds ) = 0;
        virtual HRESULT STDMETHODCALLTYPE TestCancel(void)
            = 0;
};
```

When a client makes a method call on an object that supports standard marshaling, the proxy calls CoSetCancelObject, passing it a call object to be associated with the specific method call. When the method call

returns, the proxy calls CoSetCancelObject again passing it NULL, which unregisters the call object for the returned method call. Because STA threads can be reentered, an STA-based client has a stack of call cancellation objects, one for each outstanding method call. MTA clients have only one call cancellation object at most.

A client obtains a pointer to ICeancelMethodCalls by calling CoGetCancelObject, defined as follows:

```
HRESULT CoGetCancelObject(  
    DWORD dwThreadID,    // Thread with pending call  
    REFIID riid,         // ID of requested interface  
    void **ppUnk );      // Pointer to requested interface
```

Since the client may have several threads with outstanding COM calls, dwThreadID identifies the thread for which the call cancel object is desired. If the outstanding call was made on the same thread calling CoGetCancelObject, dwThreadID may be set to 0.

Call Cancellation Handling

Calling ICeancelMethodCalls::Cancel sends a low-level DCE cancel message to the server. The ulSeconds parameter allows the client to specify how long to wait for the server to process the cancel request. The server must periodically check whether the call has been canceled by calling ICeancelMethodCalls::TestCancel.

NOTE: In the Platform SDK documentation accompanying Windows 2000 RC1, ICeancelMethodCalls is defined as having a third method, called SetCancelTimeout, to set the timeout period. However, in ObjIDL.h, the interface has only two methods, with Cancel accepting the timeout value parameter as described here.

If the server's call to ICeancelMethodCalls::TestCancel returns RPC_E_CALL_CANCELED, it may handle this cancellation request in any way that it deems appropriate. For example, the server may immediately stop execution, or it may decide to finish up some work first.

Two new APIs are available for enabling and disabling call cancellation on a thread basis: CoEnableCallCancellation and CoDisableCallCancellation. In addition, the new API CoCancelCall provides a wrapper around CoGetCancelObject, followed by ICeancelMethodCalls::Cancel. CoTestCancel provides a wrapper around CoGetCancelObject, followed by ICeancelMethodCalls::TestCancel.

As mentioned in the previous section, if your component uses standard marshaling, you get call cancellation support for free (although you still need to check for it using TestCancel). If your component uses custom marshaling, you must write your own call cancel object that implements ICeancelMethodCalls. In addition, if your server component uses asynchronous server-side execution, the custom call object returned from ICallFactory::CreateCall must implement ICeancelMethodCalls to support call cancellation.

Lightweight Handlers

In some cases, the default marshaling behavior provided by the standard marshaler is just not good enough. For example, you may want a smart proxy that can handle certain method calls without going back to the server and incurring the network round trip. Prior to Windows 2000, you needed to implement a custom marshaler to do this. Now you can implement a *lightweight handler (LWH)*.

Basically, the client gets a proxy manager, which aggregates the interface proxy. Similarly, the stub manager aggregates individual interface stubs on the server side. An LWH is an object inserted into the client space to perform certain special actions upon calling methods on the server.

[Previous](#) [Table of Contents](#) [Next](#)



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[Previous](#)
[Table of Contents](#)
[Next](#)

Standard LWH

The simpler kind of LWH, known as standard LWH, requires that the component support IStdMarshalInfo:

```
[
    local,
    object,
    uuid(00000018-0000-0000-C000-000000000046)
]
interface IStdMarshalInfo : IUnknown
{
    typedef [unique] IStdMarshalInfo * LPSTDMARSHALINFO;
    HRESULT GetClassForHandler
    (
        [in] DWORD dwDestContext,
        [in, unique] void *pvDestContext,
        [out] CLSID *pClsid
    );
};
```

The single interface method GetClassForHandler is called by CoMarshalInterface when the interface is being marshaled. Instead of transmitting the CLSID of the default proxy to the client's address space, COM transmits the CLSID of the LWH, which it obtains from the call to IStdMarshalInfo::GetClassForHandler. Figure 23.2 shows the proxy architecture when using a standard LWH.



FIGURE 23.2 The standard LWH object model

At the client-side, COM executes a CoUnmarshalInterface call, which creates an Identity object used to create

Although this may seem like a complicated affair, keep in mind that to support the standard LWH, you need only two things: the object must implement `IStdMarshalInfo`, and the LWH must call `CoGetStdMarshalEx`. This is definitely easier than writing your own custom marshaler.

If your LWH needs additional initialization information beyond that supplied in the standard marshaling packet, you must implement a more complicated version of the LWH known as a custom LWH. In this case, the COM object must aggregate the standard marshaler, implement `IStdMarshalInfo` and `IMarshal`, and delegate its `IMarshal` method calls to the aggregated standard marshaler *before* executing its own `IMarshal` code. The additional work here comes from implementing `IMarshal`.

NOTE: For more information about LWHs and custom marshaling, see Yasser Shohoud’s Web site at www.DevXpert.com. Also see “COM Handlers” in the Platform SDK documentation.

While the previous three chapters have offered an overview of COM as well as details on constructing COM applications, the topic as a whole is far more extensive than it is practical to cover here. For more information on COM and COM+ applications (including more extensive examples) you can refer to Yasser Shohoud's *Mastering COM and COM+* from Sybex.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

CHAPTER 24

Active Directory Operations

- The Microsoft Active Directory
- The Lightweight Directory Access Protocol (LDAP) and the Third Party API
- The Active Directory Service Interface (ADSI)
- Active Directory services, properties, and demos

The *Active Directory* is a Microsoft directory service feature that is included with Microsoft Windows 2000 Server. The *Microsoft Message Queue (MSMQ)* uses Active Directory to store, organize, and make available information about public queues, certificates, computers, and so on.

Microsoft Certificate Server's programmability makes it possible to issue certificates to any directory compliant with *Lightweight Directory Access Protocol (LDAP)*. LDAP is an easily implemented subset of the X.500 DAP standard for directory services. LDAP has strong industry support and is expected to become an industry standard for directory services. Certificate Server compliance with LDAP implies that it can interoperate with policy tools and other third-party applications that support LDAP directory services.

The *Active Directory Service Interfaces (ADSI)* are COM-based programming interfaces used for accessing diverse directory services on a variety of platforms. ADSI has a dual interface supporting both *virtual function table (VTBL)* access and Automation.

Active Directory

The Active Directory is an organized structure supported by Windows 2000 that is designed to provide access to data—in the form of files or directories—independent of the actual physical structure, both locally and across network drives.

For example, today's corporate networked systems commonly extend across multiple servers, which may number dozens or even hundreds, and can easily encompass servers that are physically distributed across multiple sites, multiple states, or even internationally. As a result, particularly in large corporations where

individuals in widely dispersed locations are working collaboratively, needed files may be physically located on diverse servers. Even more important, locating those files can be a complex process.

Because of this complexity, individuals often resort to distributing copies by email or keeping local copies for work purposes. As a result, people find themselves in the position of needing to reconcile separate and different versions of documents and other work files.

Due to the complexity of storage sites—contemporary enterprise networks sharing information not only across LAN and WAN systems but also across virtual networks and even through Internet access—new methods are needed to manage both access and access security.

Of course, an enterprise-wide network can separate data and access by using internal servers for corporate users, employing restricted systems for sensitive data, providing an intranet with separate areas for restricted and open data, and using a firewall system for public Internet access. However, with such a mix of sites and systems, access presents its own complexities. For example, finding an individual item may mean sorting through a host of servers while maintaining appropriate security for replicated data, which is virtually impossible.

Naturally, there are directory solutions to provide (and restrict) access to distributed, shared data but mechanisms of this type have their own problems, especially in ensuring that the solutions are adaptable and accommodate changes and additions to the network.

To accommodate such needs, the Microsoft Windows 2000 Server network operating system offers a robust directory structure with a single network logon as well as a single point of administration and replication. Simple network administration, however, is only half of the solution to today's requirements, and complex enterprise networks demand such features as distributed security, extensibility, hierarchical directory views, multi-master replication, and scalability.

The Lightweight Directory Access Protocol (LDAP)

In order for Active Directory policy information to be made available to other network devices—devices such as routers, WAN interfaces, or switches that might be interested in regulating resource admission based on the Active Directory's configurable-policy information—a programmatic interface to policy information stores in the Windows 2000 Active Directory was necessary. The interface that exposes Windows 2000 Active Directory policy information is called the Third Party API, and it is accessible through the *Lightweight Directory Access Protocol (LDAP)*.

The LDAP is a directory service protocol that runs directly over the TCP/IP stack. The information model (both for data and namespace) of LDAP is similar to that of the X.500 OSI directory service, but with fewer features and lower resource requirements than X.500. Unlike most other Internet protocols, LDAP has an associated API that simplifies writing Internet directory service applications. The LDAP API is targeted at management applications and browser applications that do not have directory service support as their primary function.

The Third Party API is an LDAP interface for accessing policy-based admission information stored in the Windows 2000 Active Directory. Its definition requires a quick discussion of the process of gaining such information.

The Admission Control Service (ACS) is the QOS component that provides a policy-based means of regulating and managing network resources associated with 802 (shared media) LANs (the ACS service encompasses SBM functionality within its base service). Network resources are regulated through permission and admission policies; the ACS interfaces indirectly with LPMs in order to gain approval or rejection of a node's requested network resources. LPMs can be considered the go-betweens for ACS admission policy queries and the user's policy store (presumably some sort of authentication/policy store—perhaps a centralized database of enterprise-wide authentication information).

In the default LPM that ships with Windows 2000, the LPM is implemented through *Msidpe.dll*, which provides a policy interface to the Windows 2000 centralized database of enterprise-wide information, the Active Directory. Thus, in the ACS component model, policy information is provided through *Msidpe.dll*, or as it is referred to throughout this documentation, the LPM component. ACS policy decisions regarding access to network device resources are realized through the LPM component's access to Windows 2000 Active Directory policy stores.

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Active Directory Services

To answer the needs of contemporary network systems, Microsoft's Active Directory is a service provider with the functionality needed to locate both network services and information. The Active Directory architecture provides scalability to equally support both small businesses and large corporations. This is accomplished by using an integrated set of APIs and protocols to extend services across multiple namespaces and to gather and display directory and resource information from a multitude of both local and distributed systems.

Features and benefits found in the Active Directory include:

- Open standards supporting cross-platform directory services, including the Domain Name System (DNS) and the LDAP
- APIs supporting both script (VB) and C/C++ applications
- Support for standard name formats, ensuring simplicity of migration and ease of use
- Drag-and-drop administration
- A simple hierarchical domain structure
- Interoperable compatibility with Novell Directory Services (NDS)
- Backward compatibility with previous Windows NT versions
- Use of extensible schema to provide directory object extensions
- Global catalogs supporting fast lookups
- Multi-master replication supporting convenient updates

The purpose of this chapter, however, is not to enumerate the features of the Active Directory services, nor is it our intention to discuss how to use Active Directory or how to set up your server systems. For this purpose, a variety of books are available that concentrate on configuring, managing, and maintaining AD systems, including *Mastering Active Directory* by Robert R. King (published by Sybex).

Instead, having offered this very brief overview of Active Directory, in this chapter we will look at some of the programming elements provided to support Active Directory.

A Caveat

Because this book is written prior to the actual release of Windows 2000, the programs and notes in this book are based on beta versions and tested against Windows 2000 Release Candidates versions 1 and 2. At this time, however, both the Active Directory and ADSI are somewhat less than fully complete and some elements continue to exhibit erratic behaviors.

Therefore, it is strongly suggested that experimentation is in order before placing complete reliance on any of the aspects of Active Directory or the features supplied by ADSI.

Preparing to Use ADSI

Before you attempt to use any of the program examples discussed in this chapter, a couple of initial provisions are necessary.

To begin, you need to install the ADSI SDK (filename SDK.ZIP), which is available by download from Microsoft's Web site at www.Microsoft.com/adsi. At the same time, you should also install the Windows 2000 SDK (also available by subscription for download from Microsoft's Web site, at www.msdn.microsoft.com/developer, and also distributed on CD as part of subscription sets).

After download, unzip the contents (and structure) to a suitable location, such as `x:\Developer Tools\ADSI_SDK` or `x:\Program Files\Microsoft Visual Studio\ADSI_SDK`.

NOTE:

The ADSI SDK includes a series of demo programs that are discussed and used as examples for Active Directory access in this chapter. For this reason, it is strongly recommended that you install the ADSI SDK.

Next, after installing the two SDKs, a pair of changes must be made to your Visual C++ environment.

1. From the Visual C++ menu, select Tools ▸ Options to display the Options dialog box shown in Figure 24.1.

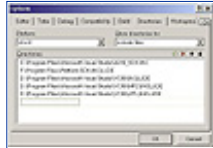


FIGURE 24.1 The Visual C++ Options dialog box

2. Select the Directories tab.
3. Add a new entry to the Directories list by:
 - Clicking on the blank at the bottom of the list
 - Typing the name of the drive and directory where the ADSI SDK include directory is found
4. Click the Up arrow at the right of the Directories list to move the `..\ADSI_SDK\INC` entry to the top of the list.
5. Repeat Step 3 to add the `..\Platform SDK\Include` directory. (This entry does not need to be moved to the top of the list.)
6. Select the Library files entry from the "Show directories for" list, as shown in Figure 24.2.

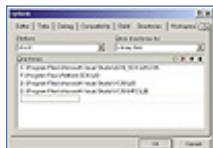


FIGURE 24.2 Adding Library Directories

7. Repeat Step 3 for the `..\ADSI_SDK\LIB\i386` and `..\Platform SDK\LIB` directories.

These changes are global—that is, these directory entries will affect all Visual C++ projects, both past and future. For individual projects using ADSI, however, one more step is required after you create each project.

Individual Project Settings

After creating a new ADSI project (using AppWizard), you need to add two libraries to the project's Link settings.

1. To perform this task, select Project Ø Settings from the menu. The Settings dialog box is shown in Figure 24.3.

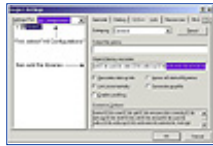


FIGURE 24.3 The Project Settings dialog box

2. Select the Link tab.
3. From the “Settings For” list, select All Configurations.
4. Add both the `activevds.lib` and `adsiid.lib` to the Object/library modules list.

And that's it—the project is now ready to use ADSI functions.

Active Directory Properties

ADSI is, as you might guess, a COM-based service object. All elements in a directory service—including users, computers, services, how user accounts and computers are organized, and the hierarchical directory trees of all file systems on the network—are represented by Active Directory objects. Further, Active Directory objects are accessed through interfaces like other COMobjects (see Chapter 21, “COM/COM+ Overview,” Chapter 22, “Building and Using COM Servers in VC++,” and Chapter 23, “New COMFeatures in Windows 2000”).

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

ADSI Objects

Each Active Directory object represents an element in the hierarchical structure forming the directory trees or directory services that are supplied through the network.

The hierarchical structure begins at the top with an ADSI Namespaces Object that comprises the ultimate parent container. Beginning with the ADSI Namespaces Object, each child object can be a parent or a host object, supporting object creation and management for subsequent child objects.

Each Active Directory object is identified by:

Relative Name The *leaf name* of the object. The leaf name distinguishes each object from sibling directory entries.

Schema Class Name Identifies the type of object, such as “user”, “computer”, or “domain”. Each Active Directory object has a single primary interface, accessed through QueryInterface to identify the schema class of the object as IID_IADsUser, IID_IADsComputer, or IID_IADsDomain. The ADSI Schema Class Object corresponding to the object’s Schema Class Name will report the currently defined well-known relationships for the object; that is, which objects may contain objects of this type, whether the object is a container, which CLSID provides an implementation, and which properties and interfaces are supported.

ADsPath Uniquely identifies an Active Directory object regardless of the directory service implementation. ADsPath strings are COM display names governed by the Active Directory naming convention. The ADsGetObject function is used to bind to an ADsPath name, or if the ADsPath is not known, IADsNamespaces::get_DefaultContainer can be employed. Two formats for ADsPath names—the URL format (preferred) and the OLE format (or Bang! format)—are supported by all providers.

URL format Takes the form “ADs:”, “WinNT:”, or “Your_ProgID://Your_Provider_Specific_String”.

OLE format Takes the form “ADs!”, “WinNT!” or “Your_ProgID!Your_Provider_Specific_String”.

TIP:

Programmatic identifiers (ProgIDs) are case-sensitive and must exactly match the value registered in the registry.

ADSI objects are implemented only when providing an Active Directory implementation for a directory

service. Since each entry in a directory tree in a network is represented by an Active Directory object, a very “lightweight” representation will greatly enhance performance.

All ADSI Objects support three COM interfaces:

IUnknown Provides the universal COM object management and interface query methods

IDispatch Provides automation methods for late-bound access to an object’s methods and properties

IADs Provides standard Active Directory object management methods and persistent property management

While the IUnknown and IDispatch interfaces have been discussed in previous chapters, the IADs interface is specific to the ADSI objects.

The IADs Interface

The IADs interface provides a series of methods for getting—and setting—information about the ADSI objects that form the building blocks of a directory service implementation. The basic functionality of the IADs interface includes:

- Identifiers for both the name and type of the object
- Binding information that uniquely identifies the object instance in a directory tree
- Methods to retrieve an interface pointer on the parent container object, which controls the creation and deletion of every Active Directory object
- Methods to retrieve the schema definition of the object—the well-defined relationship of this object to others in the directory service
- Methods to set and retrieve individual property values by name
- A simple caching system for the properties of an object

NOTE:

All ADSI objects can contain other objects with container objects supporting the additional IADsContainer interface.

The objective behind using Active Directory objects to represent all aspects of a directory service is to provide flexibility and uniformity allowing both network administrators and Active Directory providers a simple, consistent view of the various underlying directory services.

Implementing IADs

IADs is implemented only when providing an Active Directory implementation for your directory services.

Also, in order to add further (custom) functionality to the existing Active Directory objects, a custom COM interface must be defined and implemented to support those services—that is, any existing interface cannot be modified, and new functionality can be added only through a new interface.

Active Directory Interface Methods (in VTable Order)

All Active Directory objects support three interfaces: IUnknown, IDispatch, and IADs. The COM IUnknown interface supports three standard methods:

QueryInterface Returns pointers to supported interfaces

AddRef Increments the reference count

Release Decrements the reference count

The COM IDispatch interface supports four standard methods:

GetTypeInfoCount Returns the number of type descriptions

GetTypeInfo Returns a description of an object’s programmable interface

GetIDsOfNames Maps the name of a method or property to the DISPID

Invoke Invokes one of the object’s methods or gets or sets one of the object’s properties

NOTE:

Specific Active Directory objects may possess additional interfaces, but you will need to refer to the documentation for each object type for details.

Previous	Table of Contents	Next

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

The Active Directory IADs interface offers a much more extensive range of methods including:

get_Name Returns an object's relative name.

get_Class Returns an object's schema class name.

get_GUID Returns an object's globally unique identifier. For example, to identify the ADSI user objects for every implementation, begin by retrieving a single user object by enumerating the schema object for the class name User. From here, retrieve the GUID for the Active Directory object type User (using **get_GUID** or, in Visual Basic, **.GUID**), then set the enumeration filter for that GUID and enumerate through all of the provider's ADSI Namespace Objects to find the User objects for each implementation.

get_AdsPath Returns an object's **AdsPath** string, which uniquely identifies one object from all other objects. For example, given an interface point to an instance of an object—as through enumeration—the **get_AdsPath** method (or, in Visual Basic, **.Name**) retrieves the object's unique **AdsPath** name.

get_Parent Returns the **AdsPath** string for the object's parent container. For example, the **get_Parent** method (or, in Visual Basic, **.Parent**) is used to expand a directory tree upwards by determining whether an object is contained by a parent level in the directory hierarchy.

get_Schema Returns the **AdsPath** string to the schema class definition object for the current object. For example, while browsing through the directory service to discover which methods and properties are supported, the **get_Schema** method (or, in Visual Basic, **.Schema**) returns the **AdsPath** naming the object representing the schema class defining the original object.

GetInfo Reloads the property values for an object from the underlying directory service.

SetInfo Saves the changes to the object and its dependents.

Get Retrieves the value for a named property. For example, the **Get** method is called using the property names retrieved from the schema object (see **get_Schema**) to retrieve the values of all properties currently defined for the object.

Put Sets the value for a named property.

GetEx Retrieves the value(s) for a named single- or multi-valued property.

PutEx Sets the value(s) for a named single- or multi-valued property.

GetInfoEx Reloads specific property values for an object from the underlying directory service.

All ADSI client applications/tools use these methods to communicate with directory service objects, and a number of these methods are illustrated in the *ADsPropertyList* demo program discussed in the following section.

TIP:

Refer to the documentation for specific service providers to determine which ADSI errors are returned by each. Also note that these can include errors that indicate that a request may not have been completely processed.

The *ADsPropertyList* Demo

NOTE:

The *ADsPropertyList* demo program is distributed by Microsoft as part of the ADSI SDK and is found in the `..\ADSI_SDK\Samples\ActiveDir\PropertyList\vc` subdirectory.

The *ADsPropertyList* demo is written as a DOS (shell) application and begins by declaring a series of variables before initializing COM and then binding to a server using the LDAP://rootDSE specification.

```
void main()
{
    // COM result variable
    HRESULT      hr;
    // interface pointers
    IDirectorySearch *pDSSearch = NULL;
    IADs             *pIADs      = NULL;
    IADsPropertyList *pIADpl     = NULL;
    VARIANT          vRet;
    LONG             lCount;
    IADs             *pIADsrootDSE = NULL;
    BSTR             bsNamingContext;
    VARIANT          vResult;
    WCHAR            wszActualLDAPPath[1024];

    VariantInit( &vRet );
    CoInitialize(0); // Initialize COM
    // bind through rootDSE
    _putws( L"Binding to a server using LDAP://rootDSE" );
    hr = ADsGetObject( L"LDAP://rootDSE", IID_IADs,
                      (void **)&pIADsrootDSE );
    if( FAILED(hr) ) return;
```

The `ADsGetObject` method returns an `IADs` identifier and an indirect pointer to the `riid` interface identifier as well as the `HRESULT` value reporting success or failure.

You've seen the `SUCCEEDED` macro in many of the previous demos. The `FAILED` macro is simply the flip side of the same coin, returning `TRUE` on failure rather than `TRUE` on success. However, assuming that a successful result has been returned, after a little further preparation, the pointer to the Interface identifier—`pIADsrootDSE`—is used to call the `Get` method to retrieve the value, in this instance, for the `defaultNamingContext`.

```
    // Get the defaultNamingContext attribute and for
    // binding to the actual DC
    bsNamingContext =
        SysAllocString( L"defaultNamingContext" );
    VariantInit( &vResult );
    // Get the defaultNamingContext Attribute
    hr = pIADsrootDSE->Get( bsNamingContext, &vResult );
```

Once the naming context has been retrieved, the next step is to build a binding string from the naming context and then bind to the path (in other words, the actual server) using another call to the `ADsGetObject` method.

```

if( SUCCEEDED(hr) )
{
    // Make sure it's a BSTR
    hr = VariantChangeType( &pvResult, &pvResult, NULL,
                           VT_BSTR );
    if( SUCCEEDED(hr) )
    {
        // Build a binding string
        wsprintf( wszActualLDAPPath, L"LDAP://%s",
                  vResult.bstrVal );
        wsprintf( L"\nBinding to the path %s\n",
                  wszActualLDAPPath);
        // Bind to the actual server
        hr = ADsGetObject( wszActualLDAPPath, IID_IADs,
                           (void **)&pIADs );
    }
    VariantClear( &pvResult );
}
SysFreeString( bsNamingContext );
// Release the RootDse binding...
pIADsrootDSE->Release();
pIADsrootDSE = NULL;
if( FAILED(hr) ) return;
wprintf( L"\n\nSuccessfully bound to %s\n",
          wszActualLDAPPath );
PrintIADsObject( pIADs );

```

At this point, *ADsPropertyList* has identified the Active Directory server object and bound to the server and then called the *PrintIADsObject* subroutine to report as follows:

Binding to a server using LDAP://rootDSE

Binding to the path LDAP://DC=Western-isles,DC=local

Successfully bound to LDAP://DC=Western-isles,DC=local

```

NAME:      DC=Western-isles
CLASS:     domainDNS
GUID:      8444aaaca8976f428296650592a6b159
ADSPATH:   LDAP://DC=Western-isles,DC=local
PARENT:    LDAP://DC=local
SCHEMA:    LDAP://schema/domainDNS

```

TIP:

The exact results reported by the *PrintIADsObject* subroutine and other examples in the following sections will, of course, vary according to your system configuration, names and other settings.

So far, this is a pretty good start—we have the domain name and class, a globally unique ID, and paths for the domain, parent, and schema. However, there's still a lot more information available, and the *ADsPropertyList* demo proceeds by calling the *IADsPropertyList* interface with the *GetInfo* method to load all properties.

```

if( FAILED(hr) ) return;
wprintf( L"\n\nEnumerating this object's properties
        Using the IADsPropertyList interface\n\n" );
hr = pIADs->GetInfo();
if( FAILED(hr) ) return;

```

Where the *GetInfo* method retrieves all properties from the interface, an alternative is to call the *GetInfoEx* method to load only specific properties.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

In this case, however, the next step is to call the QueryInterface method for the interface to the property list and then call the get_PropertyCount method (using the property list interface) to find out how many properties there are to report.

```
//QueryInterface() for IADsPropertyList ptr.  
hr = piADs->QueryInterface( IID_IADsPropertyList,  
                           (void **)&piADpl );  
if( FAILED(hr) )return;  
hr = piADpl->get_PropertyCount( &plCount );  
printf( "\n The Object has %d properties\n", lCount );
```

In this case, the report appears as:

Enumerating this object's properties using the IADsPropertyList interface

The Object has 39 properties

Now that we know the number of properties, we can move to the first property on the list before initiating a loop to report all of the properties.

```
// Move to the FIRST property in the list  
hr = piADpl->Next( &pvRet );  
CheckHRESULT( hr, "piADpl->Next(&pvRet);" );  
for( long lElement = 0; lElement < lCount; lElement++ )  
{
```

Within the loop, there's more than a simple query taking place. While the process within the loop does begin with a query—specifically, a call to the QueryInterface method—because the query can report a number of different property types, there is additional code to extract and report details depending on the results of the initial query.

Therefore, the first step is to call the QueryInterface method to retrieve a pointer (pAdsEntry) to the next property entry in the object.


```

IDispatch * pDisp = V_DISPATCH(&pvRet);
IADsPropertyEntry * pAdsEntry = NULL;
hr = pDisp->QueryInterface( IID_IADsPropertyEntry,
                           (void**) &ppAdsEntry );

```

Upon success, all that we have thus far is a pointer to the property, and the next step is to use the pointer to the property interface (pAdsEntry) to call the get_Name method to retrieve (and report) the name of the object property.

```

if( SUCCEEDED(hr) )
{
    BSTR    bsName;
    VARIANT vValue;

    VariantInit( &pvValue );
    hr = pAdsEntry->get_Name( &pbsName );
    if( SUCCEEDED(hr) )
    {
        wprintf( L"\n NAME:%s \t\n",
                 (LPOLESTR)bsName );
        SysFreeString( bsName );
    }
}

```

Again, after testing the results from retrieving the name, the process continues using the get_Values method to determine what variant value styles (or types) are reported by each property entry.

```

hr = pAdsEntry->get_Values( &pvValue );
puts( "The Values Variant style is:" );
puts( GetVariantStyle( vValue.vt ) );
if( SUCCEEDED(hr) )
{

```

Since the actual value styles are reported as bit flags in a VARIANT value, the GetVariantStyle function—found in the ADSIhelpers.cpp source file—simply tests all possible flags to return a character string listing the variant styles.

The next step, since we now have a VARIANT ARRAY of IDispatch* pointers, is to read the safe array and enumerate each IDispatch* pointer. As each pointer is enumerated, the object is queried for the IADsPropertyValue interface that, in turn, provides the property information.

```

if( HAS_BIT_STYLE( vValue.vt, VT_ARRAY ) )
{

```

The HAS_BIT_STYLE test is simply to confirm that vValue is actually a safe array before proceeding.

```

    LONG dwSLBound = 0;
    LONG dwSUBound = 0;
    LONG i;
    hr = SafeArrayGetLBound( V_ARRAY(&pvValue),
                             1, (long FAR *)&dwSLBound );
    hr = SafeArrayGetUBound( V_ARRAY(&pvValue),
                             1, (long FAR *)&dwSUBound );

```

Before processing the safe array, the SafeArrayGetLBound and SafeArrayGetUBound functions retrieve the upper and lower bounds for the array so that another loop can proceed within these limits.

```

for( i = dwSLBound; i <= dwSUBound; i++ )
{
    VARIANT v;
    VariantInit(&v);
    hr = SafeArrayGetElement(

```

```
V_ARRAY(&pvValue), (long FAR *)&i,
&pv );
```

The SafeArrayGetElement function retrieves a single indexed element from a safe array. After an element is retrieved, the next step is another HAS_BIT_STYLE test, but this time, we need to ensure that the recovered element is an IDispatch pointer.

```
if( SUCCEEDED(hr) )
{
    if( HAS_BIT_STYLE( v.vt,
                        VT_DISPATCH ) )
    {
```

After ensuring that the type is correct, the retrieved element is cast as a pointer to IDispatch so that the object's interface can be queried.

```
IDispatch *pDispEntry =
    V_DISPATCH( &pv );
IADsPropertyValue *pAdsPV = NULL;

hr = pDispEntry->QueryInterface(
    IID_IADsPropertyValue,
    (void **) &pAdsPV );
if( SUCCEEDED(hr) )
{
```

Finally, after querying the interface, the GetIADsPropertyValueAsBSTR function retrieves the actual value in a BSTR format, which is then converted to an OLESTR format for reporting.

```
BSTR bValue;
// Get the value as a BSTR
hr = GetIADsPropertyValueAsBSTR(
    &bValue, pAdsEntry,
    pAdsPV );
if( SUCCEEDED(hr) )
{
    wprintf( L"<%s>\n",
              (LPOLESTR)bValue );
    SysFreeString( bValue );
}
pAdsPV->Release();
pAdsPV = NULL;
}
}
else
    puts( "!!NO DISPATCH ENTRY!!" );
VariantClear(&pv);
}
}
```

The remainder of the process is principally cleanup, clearing the variant array, releasing the pointer to the interface, and, of course, stepping to the next object property before the loop continues.

```
VariantClear(&pvValue);
}
else
    wprintf( L" NAME:%s \t",
              (LPOLESTR)V_BSTR( &pvValue ) );
}
pAdsEntry->Release();
pAdsEntry = NULL;
```

```
    }  
    hr = pIADpl->Next(&pvRet);  
}
```

Finally, when the loop is finished, the COM is uninitialized.

```
    CoUninitialize();  
}
```

The demo program, as provided by Microsoft, really needs to be run from a DOS shell since, if it is run from the compiler using the toolbar option, the window will simply close without allowing you to read the reported results.

However, if you add a simple `getch()` statement following the `CoUninitialize` instruction at the end of the main function (and add an include statement for `conio.h`), the window will remain open, waiting for a keystroke. This allows you to examine the reported results and, since the results are lengthy, to scroll back through the list.

Alternately, you can use the Build menu option to prevent the DOS window from closing automatically.

What you should notice from the code example—and the reason that we have gone into this code at some depth—is that we began by binding to the server Active Directory object (the root level). We then drilled down through successive layers, always using Active Directory interfaces, until we reached the lowest level properties for the Active Directory object.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Since the entire output reported by the ADsPropertyList program can run to some four pages of text, the following is only a sample of a few of the 39 properties reported.

The first value report is masteredBy, which has four variant styles and reports at some length on the server settings.

NAME: masteredBy
The Values Variant style is:
VT_R4 VT_BSTR VT_VARIANT VT_ARRAY
<CN=NTDS Settings,CN=DELOS,CN=Servers,
CN=Default-First-Site,CN=Sites,CN=Configuration,
DC=Western-isles,DC=local>

TIP:

Remember: Individual results will vary.

Some of this information, such as the domain name, we already knew from the initial reports, but other elements are first being reported here. And some of the information here is also reported by other queries, as in the next two reports:

NAME: distinguishedName
The Values Variant style is:
VT_R4 VT_BSTR VT_VARIANT VT_ARRAY
<DC=Western-isles,DC=local>

NAME: objectCategory
The Values Variant style is:
VT_R4 VT_BSTR VT_VARIANT VT_ARRAY
<CN=Domain-DNS,CN=Schema,CN=Configuration,
DC=Western-isles,DC=local>

Also, in a few cases (see the following sample) we get some rather cryptic reports:

NAME: wellKnownObjects
The Values Variant style is:

```

VT_R4 VT_BSTR VT_VARIANT VT_ARRAY
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>
<<UNRECOGNIZED>>

```

The exact meanings for each of these properties—including the cryptically <<UNRECOGNIZED>> reports—are discussed in the ADSI help files or in the ADSI object provider’s documentation.

The *ADSIAddUser* Demo

While querying interface properties is useful, being able to write information to the Active Directory is more relevant—pragmatically speaking, anyway.

The *ADSIAddUser* demo program (included on the CD in the Chapter 30 folder) is a relatively simple application that creates a new user—in this case, “Sam Hall”—and then adds a description for the user.

The *ADSIAddUser* demo begins by declaring a series of variables, including four pointers to interfaces and two BSTR values for the user name and the user description.

```

#include "stdafx.h"
#include <activeds.h>
#include <stdio.h>
#include <conio.h>

int main( int argc, char* argv[] )
{
    HRESULT          hr;
    IADsContainer     *pCont;
    IDispatch         *pDisp = NULL;
    IADs              *pUser;
    IADsUser          *pUserProp;
    BSTR              bsName, bsDesc;

```

As a preliminary step, we begin by creating BSTR objects containing the values that we need to create a user entry.

```

    bsName = SysAllocString( L"Sam Hall" ); // add this user
    bsDesc = SysAllocString( L"program generated entry" );

```

One of these values—bsName—will be passed through an IADs interface, while the second—bsDesc—will be transferred through an IADsUser interface. In a more realistic application, we would probably be assigning a full selection of data entries, as user properties. However, for the present, the description is enough.

Once the BSTR variables have been initialized, the “meat” of the application begins, as most applications of this type must, by calling `CoInitialize` to initialize COM before calling `ADsGetObject` to retrieve an `IADsContainer` interface. Also, since this is written as a demo program, we test the results and report success or failure at each step.

```

    CoInitialize( NULL ); // initialize COM
    hr = ADsGetObject( L"WinNT://Western-isles",
                      IID_IADsContainer,
                      (void**)&pCont );
    if( FAILED(hr) )
    {
        _putws(
            L"\nCould not retrieve IADsContainer object" );
        _putws( L"\nPress any key to exit" );
        getch();
        return 0;
    }

```

```
_putws( L"\nRetrieved IADsContainer object" );
```

Now that we have a pointer to the IADsContainer interface, the next step is to create the user entry and, in doing so, to initialize a pointer to an IDispatch interface.

```
hr = pCont->Create( L"user", bsName, &ppDisp );
pCont->Release();
```

Once the IDispatch interface has been generated, the IADsContainer interface—pCont—is no longer needed and can be released. And, of course, we have another error check and report:

```
if( FAILED(hr) )
{
    _putws( L"\nCould not create user" );
    _putws( L"\nPress any key to exit" );
    getch();
    return 0;
}
wprintf( L"\nCreated user: [%s]", (LPOLESTR)bsName );
SysFreeString( bsName );
```

Just as we released pCont once it was no longer needed, we also call SysFreeString to release the BSTR object once we're done with it. However, even though we've released pCont, we still have the pUser pointer to the IDispatch interface and we use this interface pointer to call the QueryInterface function to get a new pointer to an IADs interface.

```
hr = pDisp->QueryInterface( IID_IADs, (void*)&ppUser );
if( FAILED(hr) )
{
    _putws( L"\nCould not retrieve pUser interface" );
    _putws( L"\nPress any key to exit" );
    getch();
    return 0;
}
_putws( L"\nRetrieved pUser interface" );
```

And, again, we have the usual error checks and reports.

NOTE:

Like the *ADsPropertyList* demo, we're running down a chain of interfaces, using each link to reach the next level down. Unlike the *ADsPropertyList* demo, however, the *ADSIAddUser* process should be a little bit easier to follow.

In the previous step, we released the parent interface immediately after retrieving a pointer to the next interface in the chain. In contrast, this time, we need to retain the IDispatch interface because we have another use for it.

Continuing with the application, however, we use the IADs (pUser) interface to commit the new user to memory (that is, to the permanent record on the server) and then since this is the only use we'll make of the IADs interface—the pUser interface is released.

```
pUser->SetInfo();          // commit to memory
pUser->Release();          // release the object
_putws( L"\nUser committed to memory" );
```

When we called the pCont->Create function to create a new user entry in the Active Directory, we were operating in memory and, without getting a pointer to the user object (IDispatch) and calling the SetInfo function, our changes and entry would simply have been lost. Now, however, this entry has been added to the system.

Once this is done, since we still have the pointer to the IDispatch interface—pDisp—we're ready to call a second interface—of type IADsUser—which we can use to set user properties.

```
hr = pDisp->QueryInterface( IID_IADsUser,  
                             (void**)&pUserProp );  
if( FAILED(hr) )  
{  
    _putws( L"\nCould not retrieve pUser interface" );  
    _putws( L"\nPress any key to exit" );  
    getch();  
    return 0;  
}  
_putws( L"\nRetrieved pUserProp interface" );  
pDisp->Release();
```

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

After the usual error check and report, since we are now finished with the IDispatch interface, we can release pDisp before proceeding.

Then, using the IADsUser interface, we call the put_Description function to set the description for this user and to free the description string.

```
pUserProp->put_Description( bsDesc );  
SysFreeString( bsDesc );  
pUserProp->SetInfo();  
pUserProp->Release();  
_putws( L"\nUser info committed to memory" );
```

Remember, until we call the SetInfo method, our operation has occurred only in memory and nothing has been saved. But, after calling SetInfo, the information is saved and we can release the IADsUser interface.

As suggested previously, in a more realistic application, such as one where we were interested in populating a system, we might want to write a series of values at this point. To do this, we would make multiple put_xxxx calls (and then free the associated BSTR values) before calling SetInfo to save the changes.

NOTE:

For a complete list of both the put_xxxx and get_xxxx methods, refer to the online documentation for the IADsUser interface.

Finally, to conclude, we call CoUninitialize to uninitialize COM, and then—since we're expecting to execute this under Debug in a DOS window—we offer a prompt and a call to getch to leave the window open for examination.

```
CoUninitialize(); // uninitialize COM  
_putws( L"\nPress any key to exit" );  
getch();  
return 0;  
}
```

Although the location is optional, if you are executing this demo from a workstation, you can switch to the server and open the Active Directory Users and Computers window (see Figure 24.4) and you should find the entry for Sam Hall in the list with the description supplied. (The Type field, User, was specified when pCont->Create was called.)

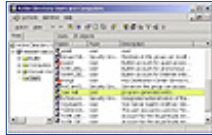


FIGURE 24.4 A new Active Directory user has been entered.

IADsUser Methods

To offer some idea of the properties that can be set (or retrieved) just for a user account under Active Directory, the IADsUser methods (most providing access to properties) are listed here:

- get_BadLoginAddress** The address of the last node considered an “Intruder”
- get_BadLoginCount** The number of bad logon attempts since the last reset
- get_LastLogin** The date and time of the last network login
- get_LastLogoff** The date and time of the last network logoff
- get_LastFailedLogin** The date and time of the last failed network login
- get_PasswordLastChanged** The date and time of the last password change
- get/put_Description** A description of the user account
- get/put_Division** Gets and sets the division within a company (organization)
- get/put_Department** Gets and sets the organizational unit within the specified organization
- get/put_EmployeeID** The end user’s employee identification
- get/put_FullName** The end user’s full name
- get/put_FirstName** The end user’s first name
- get/put_LastName** The end user’s last name
- get/put_OtherName** The end user’s additional name (nickname, middle name)
- get/put_NamePrefix** The end user’s name prefix (such as Mr., Ms., Dr., and so on)
- get/put_NameSuffix** The end user’s name suffix (such as Jr., III, and so on)
- get/put_Title** The end user’s title within the organization
- get/put_Manager** The end user’s manager
- get/put_TelephoneHome** The end user’s list of home phone numbers
- get/put_TelephoneMobile** The end user’s list of mobile phone numbers
- get/put_TelephoneNumber** The end user’s list of work-related phone numbers
- get/put_TelephonePager** The end user’s list of pager phone numbers
- get/put_FaxNumber** The end user’s list of fax phone numbers
- get/put_OfficeLocations** An array of end user locations
- get/put_PostalAddresses** An array of end user post office addresses
- get/put_PostalCodes** An array of zip codes for the Postal Addresses
- get/put_SeeAlso** An array of AdsPaths of other objects related to this user
- get/put_AccountDisabled** Whether or not this user’s account is disabled
- get/put_AccountExpirationDate** The date and time after which this user cannot log in
- get/put_GraceLoginsAllowed** The number of times this user can log on after his/her password has expired
- get/put_GraceLoginsRemaining** The number of “grace” logins left after this user’s password expires before locking the account
- get/put_IsAccountLocked** Whether or not this user’s account is locked
- get/put_LoginHours** The time periods during each day of the week indicating valid login periods
- get/put_LoginWorkstations** Workstations and their net addresses for this end-user
- get/put_MaxLogins** The maximum number of simultaneous logins allowed for this user
- get/put_MaxStorage** The maximum amount of disk space allowed for this user

get/put_PasswordExpirationDate The date and time when this user's password will expire
get/put_PasswordMinimumLength The minimum number of characters allowed in a password
get/put_PasswordRequired Whether or not a password is required for this user
get/put_RequireUniquePassword Whether or not a new password must be different than one in the password history list
get/put_EmailAddress The end user's email address
get/put_HomeDirectory The end user's home directory
get/put_Languages An array of language names for the end user
get/put_Profile The end user's profile path
get/put_LoginScript The end user's logon script path
get/put_Picture An array of bytes with the end user's image
get/put_HomePage The end user's URL defining a home page
Groups Determines the groups to which this end-user belongs
SetPassword Sets the password for this user
ChangePassword Changes the password from a specified old value to a new value

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

GO!

[Search Tips](#)

[Advanced Search](#)

PUBLICATION LOOKUP

GO!

Title

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

GO!

[Previous](#)

[Table of Contents](#)

[Next](#)

The User Properties dialog box is shown in Figure 24.5, where the properties accessed through the methods listed require 12 tabs to display. And, rather obviously, if you are generating a large number of entries for an Active Directory system—such as setting up a new server—you’re hardly likely to be interested in making these entries by hand.



FIGURE 24.5 The User Properties dialog box

Further, some of these (refer back to the `get/put_Picture` method for an example) could be more than tedious to handle manually, but could be created from a database. Or, of course, the `get_xxxxx` methods could be used to export information to a database.

Also, you should remember that other interfaces—and many Active Directory objects support multiple interfaces—will offer different sets of methods and contain different properties. Always refer to the object supplier’s documentation for full details on supported interfaces, methods, and properties.

The AddGroup Demo

The `ADSIAddUser` demo discussed previously showed you how to add a user (and potentially other information) as an Active Directory object. One of the other elements that you are likely to want is to create groups. Toward this objective, the `AddGroup` demo program, found in the `...\\ADSI_SDK\\Samples\\ActiveDir` directory, provides an excellent example. One caveat, however: While the example is fine as is, some of the following comments and suggestions may help when executing the example.

By way of introduction, the `AddGroup` example expects to be called with a minimum of three command-line parameters beginning with:

Path Of Container This is the `ADsPath` of the container where the new group will be placed (see the following section, “`AdsPath` Specifications”).

NT5 Group Name The name of the group to be created.

Downlevel NT4 SAM Account Name This name cannot exceed 20 characters and must be globally unique.

Two optional arguments may also be included:

Group Type May be either 'global' (default), 'local,' or 'universal'.

Security May be either 'Security' (default) or 'NoSecurity'. If no parameter is supplied or if security is specified, the new group will be created with ADS_GROUP_TYPE_SECURITY_ENABLED.

ADsPath Specifications

While a lot of the demo programs will accept a domain argument in the form "WinNT://Domain_Name," the *AddGroup* operation requires an LDAP argument in either of two forms:

- "LDAP://**DOMAIN_NAME**"
- "LDAP://**SERVER_NAME/DC=DOMAIN_NAME,DC=local**"

The first example is simpler, requiring only the domain name, while the second example provides a fully qualified ADsPath name with either specification serving essentially the same purpose.

If you know the correct path, you can enter your arguments in the VC++ compiler by selecting the Project Ø Settings Ø Debug tab and then typing your argument list in the Program arguments edit box. (See Figure 24.6.)

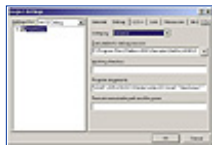


FIGURE 24.6 Adding command line arguments

Here, we've specified our command line arguments using the fully qualified format:

"LDAP://DELOS/DC=Western-isles,DC=local"NewGroup"NewGroup." However, if you are unsure of the appropriate domain specification, you can run the Name_Test.vbs script to return a fully qualified ADsPath name.

The Name_Test script (supplied by Max Vaughn at Microsoft) is included in the Chapter 30 subdirectory on the CD, or can be written directly:

```
On Error Resume Next
dim comobj
set comobj = GetObject("LDAP://NAME_OF_SERVER")
if ( err.number<> 0 ) then
    WScript.Echo "Error Number: " & err.number & vbCrLf & Err.Description
    WScript.Quit 1
Else
    wscript.echo comobj.adspath
End If
```

After modifying the script to include the name of your server, the reported ADsPath should appear something like the one shown in Figure 24.7.

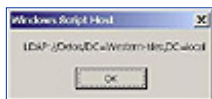


FIGURE 24.7 A reported ADsPath specification

WARNING:

A Cautionary Note: Dumb errors are absolutely the worst kind and they do happen. Therefore, do not separate your command line parameter list using commas. The commas will be interpreted as part of the parameters and you will encounter strange and inexplicable errors as a result.

AddGroup Demo Provisions

The *AddGroup* demo offers several elements of particular interest. The first is found in the *ParseCommandLine* subroutine.

TIP:

Command line arguments should always be processed immediately on entry to a program. Any necessary values should be immediately stored as local variables since the *argv* values themselves can easily be overwritten by other operations.

The *ParseCommandLine* function is called from *main* as the first operation in the process and stores each argument passed as a global variable or by setting a flag value.

```
BOOL ParseCommandLine( int argc, wchar_t *argv[] )
{
    if( argc < 4 || argc > 6 )
    {
        _putws( pwszUsage );
        return FALSE;
    }
}
```

The first test simply determines whether a minimum of four arguments was received. Remember *argv[0]* is automatically the name of the executable program, and for our purposes, it is simply ignored.

```
    if( argc >= 4 )
    {
        bsADsPathContainer = SysAllocString(argv[1]);
        bsNT5GroupName     = SysAllocString(argv[2]);
        bsSamAcctName      = SysAllocString(argv[3]);
    }
}
```

The arguments *argv[1]* through *argv[3]* are the three required arguments and provide the container *ADsPath* name, the group name for the new group, and the SAM account name.

The *SysAllocString* function is used to create a copy of the *argv* value that is assigned to a global *BSTR* variable. The *BSTR* data type is used because this is the type that further operations will require. However, this also presents a disadvantage to the programmer since *BSTR* variables are not readily monitored during debug operations. More explicitly, we cannot conveniently view the contents of a *BSTR* variable in the Watch or Local variables windows since these are displayed only as addresses, not as the actual string contents.

The next two arguments are optional and are used to set flag values, beginning with the group type:

```
    if( argc >= 5 )    // was the group type passed in?
    {
        if( _wcsicmp( L"global", argv[4] ) == 0 )
            iGroupType = ADS_GROUP_TYPE_GLOBAL_GROUP;
        else
            if( _wcsicmp( L"local", argv[4] ) == 0 )
                iGroupType = ADS_GROUP_TYPE_DOMAIN_LOCAL_GROUP;
            else
                if( _wcsicmp( L"universal", argv[4] ) == 0 )
                    iGroupType = ADS_GROUP_TYPE_UNIVERSAL_GROUP;
                else
                {
                    _putws( L"!! Incorrect Group Type Parameter !!" );
                    _putws( pwszUsage );
                    return FALSE;
                }
    }
    else    // Default to Global
        iGroupType = ADS_GROUP_TYPE_GLOBAL_GROUP;
```

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Here, depending on the argument string, the `iGroupType` variable is set to one of three possible values, defaults to global, or if the argument is not recognizable, a brief error report is issued.

In a similar fashion, the fifth parameter is used to decide whether group security is enabled (the default) or not—or, of course, if an incorrect parameter was supplied.

```
// was security enabled passed in?
if( argc == 6 )
{
    if( _wcsicmp( L"security", argv[5] ) == 0 )
        iGroupType |= ADS_GROUP_TYPE_SECURITY_ENABLED;
    // else if the parameter there is NOT nosecurity- error
    else
        if( ! ( _wcsicmp( L"nosecurity", argv[4] ) == 0 ) )
        {
            _putws( L"!! Incorrect security Parameter !!" );
            _putws( pwszUsage );
            return FALSE;
        }
}
else // Default to security
    iGroupType |= ADS_GROUP_TYPE_SECURITY_ENABLED;
return TRUE;
}
```

TIP:

Notice in both operations that the `_wcsicmp` function is used to perform a case-insensitive string comparison.

Once the command line parameters have been accepted, the main function continues by initializing COM and declaring and initializing several local variables:

```

CoInitialize(0);
HRESULT hr;
IDirectoryObject * pDirObjectContainer = NULL;
IDirectoryObject * pDirObjRet = NULL;

```

The `ADsPath` argument, now contained in the `bsADsPathContainer` variable, is used as an argument to call `ADsGetObject` with a flag value specifying a directory object. On return, the `pDirObjectContainer` variable will hold the container where the new directory will be created.

```

// Bind to the container passed
hr = ADsGetObject( bsADsPathContainer,
                  IID_IDirectoryObject,
                  (void **)&pDirObjectContainer );

```

If there is an error, the program branches to error reporting. But, assuming that everything is okay thus far, the `CreateGroup` helper function is invoked with the object container, the group and SAM account names, a pointer to a directory object (for the returned directory), and a group type flag.

```

if( SUCCEEDED(hr) )
{
    // Call the helper function to create the group
    hr = CreateGroup( pDirObjectContainer,
                     bsNT5GroupName,
                     bsSamAcctName, &pDirObjRet,
                     iGroupType );
}

```

The remainder of the main function is occupied with reporting results and, hopefully, enumerating the properties of the new group object. For the current topic, however, we will jump directly to the `CreateGroup` function.

```

HRESULT CreateGroup( IDirectoryObject * pDirObject,
                    LPWSTR             pwCommonName,
                    LPWSTR             pwSamAcctName,
                    IDirectoryObject ** ppDirObjRet,
                    int                 iGroupType )
{
    assert(pDirObject);
}

```

The first operation in `CreateGroup` is validating the SAM account name, or at least validating the length of the name, which cannot exceed 20 characters.

```

if( wcslen( pwSamAcctName ) > 20 )
{
    MessageBox( NULL,
                L"SamAccountName CANNOT be bigger than 20\n"
                L"characters",
                L"Error: CreateSimpleGroup()", MB_ICONSTOP );
    assert(0);
    return E_FAIL;
}

```

Since the supplied argument is a `BSTR` variable—a wide-char string—the `wcslen` function is used for the length.

Following this minimal validation, several local variables are declared, including an `ADS_ATTR_INFO` structure, and a number of the variables are initialized.

```

HRESULT      hr;
ADSVALUE     samValue;
ADSVALUE     classValue;
ADSVALUE     groupType;
LPDISPATCH  pDisp;

```

```

WCHAR      pwCommonNameFull[1024];
ADS_ATTR_INFO attrInfo[] =
{
    { L"objectClass", ADS_ATTR_UPDATE,
      ADSTYPE_CASE_IGNORE_STRING, &classValue, 1 },
    { L"sAMAccountName", ADS_ATTR_UPDATE,
      ADSTYPE_CASE_IGNORE_STRING, &sAMValue, 1 },
    { L"groupType", ADS_ATTR_UPDATE,
      ADSTYPE_CASE_IGNORE_STRING, &groupType, 1 } };
DWORD dwAttrs = sizeof(attrInfo) /
    sizeof(ADS_ATTR_INFO);
classValue.dwType = ADSTYPE_CASE_IGNORE_STRING;
classValue.CaseIgnoreString = L"group";

sAMValue.dwType = ADSTYPE_CASE_IGNORE_STRING;
sAMValue.CaseIgnoreString = pwSamAcctName;

groupType.dwType = ADSTYPE_INTEGER;
groupType.Integer = iGroupType;

wsprintfW( pwCommonNameFull, L"CN=%s", pwCommonName );

```

Once all of the variables are initialized, including turning the common name (passed as an argument) into the full common name format (by prepending the "CN=" string), we're finally ready to call the `CreateDSObject` function.

```

hr = pDirObject->CreateDSObject( pwCommonNameFull,
                                attrInfo,
                                dwAttrs, &ppDisp );

```

The `CreateDSObject` function finishes the task of creating the Group object, returning a pointer to the object in the `pDisp` variable. Of course, the `CreateDSObject` function isn't limited to creating a group but can create any object type that belongs to a container with the type of object and particulars specified in the `attrInfo` and `dwAttrs` arguments.

Finally, the `pDisp` pointer returned by `CreateDSObject` is used to retrieve another interface pointer—`ppDirObjRet`—which is the result actually returned to the calling operation.

```

if( SUCCEEDED(hr) )
{
    hr = pDisp->QueryInterface( IID_IDirectoryObject,
                                (void**) ppDirObjRet );

    pDisp->Release();
    pDisp = NULL;
}
return hr;
}

```

After the `CreateGroup` function returns, the main function executes a query using the returned pointer to the object interface and produces the result shown in Figure 24.8.

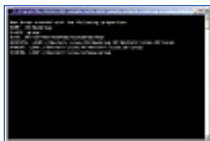


FIGURE 24.8 Successfully creating a new group

The remainder of the *AddGroup* demo is relatively straightforward, but does contain elements of interest and is worth reading through—particularly the error reporting functions, which are commended to your attention.

[HOME](#)[SUBSCRIBE](#)[SEARCH](#)[FAQ](#)[SITEMAP](#)[CONTACT US](#)[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), [Copyright © 1996-2000 EarthWeb Inc.](#) All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

Other ADSI Demo Programs

We've walked through a couple of demo programs to show how Active Directory interfaces are used and how some of the interface methods are employed. However, as mentioned earlier, the ADSI SDK includes a host of example programs demonstrating a wide diversity of applications and operations.

Before listing the demo programs offered in ADSI DSK, your first step should be to go to the ...\\ADSI_SDK\\Samples\\Provider\\Setup directory to install the Active Directory samples using the install.bat program. Running install.bat will:

- Install the sample provider
- Set up the sample DS in the registry

To view the sample provider:

1. Run ADSVW.EXE
2. Type **Sample**: as the AdsPath. Now you can navigate the name space.

The remaining samples are found in the subdirectories shown under ...\\ADSI_SDK\\Samples\\...

ActiveDir Offers a variety of Active Directory program examples:

AddGroup Adds an Active Directory group

Attributes Retrieves and lists schema attributes including Non-Replicated, Indexed, Constructed, and Global Catalog attributes

BindToParent Locates a user in the current domain, displays the parent container's AdsPath, and binds to the parent container

Computer (Visual Basic) Creates a computer object and assigns permissions

CreateUser Creates a user in Active Directory

Credentials Logs on to a current domain using an alternate user name and password

GCallUsers Lists all users by searching the global catalog

GetDomainMode Reports whether the current domain is running in native or mixed mode

GetSchemaInfo Queries the schema for specified classes

GUIDBind Binds to a directory object by ADsPath, retrieves the GUID, then rebinds by GUID

IsMemberEx Checks to determine whether a member belongs to a group

OtherWKO Performs multiple container operations

PropertyList Enumerates object properties under Active Directory

QueryUsers Queries and reports all users within the current user's domain

RootDSE Reports the root DSE and schema

SID Finds a user in the current domain and displays that user's objectSid property in string form

User Finds a user in the current domain and displays that user's IADsUser properties

WKGUIK Finds a user's container in the current domain

ASP Active Server Page examples:

First Enumerates a computer object

WAB Performs a search and displays a report

DataLink SQL database operations examples

DistQuery ADSI OLE DB query example

Exchange Exchange Server examples using Visual Basic:

Configuration Displays and sets the value of the maximum message size on an Exchange Server's Message Transfer Agent

Custom Recipient Creates a custom recipient on an Exchange Server

Distribution List Creates a distribution list on an Exchange Server

Enum Enumerates persons, custom recipients, and distribution lists in a recipient container

HomeMDB Finds and reports the home mailbox database (MDB) server

MailBox Creates and deletes Exchange 5.5 Mailboxes

Org Reports default naming context

RBDL Populates and updates Microsoft Exchange "rules-based distribution lists" (RBDL)

Search Uses an alias to perform a search on an Active Directory provider

Security Reads and modifies an Exchange Security Descriptor Attribute in Exchange 5.5/ADSI 2.5

Tombstone Demonstrates access to hidden or deleted objects

Extension ADSI Extension tutorials (including documents and a Power Point slideshow):

Tutorial Contains two tutorial examples using Visual C++ and Visual Basic

General Offers a variety of general-purpose examples:

ADOQuery (Visual Basic) Performs an ADO query

ADQI Lists all possible ADSI interfaces for a given AdsPath and allows some methods to be run for each interface supported

ADsCmd Command line example allows you to browse and display objects belonging to any Directory Service

ADsError Demonstrates Active Directory errors (see the sidebar following this list of examples)

ADSIDump Retrieves values from an ADSI path, writing results to an output file

DsSrch Performs search operations on the Active Directory tree

Internet (Visual Basic) Performs name/email name search either across Internet sources or through the Active Directory

Interopt Demonstrates interoperability between:

ADSI and NDS using Visual Basic examples

ADSI and Netscape using Visual Basic examples

ADSI and Netware using Visual Basic examples

Perl Contains Perl examples to bind, create, and modify Active Directory objects

SiteServer Contains these examples:

AddToGroup (Visual Basic) Adds a member to a group

CreateMember (Visual Basic) Creates an Active Directory member

Start This directory contains a variety of introductory examples with each example offered in both

Visual Basic and Visual C++ versions:

Binding Demonstrates binding an Active Directory object

Child Demonstrates reaching a child object in the Active Directory (see Parent)

Create Creates users in organization units

Delete Deletes a user (or another object) from the Active Directory

Enum Shows how to use Active Directory enumerator functions

Filter Shows filtered search on the Active Directory

First Demonstrates creating a user

Move Moves an object to a container object

Parent Demonstrates reaching a parent object in the Active Directory (see Child)

Read Reads a multi-valued attribute from a service object

Rename Similar to the Move example (explanation for name not offered)

Schema Demonstrates retrieving schema information

Search Performs a search for groups within a domain

Write Demonstrates writing elements to a user record

WinNT Offers a series of Windows NT-specific Visual Basic examples:

Binding Demonstrates binding an Active Directory object

ChangePwd Shows how to change a password

Computer Demonstrates connecting to a specific computer

Group Demonstrates creating a group in a domain or local computer

PrintQueue Shows how to enumerate and display print queues and print jobs

ResetPwd Demonstrates resetting a password

Service Enumerates services for a given computer, and demonstrates the ability to start, stop, and pause services

User Demonstrates creating a user, setting a user's properties, renaming a user, and deleting a user

UsrMgr Enumerates users for a given domain or computer; views user/group information including a user's membership; shows adding a user object to or removing a user object from a group

[Previous](#) [Table of Contents](#) [Next](#)

[HOME](#)

[SUBSCRIBE](#)

[SEARCH](#)

[FAQ](#)

[SITEMAP](#)

[CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#). Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
ITKNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++

by Ben Ezzell

Sybex, Inc.

ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

ADsError Source Code Correction

We could call this poetic justice, but beta versions of this example could not be compiled due to a function error in the source code.

The error in the source code, however, comes from a conflict caused because the function `InlineIsEqualGUID` is defined both for the ATL library—which is the one desired for the example program—and in the `GUIDDef.h` header.

In the `IADsError.cpp` source file, the `InterfaceSupportsErrorInfo` function appears as:

```
STDMETHODIMP ADsError::InterfaceSupportsErrorInfo( REFIID riid )
{
    static const IID* arr[] =
    {
        &IID_IADsError,
    };
    for( int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++ )
    {
        // use scope resolution for ATL versus GUIDDEF.h
        if( InlineIsEqualGUID( *arr, riid ) )
            return S_OK;
    }
    return S_FALSE;
}
```

The compiler error will appear at the highlighted line.

To correct the error, simply rewrite this line using the scope resolution operator (`::`) like this:

```
if( ::InlineIsEqualGUID( *arr, riid ) )
```

You can also run the following script to illustrate how to use the `IADsError` interface hosted by the `ADSError.dll`. (This script is titled `Error_Test.vbs` in the Chapter 30 directory on the CD and was supplied by

Max Vaughn at Microsoft.)

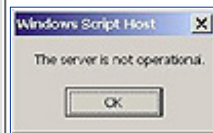
On Error Resume Next

```
dim comobj
set errobj = CreateObject("ADSError")
set comobj = GetObject("LDAP://YOUR_SERVER_NAME_HERE ")
if ( err.number<> 0 ) then
    WScript.Echo errobj.GetErrorMessage(err.Number)
    WScript.Quit 1
Else
    wscript.echo comobj.AdSPath
End If
```

To use the script:

1. Compile the ADsError sample.
2. Use RegSvr32 on ADSError.dll to register the IADsError interface.
3. Run the Error_Test.vbs script.

The function should display the error “Server is not operational” because you will be requesting a bogus computer object from the LDAP provider.



The ADsError.dll is part of the ADSI Reskit and will be shipped as an unsupported use-as-is library. The DLL displays ADSI related messages via a dual-interfaced COM object.

When the (release version) Platform SDK is installed, more information on this topic can be found in the <platform SDK root>/samples/NetDS/ADSI/ directory in the rtk.htm file. This page also offers information on other unsupported interfaces that can be used to retrieve information from the Active Directory via the ADSI.

TIP:

For additional examples, refer also to the ...\\ADSI_SDK\\ResourceKit\\... subdirectories.

With more than 70 examples, the ADSI SDK offers a good variety of programming source code samples. While there is no guarantee that the solution to your specific needs can be found in this assortment, there is certainly a good possibility that something in this collection can offer an approach to your objective.

Summary

Not only is Active Directory new, it is also a massive topic and one not easily covered briefly. Ideally, an entire volume—or several volumes—could be devoted to Active Directory operations and still not completely cover all of the possibilities. In this chapter, we’ve attempted to offer simply an introduction to Active Directory programming operations and to discuss in moderate detail some of the interface methods and how they are used. Then, to offer further opportunities for experimentation, we’ve detailed some of the variety of the example programs provided by the ADI SDK.

Of course, what we have not tried to cover are such topics as how to organize Active Directory structures, permissions, and access, since these topics also fall far beyond the scope of this book (and, themselves, can also fill several volumes).

To borrow from Marvell, “Had we but world enough and time...” virtually every chapter in this book could easily become a book in itself. However, given mortal limitations—and pressure from publishers, etc.—we’ve been forced to pick and choose which topics received coverage and how deeply to go into each subject. As a result, some areas have received more discussion and others less.

Of course, the converse is that others faced with the same choices might well have made different selections ... or, at a different time, our own selections might have been quite different.

The point, of course, is that there are simply too many topics and subtopics to be covered in any single volume, and the selections represented here in this book are simply that: representations of the principal topics that—we hope—will be most useful to you.

Previous	Table of Contents	Next

HOME	SUBSCRIBE	SEARCH	FAQ	SITEMAP	CONTACT US

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.



IT KNOWLEDGE.COMSM
Need IT. Find IT. Know IT.

Enterprise Subscription
IT KNOWLEDGE.COM



KEYWORD SEARCH

[GO!](#)[Search Tips](#)[Advanced Search](#)

PUBLICATION LOOKUP

[GO!](#)

JUMP TO TOPIC



Mastering Windows 2000 Programming with Visual C++
by Ben Ezzell
Sybex, Inc.
ISBN: 0782126421 Pub Date: 02/01/00

Search this book:

[GO!](#)[Previous](#)[Table of Contents](#)[Next](#)

INDEX

Note to the Reader: Page numbers in **bold** indicate the principal discussion of a topic or the definition of a term. Page numbers in *italic* indicate illustrations.

NUMBERS

64-bit version of Windows 2000, 6, 328

A

absolute security descriptors (SDs), 401–403, **402**

abstraction, 850

Accelerated Graphics Port (AGP)

AGP video cards and multiple monitor support, 90
support for, 15

access control entries. *See* ACEs

access control lists. *See* ACLs

access parameters in CreateNamedPipe function, 211–212

access rights. *See also* security

access checking functions, 413–414

adding, 406–407

checking, 406

defined, **406–407**

generic rights, 407

versus privileges, 401

specific rights, 407

- standard rights, 407
- access tokens
 - access token functions, 408–409
 - defined, **400**
- accessing device contexts, 452–453
- ACEs (access control entries)
 - access control lists (ACLs) and, 401, 402, 403, 404
 - ACE functions, 412–413
 - in FileUser demo program, 421–423
- ACLs (access control lists)
 - access control entries (ACEs) and, 401, 402, 403, 404
 - ACL functions, 412
 - defined, **404**
 - discretionary (owner) access control lists (DACLs), 394, 401, 402, 404, 412, 420–423
 - GetAclInformation function, 420–421
 - IsValidAcl function, 421
 - security descriptors (SDs) and, 401, 402, 403, 404, 423
 - system access control lists (SACLs), 394, 401, 402, 412
- acquiring
 - information device contexts, 454–459, 455
 - mutexes and semaphores, 161–162
- ACS (Admission Control Service), 990
- activating COM interfaces, 845–846
- Active Directory, **988–1031**
 - Active Directory Service Interfaces (ADSI), **988, 992–1016**
 - adding libraries to project Link settings, 994, 994
 - ADSI objects, 995–996
 - ADSIAddUser demo program, 1007–1016
 - ADsProperty demo program, 999–1007
 - defined, **988**
 - IADs interface, 996–997
 - IADs interface methods, 997–999
 - IADsUser methods, 1012–1016, 1015
 - implementing IADs interface, 997
 - installing ADSI SDK, 992
 - preparing Visual C++ to use, 992–993, 992, 993
 - AddGroup demo program, **1016–1024**
 - ADsPath specifications, 995, 1017–1018, 1017, 1018
 - CreateDSObject function, 1023
 - CreateGroup function, 1021–1024, 1024
 - overview of, 1016
 - ParseCommandLine function, 1018–1019
 - SysAllocString function, 1019
 - Admission Control Service (ACS), 990
 - defined, **988–991**
 - LDAP (Lightweight Directory Access Protocol), **988, 989–990**
 - Microsoft Message Queue (MSMQ) and, 988
 - other ADSI demo programs, **1024–1031**
 - in ActiveDir subdirectory, 1025–1026

- ADsError source code correction, 1029–1030
- in ASP subdirectory, 1026
- in DataLink subdirectory, 1026
- in DistQuery subdirectory, 1026
- in DsSrch subdirectory, 1027
- in Exchange subdirectory, 1026
- in Extension subdirectory, 1027
- in General subdirectory, 1027
- installing, 1024–1025
- in Internet subdirectory, 1027
- in Interopt subdirectory, 1027
- in Perl subdirectory, 1027
- in SiteServer subdirectory, 1027
- in Start subdirectory, 1027–1028
- in WinNT subdirectory, 1028–1029
- Third Party API, 990
- virtual function tables (VTBLs) and, 988
- Active Template Library (ATL), **896–925**
 - apartments and, 930–931
 - creating in-process COM servers, **847, 896–920**
 - adding a CoClass, 901–902, 902, 903, 910–914
 - adding methods, 906–908, 907, 915–916
 - adding properties, 903–904, 903, 904, 914
 - CComCoClass, CComObjectRoot classes and, 896–897, 897
 - CoCreateInstance function, 918
 - CoInitialize function, 919
 - CoUninitialize function, 919
 - creating DLL files, 909
 - creating MFC clients, 916–918, 916, 917
 - versus creating out-of-process COM servers, 921–923
 - creating and running the test client, 919–920
 - DllMain function, 900
 - editing .cpp files, 919
 - header files, 901
 - implementing methods, 908–909
 - implementing properties, 905–906
 - in-process servers defined, **847**
 - overview of, 897–898, 898
 - source files, 899–901
 - testing ATL servers, 916–920
 - viewing project files, 899, 899
 - Visual Basic and, 907
 - creating out-of-process COM servers, **847, 896–897, 920–925**
 - adding CoClasses, properties, and methods, 923
 - versus creating in-process COM servers, 921–923
 - out-of-process servers defined, **847**
 - overview of, 920, 921
 - SecurityMgrExe.cpp file, 922–923

- StdAfx.h file, 922
 - testing out-of-process servers, 923–925, 924
 - IDispatch Automation with, 937–939
 - overview of, **896–897**, 897
- active window tracking, 109
- ActiveX controls
 - ActiveX Web controls, **775–780**
 - defined, **775**
 - initializing Web browsers, 777–780, 778
 - WebView demo program, 775–780, 776
 - COM objects as, 848
- ActiveX Data Objects (ADO), **602–603**, **681–714**. *See also* OLE DB; Universal Data Access (UDA)
 - AddNew method, 710–712
 - AddNewEntry function, 708–709
 - ADO object model, **683–686**
 - Command objects, 684
 - Connection objects, 684
 - defined, **683–684**, 683
 - Errors collection, 685
 - Fields collection, 685
 - Parameters collection, 685–686
 - Properties collection, 686
 - Recordset objects, 685
 - advantages of, 603
 - availability of, 600
 - Close function, 699
 - CreateInstance function, 694
 - creating ADO database programs, 689–693, 689
 - creating item lists, 696–699, 699
 - cross-platform support with, 604, 604
 - data sources
 - closing connections, 696
 - defined, **682–683**
 - opening connections to, 693–696
 - defined, **602**, **681**
 - FetchSingleTitle function, 701–703
 - GetCollect function, 698
 - Internet advantages of, 603
 - limitations of, 686
 - MoveNext function, 698
 - OLE DB and, 602
 - OnClose function, 696
 - OnDbclkResultList function, 699–700
 - OnInitDialog function, 693–694, 697, 701
 - Open function, 695
 - populating data fields, 700–704, 703
 - PutCollect function, 707
 - records, **704–713**
 - appending, 708–712

- deleting, 712–713
 - FetchRecords function, 696–697, 702–703
 - updating, 704–708
- selecting list box entries, 699–700
- and support for transient environments, 681
- Supports tests, 708, 713–714
- UpdateData function, 699, 703
- UpdateEntry function, 704–705
- Visual Studio 6.0 Enterprise Setup dialog box and, 687–689, 687, 688
- Add Entry form in Booklist program, 670–674, 672
- AddEntry function, 673
- AddGroup demo program, **1016–1024**. *See also* Active Directory
 - ADsPath specifications, 995, 1017–1018, 1017, 1018
 - CreateDSObject function, 1023
 - CreateGroup function, 1021–1024, 1024
 - overview of, 1016
 - ParseCommandLine function, 1018–1019
 - SysAllocString function, 1019
- adding
 - access control entries, 421–423
 - access rights, 406–407
 - a CoClass to in-process COM servers, 901–902, 902, 903, 910–914
 - CoClasses, methods, and properties to out-of-process COM servers, 923
 - Edit operations to Booklist program, **662–675**
 - AddEntry function, 673
 - adding records, 666–670
 - canceling Edit or Add Record operations, 674–675
 - CancelUpdate function, 674
 - changing the Display form, 662–664, 663
 - customizing Add Entry form, 670–674, 672
 - direct SQL statements, 669–670
 - generating record IDs, 667–668
 - IsOnFirstRecord and IsOnLastRecord functions, 666
 - LoadDealerList function, 668–669
 - MoveFirst function, 675
 - NewTitleID function, 667–668, 671
 - OnCancel function, 674
 - OnEditEntry function, 674
 - OnEditTitle function, 662–663
 - OnInitialUpdate function, 668
 - OnNewTitle function, 670–671
 - populating data from another table, 668
 - Requery function, 664
 - ResetControls function, 663
 - resetting controls, 665–666
 - resetting edit fields, 664–665
 - Update function, 674
 - UpdateData function, 671–673, 675

- encryption support to applications, **432–444**
 - and compromising security, 442–444
 - creating default Cryptographic Service Providers, 435, 435, 436
 - creating random keys, 437–440, 439
 - CryptAcquireContext function, 440
 - CryptDecrypt function, 441
 - CryptEncrypt function, 438–439
 - CryptEncryptMessage function, 440
 - CryptExportKey function, 438
 - CryptExportPublicKeyInfo function, 440
 - CryptGetUserKey and CryptGenKey functions, 437–438
 - CryptImportKey function, 440
 - decrypting files, 440–441, 441
 - encrypting files, 436–440
 - installing Crypto API runtime support, 432–433
 - project settings, 434–435, 434
 - ReportStatus function, 437
 - libraries to ADSI project Link settings, 994, 994
 - methods to in-process COM servers, 906–908, 907, 915–916
 - properties to in-process COM servers, 903–904, 903, 904, 914
 - records in Booklist program, 647–648, 666–670
 - table access to Publisher program, **632–636**
 - CDealerSet and CDealerForm classes, 636
 - creating classes for record views, 634–635, 635
 - creating procedure for, 632–634, 633, 634
- AddNew function, 648, 650
- AddNew method in ADO, 710–712
- AddRef method, 884, 893
- address space, 338–339, 338, 349–350
- Admission Control Service (ACS), 990
- ADO. *See* ActiveX Data Objects
- ADSI. *See* Active Directory
- ADsPath specifications, 995, 1017–1018, 1017, 1018
- Advanced Power Management (APM) system, 309
- AfxBeginThread function, 152
- AGP (Accelerated Graphics Port)
 - AGP video cards and multiple monitor support, 90
 - support for, 15
- allocating memory
 - allocating heap, 351–352, 355
 - defined, **336–337**
 - GlobalAlloc and LocalAlloc functions, 354, 355
 - heap and, 350
 - system limits and, 355
 - VirtualAlloc function, 341–345, 355, 359, 366
- Alpha processors, 7
- Alphablend function, 108
- animation, 510
- anisotropic mapping mode, 480–481, 481

anonymous pipes. *See also* pipes

AnonPipe demo program, **195**, **231–247**

child process, **243–247**

CloseHandle function, 243

CreatePipe function, 237

CreateProcess function, 237–241, 243, 245

creating parent and child pipes, 236–241

defined, **231–232**, 232

DuplicateHandle function, 237

GetStdHandle function, 238, 245

inheriting pipe handles and creating threads, 244–245

initializing the parent process, 233

parent process, **232–243**

ReadFile function, 247

reading from pipes, 245–247

responding to system messages, 233–236, 234

SendCommand function, 241–243

StartProcess function, 237, 238–241

warning about testing, 195

WriteFile function, 242

writing to pipes, 241–243

creating with CreatePipe function, 208, 237

defined, **193–194**

anti-virus software, 429

apartments. *See also* COM servers, threads

Active Template Library and, 930–931

defined, **927–928**, 928

multithreaded apartments (MTAs), 927–929, 928

primary versus secondary single-threaded apartments (STAs), 929

single-threaded apartments (STAs), 927–928, 928

APIs. *See also* functions

COM synchronization API, 954–955

CoWaitForMultipleHandles API function, 954–955

Crypto API functions, **429–432**

CryptAcquireContext, 440

CryptDecrypt, 441

CryptEncrypt, 438–439

CryptEncryptMessage, 440

CryptExportKey, 438

CryptExportPublicKeyInfo, 440

CryptGetUserKey, 437–438

CryptImportKey, 440

Cryptographic Service Provider (CSP) functions, 430

encryption functions, 431

hashing functions, 431–432

installing runtime support for, 432–433

key functions, 430–431

overview of, 429

Internet API functions, **752–774**

- AutoDial functions, 755
- connect and disconnect functions, 754–755
- cookie functions, 761
- defined, **752–753**
- error and status functions, 762–763
- file locating, reading, and writing functions, 758–759
- FindOneOf, 768
- FTP, Gopher, and HTTP functions, 759–760
- FtpGet demo program, 764–774, 765
- FtpGetFile, 760, 764, 771–773
- FtpOpenFile, 760, 774
- GetLastError, 762, 766
- Internet applications, 774
- InternetAttemptConnect, 754, 756
- InternetAutodial, 755, 757–758
- InternetConnect, 755, 764, 769–770
- InternetGetConnectedState, 755, 756–757, 758
- InternetGetLastResponseInfo, 762, 766
- InternetOpen, 755, 764, 768
- InternetOpenUrl, 762, 774
- InternetReadFile, 759, 774
- LastErrorMsg, 766
- limitations of, 753
- MFC wrapper classes, 763–764
- OnOK, 767
- opening dial-up connections, 756–758
- time/date conversion functions, 762
- URL-related functions, 761–762

Internet-related APIs, 723–725

LAN Manager API functions, 792

LDAP (Lightweight Directory Access Protocol) API, 990

Registry API functions

- defined and listed, **276–278**
- RegCreateKey, 280–281, 285, 287
- RegCreateKeyEx, 288
- RegEnumKey, 284–285
- RegOpenKeyEx, 288
- RegQueryInfoKey, 282–283
- RegQueryValueEx, 274, 285
- RegSetValueEx, 274, 287–289

security API functions, **391–392, 407–417**

- access checking functions, 413–414
- access control entry (ACE) functions, 412–413
- access control list (ACL) functions, 412
- access token functions, 408–409
- auditing functions, 417
- client impersonation functions, 409–410
- converting self-relative SDs to absolute SDs, 403

- discretionary (owner) access control list (DACL) functions, 412
- GetAclInformation, 420–421
- GetSecurityDescriptorDacl, 420–421
- IsValidAcl, 421
- local service authority (LSA) functions, 415–416
- overview of, 407–408
- privilege functions, 415
- security descriptor (SD) functions, 401, 410
- security identifier (SID) functions, 411
- Set/Get security information functions, 416
- SetFileSecurity, 423–424
- system access control list (SACL) functions, 412
- window station functions, 417
- in Windows 95/98 versus Windows 2000, 391–392
- Third Party API, 990
- virtual memory APIs, 337
- Win32 APIs
 - defined, **142**
 - thread methods and, 142
- Winsock 2 API functions, **730–752**
 - asynchronous functions, 735–738
 - database-lookup functions, 732–735
 - defined, **730**
 - FindAddr demo program, 738–743, 739, 740
 - gethostbyaddr, 732, 742
 - gethostbyname, 732, 741–742, 745–746, 747
 - HOSTS and SERVICES files, 733–735
 - network byte order and host byte order conversion functions, 731–732
 - SendMail demo program, 743–750, 744
 - SendMail2 demo program, 751
 - socket functions, 730–731
 - Winsock applications, 751–752
- WNet API functions, **787–791**
 - defined, **787–788**
 - network resources and, 788–789
 - WNetDemo program, 789–791, 790
- APM (Advanced Power Management) system, 309
- appending records
 - in ADO programming, 708–712
 - in Booklist program, 650
- AppleTalk protocol, 794–799, 798
- application clusters, 871
- Application Programming Interfaces. *See* APIs
- application windows in “Hello, World” example, 34–36
- applications
 - adding encryption support to applications, **432–444**
 - and compromising security, 442–444
 - creating default Cryptographic Service Providers, 435, 435, 436

- creating random keys, 437–440, 439
- CryptAcquireContext function, 440
- CryptDecrypt function, 441
- CryptEncrypt function, 438–439
- CryptEncryptMessage function, 440
- CryptExportKey function, 438
- CryptExportPublicKeyInfo function, 440
- CryptGetUserKey and CryptGenKey functions, 437–438
- CryptImportKey function, 440
- decrypting files, 440–441, 441
- encrypting files, 436–440
- installing Crypto API runtime support, 432–433
- project settings, 434–435, 434
- ReportStatus function, 437
- application-defined security objects, 395
- built-in Windows NT/2000 security versus application security, 395–396
- creating power-aware applications, 98–99
- creating “power-friendly” applications, 101–102
- displaying multiple applications on multiple monitors, 93
- dual-installing on dual-boot systems, 23–24
- Internet applications, 774
- multithreaded Win32 applications, 926–927
- switching application views, 642–645
 - preventing problems with, 645–647
- Winsock applications, 751–752
- approximation SIMD operations, 580
- arguments
 - arguments parameters for CreateProcess function, 195–196
 - in WinMain procedure, 30–32
- arithmetic SIMD instructions, 579–580
- assigning priority classes to processes, 203
- AssignProcessToJobObject function, 105
- Asynchronous COM, **957–972**. *See also* COM in Windows 2000
 - asynchronous clients, 964
 - asynchronous interface calls, 960–964
 - asynchronous interface construction, 958–960
 - asynchronous server modifications, 970–971
 - asynchronous server processing modes, 964–966, 965
 - asynchronous server processing runtime events, 966–967
 - call object implementation, 967–970
 - call serialization and auto complete, 971–972
 - CoCreateInstance function, 962
 - CoInitialize function, 962
 - CreateCall function, 963
 - defined, **957–958**
 - ProcessData function, 962–963, 969
 - Visual C++ and, 960
- asynchronous I/O parameter for pipe-related functions, 224
- asynchronous Winsock 2 API functions, 735–738

- ATL. *See* Active Template Library
- attacks on encryption, 443–444
- attributes
 - file attributes in NTFS file system, 68–69
 - security attributes
 - for CreateNamedPipe function, 216
 - for CreateProcess function, 196
 - SECURITY_ATTRIBUTES structure, 398–400
 - thread object attributes, 141–142, 142
- audio. *See* sound
- auditing functions, 417
- authentication, 427
- Authenticode support, 389
- auto complete, 971–972
- AutoChk utility, 63
- AutoDial functions, 755
- Automation of IDispatch interface, **834–836**, **931–949**. *See also* COM servers
 - with Active Template Library, 937–939
 - Automation data types, 939–943
 - BSTR data type, 939–940
 - C++ Automation clients, 945–948
 - CoCreateInstance function, 947
 - CoInitialize function, 948
 - CURRENCY data type, 940
 - dispinterfaces in Visual C++, 934–935, 936
 - dual-interface components in Visual C++, 936–937, 936
 - GetIdsOfNames method, 933, 947
 - IDispatch interface defined, **834–836**, **932**
 - implementing IDispatch interface with Visual C++, 932–937
 - IsUserAllowed method, 947
 - overview of, 931–932
 - SAFEARRAY data type, 941
 - type libraries and, 943–945, 944, 945
 - VARIANT data type, 942–943
 - Visual Basic Automation clients, 948–949

B

- background processes, 139
- base priority of threads, 143–145, 144
- Be operating system multiboot manager, 19
- BeginPaint function, 452–453
- BeginTrans function, 652–653
- binary compatibility, 826
- binding, 852–853
- bitmap operations, 473
- block encryption, 428
- blocking inheritance, 196–197
- blocking parameter for pipe-related functions, 224

blocking pipes, 192

Booklist demo program, **647–675**. *See also* databases; Publisher demo program

adding Edit operations, **662–675**

AddEntry function, 673

adding records, 666–670

canceling Edit or Add Record operations, 674–675

CancelUpdate function, 674

changing the Display form, 662–664, 663

customizing Add Entry form, 670–674, 672

direct SQL statements, 669–670

generating record IDs, 667–668

IsOnFirstRecord and IsOnLastRecord functions, 666

LoadDealerList function, 668–669

MoveFirst function, 675

NewTitleID function, 667–668, 671

OnCancel function, 674

OnEditEntry function, 674

OnEditTitle function, 662–663

OnInitialUpdate function, 668

OnNewTitle function, 670–671

populating data from another table, 668

Requery function, 664

ResetControls function, 663

resetting controls, 665–666

resetting edit fields, 664–665

Update function, 674

UpdateData function, 671–673, 675

adding and updating records, **647–648**

BeginTrans, CommitTrans, and Rollback functions, 652–653

creating classes, 653–654

CRecordset update transactions, **648–651**

AddNew function, 648, 650

appending records, 650

CancelUpdate function, 649

CanUpdate, CanAppend, and CanRestart functions, 650

checking the legality of operations, 650–651

CRecordset class, 648–649

Delete function, 649

Edit function, 649

editing records, 649

IsOpen, IsBOF, IsEOF, and IsDeleted functions, 651

locking records, 651

SetLockingMode function, 651

Update function, 649, 650

defined, **653–655**, 654

GetCursorCommitBehavior and GetCursorRollbackBehavior functions, 653

RFX_XXXX function calls, 648, 655–659, 656

sequencing operations, 661–662, 661

tracing SQL operations, 659–660

- transactions, **652–653**
- Boot Magic software, 19–20
- broadcast SIMD shuffle operation, 581, 588–589
- Browser demo program, **377–385**. *See also* file-mapping objects
 - ChangeAccess function, 384–385
 - CreateFile function, 380
 - CreateFileMapping function, 379–380
 - creating and using mapped files, 378–379
 - defined, **377–378**, 378
 - DoWrite, DoRead, and DoClear functions, 382–384
 - editing shared objects, 383–385
 - FindWindow function, 380
 - initializing mapped files, 379–380
 - OpenFileMapping function, 380
 - running the dialog box, 380–382
- brute force security attacks, 443
- BSTR data type, 939–940
- buffers. *See also* memory
 - buffer sizes in CreateNamedPipe function, 215
 - FlushFileBuffers function, 227
 - Translation Look-aside Buffers (TLBs), 333, 943
- byte padding, 428
- byte pipes, 192

C

C language

- C exception handling, 297
- C runtime library
 - CopyMemory, FillMemory, MoveMemory, and ZeroMemory commands, 356
 - CreateProcess equivalents, 201
 - memory function equivalents, 356
 - thread-related function equivalents, 158–159

C++ language. *See also* Visual C++

- C++ Automation clients, 945–948
- C++ exception handling, 297

cacheability control instructions, **589–591**. *See also* SIMD

- defined, **589**
- move masked byte to memory instruction, 590
- nontemporal, temporal, and spatially cached data, 589–590
- prefetch instructions, 590
- store fence instruction, 591
- streaming store instructions, 589–590

calculating pop-up window size, 126–128

CalcWndSize method, 123

call objects and call cancellation, **979–981**. *See also* COM in Windows 2000

- call cancellation handling, 980–981
- call cancellation requests, 979–980
- call object implementation, 967–970

- defined, **979**
- call serialization, 971–972
- calling
 - COM interfaces, 845–846
 - threads, 929–930
- CallNamedPipe function, 228, 229
- canceled Edit or Add Record operations, 674–675
- CancelUpdate function, 649, 674
- CanUpdate, CanAppend, and CanRestart functions, 650
- captions for windows, 631–632
- capturing mouse and keyboard input for pop-up message windows, 125–126
- Cartesian coordinate mapping modes, 479–480, 480
- catch statements. *See* exception handling, Exceptions demo program
- CComCoClass and CComObjectRoot classes, 896–897, 897
- CDealerSet and CDealerForm classes, 636
- CFile:Open function, 83
- CFileDialog class, 70–71
- Change Journal, 94
- ChangeAccess function, 384–385
- changing
 - classes after creation, **636–647**
 - CheckOrders function, 645–646
 - Class Info tab in Class Wizard and, 636, 636
 - customizing COrderSet and COrderForm classes, 637–642
 - defining Filter parameter, 638–640
 - GetDocument function, 641
 - initializing COrderForm class, 640–642
 - OnActiveView function, 645
 - OnInitialUpdate function, 640, 641, 642
 - OnShowOrders function, 644
 - overview of, 636–637
 - preventing problems with switching application views, 645–647
 - SelectForm function, 644, 645, 646
 - switching application views, 642–645
 - Display form in Booklist program, 662–664, 663
 - volume in ShowWave demo program, 561–563, 562
- channels, Secure Channels support, 389
- Check function, 419
- checking
 - access rights, 406, 413–414
 - the legality of operations in database applications, 650–651
- CheckMenuItem and CheckMenuRadioItem functions, 131
- CheckOrders function, 645–646
- child pipes, 236–241
- child processes
 - in AnonPipe demo program, 243–247
 - defined, **139, 187–189**
 - launching, 253–255
 - letting children inherit handles in CreateProcess function, 204–205

- in NamePipe demo program, 262–265
- ChkDsk utility, 63
- chunk operations using multimedia I/O functions, 520–521
- classes
 - assigning priority classes to processes, 203
 - CComCoClass and CComObjectRoot, 896–897, 897
 - CDealerSet and CDealerForm classes, 636
 - CFileDialog, 70–71
 - changing after creation, **636–647**
 - CheckOrders function, 645–646
 - Class Info tab in Class Wizard and, 636, 636
 - customizing COrderSet and COrderForm classes, 637–642
 - defining Filter parameter, 638–640
 - GetDocument function, 641
 - initializing COrderForm class, 640–642
 - OnActiveView function, 645
 - OnInitialUpdate function, 640, 641, 642
 - OnShowOrders function, 644
 - overview of, 636–637
 - preventing problems with switching application views, 645–647
 - SelectForm function, 644, 645, 646
 - switching application views, 642–645
 - class factories, 846, 884–888
 - CNTFS_OpsDlg, 70–71
 - CoClass
 - adding to in-process COM servers, 901–902, 902, 903, 910–914
 - adding to out-of-process COM servers, 923
 - defined, **876**
 - creating
 - in Booklist program, 653–654
 - for record views, 634–635, 635
 - CRecordset update transactions, **648–651**
 - AddNew function, 648, 650
 - appending records, 650
 - CancelUpdate function, 649
 - CanUpdate, CanAppend, and CanRestart functions, 650
 - checking the legality of operations, 650–651
 - CRecordset class, 648–649
 - Delete function, 649
 - Edit function, 649
 - editing records, 649
 - IsOpen, IsBOF, IsEOF, and IsDeleted functions, 651
 - locking records, 651
 - SetLockingMode function, 651
 - Update function, 649, 650
 - CTipWnd, 122–125
 - CTitleForm, **628–632**
 - GetTableName function, 632

- header file, 628–631
- OnGetRecordset function, 629, 630
- OnInitialUpdate function, 630, 632
- setting sort order, 630–631
- setting window captions, 631–632
- CTitleSet, **623–628**
 - DoDataExchange function, 627–628
 - DoFieldExchange function, 625–626
 - GetDefaultConnect function, 626
 - GetDefaultConnectSQL function, 626–627
 - header file, 623–627
 - transferring data from databases to recordsets, 627–628
- versus interfaces, 833
- MFC (Microsoft Foundation Class)
 - creating MFC clients for testing in-process COM servers, 916–918, 916, 917
 - creating MFC threads with CWinThread class, 151–153
 - “Hello, World” example and, 30, 44
 - MFC exception classes, 305–306, 312–316
 - MFC exception handling, 297
 - MFC wrapper classes, 763–764
 - SQL (Structured Query Language) and, 612–613
 - Template example and, 48
- for record views, 634–635, 635
- registering window classes
 - in “Hello, World” example, 32–34
 - in ShowWave demo program, 566–568
- Remote Procedure Call interface classes, 803
- client windows, 40
- clients
 - asynchronous COM clients, 964
 - C++ Automation clients, 945–948
 - client impersonation
 - client impersonation functions, 409–410
 - defined, **405–406**
 - ImpersonateNamedPipeClient function, 229–230
 - client-side dead letter queues, 869
 - COM objects as COM clients, 846–847
 - creating MFC clients for testing in-process COM servers, 916–918, 916, 917
 - QC client components, 867
 - RPC client testing, 818–819, 818
 - RPCClient demo program, 812–814
 - security and client/server model, 396–398
- clipping capabilities of devices, 464, 473–474
- Close function, 699
- CloseHandle function
 - in pipe-related operations, 227, 229, 230, 243
 - in thread-related operations, 148, 150–151, 156, 157
- closing
 - data source connections, 696

- multimedia devices, 524–527
- ShowWave demo program, 550–553
- CNTFS_OpsDlg class, 70–71
- CNTFS_StreamDlg::OnRead function, 83–84
- CoClass class
 - adding to in-process COM servers, 901–902, 902, 903, 910–914
 - adding to out-of-process COM servers, 923
 - defined, **876**
- CoCreateInstance function
 - Asynchronous COM and, 962
 - creating COM servers, 885, 893, 895
 - creating in-process COM servers, 918
 - IDispatch interface Automation and, 947
- coherence in file-mapping, 375–377, 376
- CoInitialize function, 919, 929, 948, 962
- CoInitializeEx function, 929
- color capabilities of devices, 461, 467–469, 467
- COM (Component Object Model), **599–600, 801–802, 824–984**. *See also* Active Directory
 - advantages of, **826–828**
 - adaptability of components to changing business requirements, 827
 - facilitation of parallel development, 828
 - replaceability of components, 827
 - reusability of components, 828
 - binary compatibility, 826
 - COM+ model, **855–863**. *See also* queued components (QC)
 - COM+ load balancing, 870–871
 - versus Component Object Model, 855–856
 - Component Services Snap-in or COM+ Explorer, 858–860
 - database tier technologies, 858
 - defined, **856**
 - Forms+ technology, 856, 857
 - Microsoft Distributed Transaction Coordinator (MS DTC) and, 859–860, 861
 - Microsoft Message Queue (MSMQ) and, 857–858, 863, 870
 - Microsoft Transaction Server (MTS) and, 857–858, 872
 - middle tier technologies, 857–858
 - Storage+ technology, 856–857, 858
 - transactions, 861–863, 862
 - user interface tier technologies, 857
 - Windows DNA (Distributed interNet Applications Architecture) and, 856–858
- COM objects, **846–848**
 - as ActiveX controls, 848
 - as COM clients, 846–847
 - defined, **848**
 - as in-process or out-of-process server objects, 847
- Component Services Explorer tool, 858, 859
- Component Services Snap-in or COM+ Explorer, 858–860
- components
 - adaptability of to changing business requirements, 827

- component code, 825
- component or load-balancing routers, 871
- defined, **825, 848**
- dual-interface components, 836–837, 837
- dual-interface components in Visual C++, 936–937, 936
- GUIDs (globally unique identifiers) and, 828–829
- implement IUnknown rule, 844
- implementation rules, **843–845**
- interfaces and, 830, 832
- memory management rule, 844
- versus objects, 848
- reference counting rules, 845
- replaceability of, 827
- reusability of, 828
- threading, 927–931, 928
- versions of, 828–829
- virtual function tables, vtables, or VTBLs and, 831, 988

Computer Management Explorer, 860, 860

DCOM (Distributed Component Object Model), 801–802

defined, **824–825**

dynamic load balancing, **870–871**

- application clusters and, 871
- COM+ load balancing, 870–871
- component or load-balancing routers, 871
- defined, **870**
- Windows Load Balancing (WLB), 870–871

GUIDs (globally unique identifiers)

- COM and, 826
- components and, 828–829
- creating, 839, 842
- defined, **802–803, 826**
- interfaces and, 826, 833, 839, 842

IDispatch interface, **834–836, 931–949**

- Automation with Active Template Library, 937–939
- Automation data types, 939–943
- BSTR data type, 939–940
- C++ Automation clients, 945–948
- CoCreateInstance function, 947
- CoInitialize function, 948
- CURRENCY data type, 940
- defined, **834–836, 932**
- dispinterfaces in Visual C++, 934–935, 936
- dual-interface components in Visual C++, 936–937, 936
- GetIdsOfNames method, 933, 947
- implementing with Visual C++, 932–937
- IsUserAllowed method, 947
- overview of, 931–932
- SAFEARRAY data type, 941
- type libraries and, 943–945, 944, 945

- VARIANT data type, 942–943
- Visual Basic Automation clients, 948–949
- interfaces, **825–827**, **837–846**
 - activating and calling, 845–846
 - asynchronous interface construction, 958–960
 - asynchronous pipe interfaces, 977–978
 - characteristics of, 833
 - versus classes, 833
 - for COM servers, 876
 - COM synchronization interfaces, 955–957
 - components and, 830, 832
 - creating unique interface identifiers, 839, 842
 - custom interfaces, 836, 880–884
 - defined, **825–827**, **831–833**, 831
 - deriving interfaces from IUnknown interface, 831, 838–839
 - designing, 837–843
 - designing for remote use, 838
 - dual-interface components, 836–837, 837
 - GUIDs (globally unique identifiers) and, 826, 833, 839, 842
 - how interfaces work, 831–832, 831
 - IDispatch interface, 834–836, 931–949
 - immutability of, 832, 833, 839
 - interface IDs (IIDs), 839, 842
 - interface inheritance, 853–855
 - interface inheritance using early binding, 852–853
 - interface pointers, 831
 - interface rules, 837–841
 - interface semantics, 843
 - interface signatures or syntax, 842–843
 - IObjectControl interface, 872
 - ISynchronize interface, 955–956
 - ISynchronizeContainer interface, 956
 - ISynchronizeHandle interface, 956–957
 - IUnknown interface, 831, 834, 838–839, 844, 881–884
 - MIDL (Microsoft Interface Definition Language) and, 836, 876
 - versus objects, 833
 - persistence of, 829
 - planning, 829, 832
 - querying for object interfaces, 851–852
 - returning HRESULT type values, 839–841, 883, 893
 - string parameters as Unicode strings, 841
 - type libraries and, 837
 - virtual function tables, vtables, or VTBLs and, 831, 988
 - Visual Basic, Visual C++ and, 832
- IUnknown interface
 - and creating COM servers, 881–884
 - defined, **834**
 - deriving interfaces from, 831, 838–839

- implement IUnknown rule, 844
- limitations of, **828–829**
 - component versions, 828–829
 - interface persistence, 829
 - interface planning, 829, 832
- object pooling, **872**
- object-oriented programming concepts, **848–855**
 - abstraction, 850
 - Component Object Model and, 848–849
 - encapsulation, 849–850
 - interface inheritance, 853–855
 - interface inheritance using early binding, 852–853
 - late binding, 852
 - polymorphism, 851–853
- queued components (QC), **863–870**
 - architecture of, 865, 865
 - asynchronous calling mechanism, 957
 - client-side dead letter queues, 869
 - defined, **863**
 - failure recovery, 867–869
 - limitations of real-time architectures, 863–864
 - Microsoft Message Queue (MSMQ) and, 863, 870
 - QC client components, 867
 - QC server components, 866
 - runtime architecture, 865–867
 - security, 869–870
 - server-side retry queues, 867–869
 - transactional message queuing, 864, 864
- Service Control Manager (SCM), 846
- Universal Data Access (UDA) architecture and, 599–600
- Visual Basic, Visual C++ and, 832, 848
- COM servers, **876–950**
 - CoClass class
 - adding to in-process COM servers, 901–902, 902, 903, 910–914
 - adding to out-of-process COM servers, 923
 - defined, **876**
 - creating, **879–896**
 - AddRef method, 884, 893
 - class factories and, 884–888
 - CoCreateInstance function, 885, 893, 895
 - completing the COM server, 884–893
 - CoUninitialize function, 893
 - CreateInstance function, 885–887
 - creating DLL files, 888–893
 - creating test clients, 893–896, 895
 - defining custom interfaces, 880–881
 - DllCanUnloadNow function, 891–892, 895
 - DllGetClassObject function, 892, 895
 - DllRegisterServer function, 892, 893

- DllUnregisterServer function, 892
- HRESULT data type and, 883, 893
- implementing IUnknown and the custom interface, 881–884
- LockServer function, 885–887
- overview of, 879
- QueryInterface method, 884, 888
- Release method, 884, 888, 893
- creating in-process COM servers with Active Template Library (ATL), **847, 896–920**
 - adding a CoClass, 901–902, 902, 903, 910–914
 - adding methods, 906–908, 907, 915–916
 - adding properties, 903–904, 903, 904, 914
 - CComCoClass, CComObjectRoot classes and, 896–897, 897
 - CoCreateInstance function, 918
 - CoInitialize function, 919
 - CoUninitialize function, 919
 - creating DLL files, 909
 - creating MFC clients, 916–918, 916, 917
 - versus creating out-of-process COM servers, 921–923
 - creating and running the test client, 919–920
 - DllMain function, 900
 - editing .cpp files, 919
 - header files, 901
 - implementing methods, 908–909
 - implementing properties, 905–906
 - in-process servers defined, **847**
 - overview of Active Template Library, 896–897, 897
 - overview of, 897–898, 898
 - source files, 899–901
 - testing ATL servers, 916–920
 - viewing project files, 899, 899
 - Visual Basic and, 907
- creating out-of-process COM servers with Active Template Library (ATL), **847, 896–897, 920–925**
 - adding CoClasses, properties, and methods, 923
 - versus creating in-process COM servers, 921–923
 - out-of-process servers defined, **847**
 - overview of Active Template Library, 896–897, 897
 - overview of, 920, 921
 - SecurityMgrExe.cpp file, 922–923
 - StdAfx.h file, 922
 - testing out-of-process servers, 923–925, 924
- IDispatch interface, **834–836, 931–949**
 - Automation with Active Template Library, 937–939
 - Automation data types, 939–943
 - BSTR data type, 939–940
 - C++ Automation clients, 945–948
 - CoCreateInstance function, 947
 - CoInitialize function, 948
 - CURRENCY data type, 940

- defined, **834–836, 932**
- dispinterfaces in Visual C++, 934–935, 936
- dual-interface components in Visual C++, 936–937, 936
- GetIdsOfNames method, 933, 947
- implementing with Visual C++, 932–937
- IsUserAllowed method, 947
- overview of, 931–932
- SAFEARRAY data type, 941
- type libraries and, 943–945, 944, 945
- VARIANT data type, 942–943
- Visual Basic Automation clients, 948–949
- IDL files, **876–879**
- interfaces for, 876
- Microsoft Interface Definition Language (MIDL) and, 876
- threads, **926–931**
 - apartments and Active Template Library, 930–931
 - apartments defined, **927–928, 928**
 - CoInitialize and CoInitializeEx functions, 929
 - execution contexts, 897
 - InterlockedIncrement and InterlockedDecrement functions, 931
 - marshaling services, 930
 - multithreaded apartments (MTAs), 927–929, 928
 - multithreaded Win32 applications, 926–927
 - object threading models and calling threads, 929–930
 - primary versus secondary single-threaded apartments (STAs), 929
 - single-threaded apartments (STAs), 927–928, 928
 - threading COM components, 927–931, 928
- type libraries and, 876
- COM in Windows 2000, **954–984**
 - Asynchronous COM, **957–972**
 - asynchronous clients, 964
 - asynchronous interface calls, 960–964
 - asynchronous interface construction, 958–960
 - asynchronous server modifications, 970–971
 - asynchronous server processing modes, 964–966, 965
 - asynchronous server processing runtime events, 966–967
 - call object implementation, 967–970
 - call serialization and auto complete, 971–972
 - CoCreateInstance function, 962
 - CoInitialize function, 962
 - CreateCall function, 963
 - defined, **957–958**
 - ProcessData function, 962–963, 969
 - Visual C++ and, 960
 - call objects and call cancellation, **979–981**
 - call cancellation handling, 980–981
 - call cancellation requests, 979–980
 - defined, **979**

- COM pipes, **972–978**
 - asynchronous pipe interfaces and read-ahead, 977–978
 - COM pipe interfaces, 972–977
 - defined, **972**
 - IPipeByte interface, 972–977
- lightweight handlers (LWHs), **981–983**
 - custom lightweight handlers, 983, 983
 - defined, **981**
 - standard lightweight handlers, 981–982, 982
 - Web site about, 983
- synchronization mechanisms, **954–957**
 - COM synchronization API, 954–955
 - COM synchronization interfaces, 955–957
 - CoWaitForMultipleHandles API function, 954–955
 - history of, 954
 - ISynchronize interface, 955–956
 - ISynchronizeContainer interface, 956
 - ISynchronizeHandle interface, 956–957
- Command objects in ADO, 684
- CommitMemFilter function, 360–361
- committed pages, 333
- committing memory, 336–337, 349–350
- CommitTrans function, 652–653
- common controls support, 108
- Compact Disc-Digital Audio (Red Book audio) format, 510
- Compaq Alpha processors, 7
- compare and set EFLAGS SIMD instructions, 584–585
- compare and set mask SIMD instructions, 583
- compatibility
 - binary compatibility, 826
 - of hardware, 6–7, 8
 - long filename compatibility between file systems, 78
 - multiple monitor compatibility issues, 93–94
- compound multimedia devices, 525
- compromising security, 442–444
- Computer Management Explorer, 860, 860
- configuring multiple monitors, 91, 92
- connect Internet API functions, 754–755
- connecting with clients in NamePipe demo program, 257–260
- Connection objects in ADO, 684
- ConnectNamedPipe function, 225–226, 227, 257
- constants, predefined constants, 55–56
- control status register (MXCSR), 591–592
- controls. *See also* ActiveX controls
 - common controls support, 108
- conversion operation SIMD instructions, 585
- converting
 - network byte order and host byte order conversion functions, 731–732
 - NTFS filenames to 8.3 format, 64

- selectors to physical memory addresses, 331
- self-relative security descriptors to absolute security descriptors, 403
- time/date conversion functions in Internet API, 762
- cookies
 - defined, **761**
 - Internet API cookie functions, 761
- CopyMemory command, 356
- copy-on-write protection, 377
- COrderForm class
 - customizing, 637–642
 - initializing, 640–642
- COrderSet class, 637–642
- CoUninitialize function, 893, 919
- .cpp files, 919
- CPUs. *See* processors
- crashes, NTFS file system and, 63
- CreateDC function, 453, 454
- CreateDSObject function, 1023
- CreateEvent function, 163
- CreateEx function, 124
- CreateFile function
 - in file-mapping operations, 370, 380
 - in pipe-related operations, 218–219, 225, 229, 263–264
 - security and, 398–399
- CreateFileMapping function, 370, 371–373, 379–380
- CreateGroup function, 1021–1024, 1024
- CreateHardLink function, 85–86
- CreateIC function, 453, 454
- CreateInstance function, 694, 885–887
- CreateJobObject function, 105
- CreateList function, 358–359
- CreateNamedPipe function, **209–218, 251–253**
 - access, write-through, and overlapping parameters, 211–212
 - buffer sizes, 215
 - defined, **209–210**
 - in NamePipe demo program, 251–253
 - naming pipes, 210–211
 - overview of parameters, 216–217
 - pipe instances, 214–215
 - return values, 217–218
 - security attributes parameter, 216
 - time-out period parameter, 216
 - type, read mode, and wait parameter, 212–214
- CreatePopupMenu method, 130
- CreateProcess function, **194–201, 203, 204–205**
 - in AnonPipe demo program, 237–241, 243, 245
 - assigning priority classes, 203
 - blocking inheritance, 196–197

- C runtime library equivalents, 201
- defined, **194**
- environment and directory parameters, 197–198
- executable files and arguments parameters, 195–196
- I/O handles, 200
- letting children inherit handles, 204–205
- in NamePipe demo program, 253–255, 262
- parameters pointing to structures, 198–200
- process type and priority parameters, 197
- return values, 201
- security attributes, 196
- CreateThread function, 148–151, 152–153, 159
- CreateWindow function, 34–36, 339
- CreateWindows function, 167, 168–169
- creating
 - ADO database programs, 689–693, 689
 - anonymous pipes with CreatePipe function, 208, 237
 - application windows in “Hello, World” example, 34–36
 - classes in Booklist program, 653–654
 - classes for record views, 634–635, 635
 - COM servers, **879–896**
 - AddRef method, 884, 893
 - class factories and, 884–888
 - CoCreateInstance function, 885, 893, 895
 - completing the COM server, 884–893
 - CoUninitialize function, 893
 - CreateInstance function, 885–887
 - creating DLL files, 888–893
 - creating test clients, 893–896, 895
 - defining custom interfaces, 880–881
 - DllCanUnloadNow function, 891–892, 895
 - DllGetClassObject function, 892, 895
 - DllRegisterServer function, 892, 893
 - DllUnregisterServer function, 892
 - HRESULT data type and, 883, 893
 - implementing IUnknown and the custom interface, 881–884
 - LockServer function, 885–887
 - overview of, 879
 - QueryInterface method, 884, 888
 - Release method, 884, 888, 893
 - CTipWnd class, 122–125
 - default Cryptographic Service Providers, 435, 435, 436
 - DLL files
 - for COM servers, 888–893
 - for in-process COM servers, 909
 - file-mapping objects, 371–373
 - GUIDs (globally unique identifiers), 839, 842
 - heap, 350–351
 - in-process COM servers with Active Template Library (ATL), **896–920**

- adding a CoClass, 901–902, 902, 903, 910–914
- adding methods, 906–908, 907, 915–916
- adding properties, 903–904, 903, 904, 914
- CCoComClass, CComObjectRoot classes and, 896–897, 897
- CoCreateInstance function, 918
- CoInitialize function, 919
- CoUninitialize function, 919
- creating DLL files, 909
- creating MFC clients, 916–918, 916, 917
- versus creating out-of-process COM servers, 921–923
- creating and running the test client, 919–920
- DllMain function, 900
- editing .cpp files, 919
- header files, 901
- implementing methods, 908–909
- implementing properties, 905–906
- in-process servers defined, **847**
- overview of Active Template Library, 896–897, 897
- overview of, 897–898, 898
- source files, 899–901
- testing ATL servers, 916–920
- viewing project files, 899, 899
- Visual Basic and, 907
- item lists in ADO, 696–699, 699
- memory-mapped files, 378–379
- MFC clients for testing in-process COM servers, 916–918, 916, 917
- multistream examples, 80
- named pipe instances, 251–253
- named pipes, 249–251
- out-of-process COM servers with Active Template Library (ATL), **896–897, 920–925**
 - adding CoClasses, properties, and methods, 923
 - versus creating in-process COM servers, 921–923
 - overview of Active Template Library, 896–897, 897
 - overview of, 920, 921
 - SecurityMgrExe.cpp file, 922–923
 - StdAfx.h file, 922
 - testing out-of-process servers, 923–925, 924
- parent and child pipes, 236–241
- power-aware applications, 98–99
- “power-friendly” applications, 101–102
- Publisher demo program, 618–623, 618, 623
- random keys, 437–440, 439
- record IDs, 667–668
- streams, 81–82, 82
- test clients for in-process COM servers, 919–920
- threads
 - and connecting to pipes, 263–265
 - with CreateThread function, 148–151, 152–153, 159

- MFC threads with CWinThread class, 151–153
 - unique interface identifiers, 839, 842
- CRecordset update transactions, **648–651**. *See also* Booklist demo program
 - AddNew function, 648, 650
 - appending records, 650
 - CancelUpdate function, 649
 - CanUpdate, CanAppend, and CanRestart functions, 650
 - checking the legality of operations, 650–651
 - CRecordset class, 648–649
 - Delete function, 649
 - Edit function, 649
 - editing records, 649
 - IsOpen, IsBOF, IsEOF, and IsDeleted functions, 651
 - locking records, 651
 - SetLockingMode function, 651
 - Update function, 649, 650
- critical section objects
 - critical section commands, 165
 - defined, **146**
 - processes and, 147
- cross-platform support with ActiveX Data Objects and OLE DB, 604–605, 604
- cross-system application development, 17–18
- Crypto API functions, **429–432**. *See also* encryption
 - CryptAcquireContext, 440
 - CryptDecrypt, 441
 - CryptEncrypt, 438–439
 - CryptEncryptMessage, 440
 - CryptExportKey, 438
 - CryptExportPublicKeyInfo, 440
 - CryptGetUserKey, 437–438
 - CryptImportKey, 440
 - Cryptographic Service Provider (CSP) functions, 430
 - encryption functions, 431
 - hashing functions, 431–432
 - installing runtime support for, 432–433
 - key functions, 430–431
 - overview of, 429
- Crypto demo program, **432–444**. *See also* encryption
 - and compromising security, 442–444
 - creating default Cryptographic Service Providers, 435, 435, 436
 - creating random keys, 437–440, 439
 - CryptAcquireContext function, 440
 - CryptDecrypt function, 441
 - CryptEncrypt function, 438–439
 - CryptEncryptMessage function, 440
 - CryptExportKey function, 438
 - CryptExportPublicKeyInfo function, 440
 - CryptGenKey function, 437–438
 - CryptGetUserKey function, 437–438

- CryptImportKey function, 440
 - decrypting files, 440–441, 441
 - encrypting files, 436–440
 - installing Crypto API runtime support, 432–433
 - project settings, 434–435, 434
 - ReportStatus function, 437
- CTipWnd::Create method, 124–125
- CTitleForm class, **628–632**. *See also* Publisher demo program
 - GetTableName function, 632
 - header file, 628–631
 - OnGetRecordset function, 629, 630
 - OnInitialUpdate function, 630, 632
 - setting sort order, 630–631
 - setting window captions, 631–632
- CTitleSet class, **623–628**. *See also* Publisher demo program
 - DoDataExchange function, 627–628
 - DoFieldExchange function, 625–626
 - GetDefaultConnect function, 626
 - GetDefaultConnectSQL function, 626–627
 - header file, 623–627
 - transferring data from databases to recordsets, 627–628
- CURRENCY data type, 940
- curve capabilities of devices, 462, 474
- custom interfaces
 - for COM servers, 880–884
 - defined, **836**
- custom lightweight handlers, 983, 983
- CWinThread class, 151–153
- CWnd::CreateEx method, 124

D

- DACLs. *See* discretionary (owner) access control lists
- data consumers and providers in OLE DB, 600, 682
- data movement SIMD instructions, 582–583
- data sources
 - closing connections, 696
 - defined, **682–683**
 - opening connections to, 693–696
- data structures. *See also* structures
 - defined, **57**
 - initializing in List demo program, 358–359
 - MSG (message structure), 39–41, 57
- data types
 - Automation data types, **939–943**
 - BSTR, 939–940
 - CURRENCY, 940
 - SAFEARRAY, 941
 - VARIANT, 942–943

- FAR, 39, 56
- HRESULT, 883, 893
- Hungarian notation conventions, 54–55
- listed, 56–57
- MSG (message structure) data type, 39–41, 57
- NEAR, 39, 57
- Registry data types, 274–275
- database tier technologies, 858
- database-lookup Winsock 2 API functions, 732–735
- databases, **610–612**. *See also* ActiveX Data Objects (ADO); Booklist demo program; Publisher demo program; Universal Data Access (UDA)
 - flat-file databases, 610
 - foreign key values, 611
 - keys, primary keys, and multi-value keys, 611
 - OLE DB, **600–602**, **680–681**
 - ActiveX Data Objects and, 602
 - advantages of, 601–602
 - availability of, 600
 - cross-platform support with, 604–605, 604
 - data consumers, data providers, and service components, 600, 682
 - defined, **600–601**, 601, **680–681**
 - Internet advantages of, 603
 - ODBC (Open Database Connectivity) and, 600–601, 680–681
 - Open Database Connectivity (ODBC)
 - OLE DB and, 600–601, 680–681
 - registering databases with, 615–618, 616, 617
 - organization of, 610–611
 - relational databases, 611
 - SQL (Structured Query Language), **612–615**
 - defined, **612**
 - GetDefaultConnectSQL function, 626–627
 - MFC (Microsoft Foundation Class) and, 612–613
 - restricting record selections, 613–614
 - sorting records, 614–615
 - SQL statements in Booklist program, 669–670
 - tracing SQL operations, 659–660
 - tables
 - defined, **611–612**, 611
 - GetTableName function, 632
 - populating data from another table, 668
- dates. *See also* time
 - date/time stamps, 73
 - time/date conversion functions in Internet API, 762
- DCOM (Distributed Component Object Model), 801–802
- DCs. *See* device contexts
- debuggers, 304
- decrypting files, 440–441, 441
- .DEF definition file, in “Hello, World” example, 45–48
- default text mapping mode, 479, 479

- Delete function, 649
- DeleteCriticalSection function, 165
- DeleteList function, 363
- deleting
 - records in ADO programming, 712–713
 - streams, 84
- demand paging, 335
- deriving interfaces from IUnknown interface, 831, 838–839
- designing COM interfaces, 837–843
 - for remote use, 838
- destroying
 - heap, 352–353
 - mutexes, semaphores, and events, 164–165
 - pipes, 230
- device contexts (DCs), **450–459**. *See also* Graphics Device Interface
 - accessing, 452–453
 - BeginPaint function, 452–453
 - CreateDC function, 453, 454
 - CreateIC function, 453, 454
 - DC demo program, **454–459, 466–477**
 - acquiring information device contexts, 454–459, 455
 - bitmap operations, 473
 - clipping capabilities, 473–474
 - curve capabilities, 474
 - defined, **466**
 - device dimension and resolution, 470–471
 - device palette (color) capabilities, 467–469, 467
 - driver version and device types, 469
 - GetDeviceCaps function, 466, 473
 - line capabilities, 475
 - polygon capabilities, 475–476
 - raster capabilities, 471–473
 - text capabilities, 476–477
 - defined, **450**
 - device-context handles, 450–451, 454
 - device-context handles in MFC, 451
 - EndPaint function, 452–453
 - GDI information indexes and flags, **459–477**
 - clipping capabilities, 464, 473–474
 - curve capabilities, 462, 474
 - defined, **459**
 - device color capabilities, 461, 467–469, 467
 - device dimensions, 460, 470–471
 - driver version and device types, 460, 469
 - graphics, image, and font-handling capabilities, 462–465
 - line capabilities, 463, 475
 - polygon capabilities, 463, 475–476
 - printer-related device capabilities, 462
 - raster capabilities, 464–465, 471–473

- text capabilities, 463–464, 476–477
- GetDC function, 452–454
- GetProfileString function, 454
- GetWindowsDC function, 453
- OnPaint function, 311, 451
- PAINTSTRUCT structure and, 452–453
- querying, 451–452
- ReleaseDC function, 453
- TextOut function, 451
- device drivers. *See* drivers
- device elements, 526
- dialog boxes
 - versus pop-up tip windows, 116
 - pop-up windows in, 132–135, 132, 133
- digital signatures, 427
- dimensions of devices, 460, 470–471
- directory parameters for CreateProcess function, 197–198
- DirectX technology, **492–498**. *See also* OpenGL technology
 - advantages of, 492–493
 - defined, **492**
 - Direct3D component, 493, 494
 - Direct3D Immediate mode, 493
 - Direct3D Retained mode, 494
 - DirectAnimation component, 494
 - DirectDraw component, 493
 - DirectInput component, 494
 - DirectMusic component, 493
 - DirectPlay component, 494
 - DirectSetup component, 494
 - DirectShow component, 494–495
 - DirectSound component, 493
 - DirectX Transform component, 495
 - DirectX version 7.0, 495–496
 - examples, 496–498, 497
 - versus Fahrenheit technology, 502
 - versus OpenGL technology, 498
 - Pentium III processors and, 495
 - SIMD extensions and, 577
- disconnect Internet API functions, 754–755
- DisconnectNamedPipe function, 227, 257
- discretionary (owner) access control lists (DACLS)
 - DACL functions, 412
 - in FileUser demo program, 420–423
 - overview of, 394, 401, 402, 404
- disk drives. *See* hard disk drives
- DispatchMessage function, 37, 38–39
- dispinterfaces in Visual C++, 934–935, 936
- Display form in Booklist program, 662–664, 663

- display windows, 40
- displaying
 - information in static controls in ShowWave demo program, 553–554
 - multiple applications on multiple monitor, 93
 - windows in “Hello, World” example, 36
- Distributed Component Object Model (DCOM), 801–802
- Distributed interNet Applications Architecture (DNA), 856–858
- DLL files
 - address space and, 339
 - creating
 - for COM servers, 888–893
 - for in-process COM servers, 909
 - .DEF definition files and, 46
 - DLL file protection, 16–17
 - DllCanUnloadNow function, 891–892, 895
 - DllGetClassObject function, 892, 895
 - DllMain function, 900
 - DllRegisterServer function, 892, 893
 - DllUnregisterServer function, 892
- DNA (Distributed interNet Applications Architecture), 856–858
- DoDataExchange function, 627–628
- DoFieldExchange function, 625–626
- DOS
 - DOS applications and NTFS file system, 64
 - DOS exception handling, 298
 - “Hello, World” example in DOS versus Windows 2000, 28–29, 30, 42, 46, 47
 - mapping modes, 478
 - Registry versus DOS configuration files, 270
- DoThread procedure, 174–177
- DoWrite, DoRead, and DoClear functions, 382–384
- drawing sound waves, 568–571, 568
- DrawProc procedure, 178–179
- DrawText function, 42–43
- DrawText method, 125
- drivers
 - Encrypting File System (EFS) driver, 96
 - version and device types information, 460, 469
 - Win32 driver model, 18
 - writing installable file system filter drivers, 95
- drives. *See* hard disk drives
- dual-boot systems, **12, 18–24**
 - Be operating system multiboot manager, 19
 - and dual-installing applications, 23–24
 - file systems and, 12
 - hardware boot management with removable hard drives, 19
 - LILO (Linux Loader) software, 20
 - network names and, 23
 - NT Loader (NTLDR) software, 19, 20–22
 - overview of, 18–19

- PowerQuest Boot Magic software, 19–20
- setting up, 21–22
- V Communications System Commander software, 20
- dual-interface components
 - defined, **836–837**, 837
 - in Visual C++, 936–937, 936
- duplex pipes, 191
- DuplicateHandle function
 - in process-related operations, 205, 237
 - in thread-related operations, 156, 164–165
- DVD drive support, 15
- dynamic load balancing, **870–871**
 - application clusters and, 871
 - COM+ load balancing, 870–871
 - component or load-balancing routers, 871
 - defined, **870**
 - Windows Load Balancing (WLB), 870–871
- dynamic priority of threads, 143–145, 144, 295

E

- early binding, 852–853
- Edit function, 649
- Edit operations in Booklist program, **662–675**
 - AddEntry function, 673
 - adding records, 666–670
 - canceling Edit or Add Record operations, 674–675
 - CancelUpdate function, 674
 - changing the Display form, 662–664, 663
 - customizing Add Entry form, 670–674, 672
 - direct SQL statements, 669–670
 - generating record IDs, 667–668
 - IsOnFirstRecord and IsOnLastRecord functions, 666
 - LoadDealerList function, 668–669
 - MoveFirst function, 675
 - NewTitleID function, 667–668, 671
 - OnCancel function, 674
 - OnEditEntry function, 674
 - OnEditTitle function, 662–663
 - OnInitialUpdate function, 668
 - OnNewTitle function, 670–671
 - populating data from another table, 668
 - Requery function, 664
 - ResetControls function, 663
 - resetting controls, 665–666
 - resetting edit fields, 664–665
 - Update function, 674
 - UpdateData function, 671–673, 675
- editing

- .cpp files, 919
- records in Booklist program, 649
- the Registry, 273
- EFS (Encrypting File System) driver, 96
- e-mail
 - SendMail demo program, 743–750, 744
 - SendMail2 demo program, 751
- EnableMenuItem function, 130
- encapsulation, 849–850
- encryption, **96, 392, 424–444**. *See also* security
 - adding encryption support to applications, **432–444**
 - and compromising security, 442–444
 - creating default Cryptographic Service Providers, 435, 435, 436
 - creating random keys, 437–440, 439
 - CryptAcquireContext function, 440
 - CryptDecrypt function, 441
 - CryptEncrypt function, 438–439
 - CryptEncryptMessage function, 440
 - CryptExportKey function, 438
 - CryptExportPublicKeyInfo function, 440
 - CryptGenKey function, 437–438
 - CryptGetUserKey function, 437–438
 - CryptImportKey function, 440
 - decrypting files, 440–441, 441
 - encrypting files, 436–440
 - installing Crypto API runtime support, 432–433
 - project settings, 434–435, 434
 - ReportStatus function, 437
- anti-virus software and, 429
- attacks, 443–444
- brute force attacks, 443
- Crypto API functions, **429–432**
 - CryptAcquireContext, 440
 - CryptDecrypt, 441
 - CryptEncrypt, 438–439
 - CryptEncryptMessage, 440
 - CryptExportKey, 438
 - CryptExportPublicKeyInfo, 440
 - CryptGetUserKey, 437–438
 - CryptImportKey, 440
 - Cryptographic Service Provider (CSP) functions, 430
 - encryption functions, 431
 - hashing functions, 431–432
 - installing runtime support for, 432–433
 - key functions, 430–431
 - overview of, 429
- Cryptographic Service Providers (CSPs)
 - creating default CSPs, 435, 435, 436
 - CSP functions, 430

- defined, **429**
- digital signatures and authentication, 427
- Encrypting File System (EFS) driver, 96
- exclusive-OR-ing (XORing), 442
- government regulation of, 425
- hashing
 - defined, **428–429**
 - hashing functions, 431–432
 - overview of, 424–425
 - public-key encryption, 427, 428
 - stream versus block encryption and byte padding, 428
 - symmetric-key encryption, 428
 - when to use, 426
 - in Windows 95/98 versus Windows 2000, 392
- EndPoint function, 43
- EnterCriticalSection function, 165
- environment parameters for CreateProcess function, 197–198
- error handling. *See also* exception handling; messages
 - error functions in Internet API, 762–763
 - versus exception handling, 299–300
 - in ShowWave demo program, 533, 543
- Errors collection in ADO, 685
- event messages. *See* messages
- event objects
 - CreateEvent function, 163
 - defined, **146–147**
 - OpenEvent function, 164–165
 - processes and, 147
 - ResetEvent function, 162–163, 258–259
 - SetEvent function, 162–163
 - sharing and destroying, 164–165
- event-driven programming, 38–39
- events, CreateEvent function, 163
- exception handling, **294–325**. *See also* error handling
 - C/C++ exception handling, 297
 - debuggers and, 304
 - DOS exception handling, 298
 - versus errors, 299–300
 - exception-handling macros, 304
 - exceptions defined, **295, 299–300**
 - Exceptions demo program, **309–324**
 - failed catch example, 316–318, 318
 - GetTextExtent function, 311
 - Invalidate function, 311
 - MFC exception classes, 312–316
 - nested exception handler example, 312–316, 316
 - OnPaint function, 311
 - overview of, 309–310

- resource exception example, 318–322, 321
- simple exception handler example, 310–312, 310
- user exception example, 322–324, 323
- filtering exceptions, 298
- frame-based exception handling, 301–302, 301
- GetErrorMessage function, 306
- interrupt dispatch tables (IDTs), 296
- interrupt objects, 296
- interrupt request levels (IRQLs), 295
- interrupt service routines (ISRs), 295–296
- MFC exception handling
 - defined, **297**
 - exception classes, 305–306, 312–316
- order of execution and, 302–304
- Plug-and-Play event messages, **206–209**
 - defined, **306–307**
 - how Plug-and-Play messages function, 307
 - listed, 308–309
- ReportError function, 306
- traps and the trap handler, 294–296
- Visual C++ exception handling, 297–298, 301–302, 301
- exclusive-OR-ing (XORing), 442
- .EXE files, address space and, 339
- executable files parameters for CreateProcess function, 195–196
- execution contexts, 897
- ExitProcess function, 206–207, 230, 258
- ExitThread function, 156–158, 159
- exported functions, 31
- extract SIMD instruction, 587

F

- Fahrenheit graphics technology, 502
- failed catch exception handler example, 316–318, 318
- failure recovery for queued components, 867–869
- FAR data type, 39, 56
- fast approximation SIMD operations, 580
- FAT file systems. *See also* file systems; NTFS file system
 - converting FAT16 to FAT32, 13
 - defined, **67**
 - FAT16 versus FAT32 versus NTFS file systems, 9–13, 62–63
 - filenames, 67
 - protected-mode FAT file system (VFAT), 68
 - security and, 395
- FetchRecords function, 696–697, 702–703
- FetchSingleTitle function, 701–703
- Fields collection in ADO, 685
- file attributes in NTFS file system, 68–69
- file systems. *See also* FAT file systems; NTFS file system

- dual-boot systems and, 12
- HPFS (High Performance File System), 62–63, 66
- long filename compatibility between, 78
- querying, 76–78, 76
- writing installable file system filter drivers, 95

File Transfer Protocol. *See* FTP

file-mapping objects, **339–340**, **369–385**. *See also* memory management

- Browser demo program, **377–385**

 - ChangeAccess function, 384–385

 - CreateFile function, 380

 - CreateFileMapping function, 379–380

 - creating and using mapped files, 378–379

 - defined, **377–378**, 378

 - DoWrite, DoRead, and DoClear functions, 382–384

 - editing shared objects, 383–385

 - FindWindow function, 380

 - initializing mapped files, 379–380

 - OpenFileMapping function, 380

 - running the dialog box, 380–382

- CreateFile function, 370, 380

- CreateFileMapping function, 370, 371–373, 379–380

- creating, **371–373**

- defined, **369**

- mapping views, 374

- MapViewOfFile function, 370, 374

- memory-mapped files defined, **339–340**, **369–370**, 371

- OpenFileMapping function, 375, 380

- preserving coherence, 375–377, 376

- sharing, 375–377, 376

- uses of copy-on-write protection, 377

filenames in NTFS file system

- converting to 8.3 format, 64

- guidelines for compatibility between file systems, 78

- NetWare and, 79

- overview of, 63–64

files

- .cpp files, 919

- .DEF definition file, in “Hello, World” example, 45–48

- DLL files

 - address space and, 339

 - creating for COM servers, 888–893

 - creating for in-process COM servers, 909

 - .DEF definition files and, 46

 - DLL file protection, 16–17

 - DllCanUnloadNow function, 891–892, 895

 - DllGetClassObject function, 892, 895

 - DllMain function, 900

 - DllRegisterServer function, 892, 893

 - DllUnregisterServer function, 892

- .EXE files and address space, 339
- file locating, reading, and writing functions in Internet API, 758–759
- file operations using multimedia I/O functions, 519–520
- header files
 - in creating in-process COM servers, 901
 - for CTitleForm class, 628–631
 - for CTitleSet class, 623–627
 - for ShowWave demo program, 523
 - Template.H header file, 49, 51
 - Windows.H header file in “Hello, World” example, 29–30
 - Windows.H header file in Template example, 49
- HOSTS and SERVICES files, 733–735
- .ICO files, 53
- IDL files, 876–879
- .INI files, 270
- linking
 - hard links, 84–86
 - in NTFS file system, 84–86
- Mathlib.ACF marshalling interface handle file, 810–811
- Mathlib.CPP function library, 806
- Mathlib.H header file, 806–808, 811–812
- Mathlib.IDL function interface definition file, 808–810
- Mathlib_c.C and Mathlib_s.C files, 811–812
- querying file information, **69–78**
 - NTFS_Ops demo program, 69–76, 70
 - querying file systems, 76–78, 76
- retrieving filenames in ShowWave demo program, 555–556
- RIFF (Resource Interchange File Format), 509, 517–518, 518
- SecurityMgrExe.cpp file, 922–923
- sparse file support, 95–96
- StdAfx.h file, 922
- Template.H header file, 49, 51
- .TLB files, 333, 943
- Windows.H header file
 - in “Hello, World” example, 29–30
 - in Template example, 49
- FileUser demo program, **418–424**. *See also* security, SDs (security descriptors)
 - adding access control entries, 421–423
 - Check function, 419
 - defined, **418**
 - discretionary (owner) access control list operations, 420–423
 - GetAclInformation function, 420–421
 - GetFileSecurity function, 419–420
 - GetSecurityDescriptorDacl function, 420–421
 - IsValidAcl function, 421
 - LookupAccountName function, 419
 - retrieving security descriptors, 419–420
 - retrieving security identifiers, 418–419

- SetFileSecurity function, 423–424
- setting security descriptors, 423–424
- FillMemory command, 356
- filtering exceptions, 298
- FindOneOf function, 768
- FindWindow function, 380
- flags
 - in NTFS_Ops demo program, 74–76, 77–78
 - in Windows 2000, 74
- flat-file databases, 610
- floating menus, 128–132, 129
- floating-point instructions, **578–585**. *See also* SIMD
 - arithmetic instructions, 579–580
 - broadcast shuffle operation, 581
 - compare and set EFLAGS instructions, 584–585
 - compare and set mask instructions, 583
 - conversion operation instructions, 585
 - data movement instructions, 582–583
 - defined, **578–579**, 578
 - fast approximation operations, 580
 - logical instructions, 584
 - min/max comparisons, 580
 - move mask from floating point to integer instruction, 583
 - packed operations, 578
 - rotate shuffle operation, 581
 - scalar operations, 579
 - shuffle instruction, 580–581
 - square root instructions, 580
 - swap shuffle operation, 581
 - unpack instruction, 581–582
- floppy disks, NTFS file system and, 63, 65
- FlushFileBuffers function, 227
- font-handling capabilities of devices, 462–465
- foreground switching, 109–110
- FormatMessage function, 218, 289–290
- Forms+ technology, 856, 857
- foundation classes. *See* MFC (Microsoft Foundation Class)
- frame-based exception handling, 301–302, 301
- free pages, 333
- freeing memory, 345–346, 354
- FTP (File Transfer Protocol)
 - defined, **728**
 - FTP functions in Internet API, 759–760
 - FtpGet demo program, 764–774, 765
 - FtpGetFile function, 760, 764, 771–773
 - FtpOpenFile function, 760, 774
- functions. *See also* APIs; methods
 - AddEntry, 673
 - AddNew, 648, 650

- AlphaBlend, 108
- AssignProcessToJobObject, 105
- BeginPaint, 452–453
- BeginTrans, 652–653
- CancelUpdate, 649, 674
- CanUpdate, CanAppend, and CanRestart, 650
- CFile::Open, 83
- ChangeAccess, 384–385
- Check, 419
- CheckMenuItem and CheckMenuRadioItem, 131
- CheckOrders, 645–646
- Close, 699
- CNTFS_StreamDlg::OnRead, 83–84
- CoCreateInstance
 - Asynchronous COM and, 962
 - creating COM servers, 885, 893, 895
 - creating in-process COM servers, 918
 - IDispatch interface Automation and, 947
- CoInitialize, 919, 929, 948, 962
- CoInitializeEx, 929
- CommitMemFilter, 360–361
- CommitTrans, 652–653
- CoUninitialize, 893, 919
- CoWaitForMultipleHandles API function, 954–955
- CreateDC, 453, 454
- CreateDSObject, 1023
- CreateEx, 124
- CreateFile
 - in file-mapping operations, 370, 380
 - in pipe-related operations, 218–219, 225, 229, 263–264
 - security and, 398–399
- CreateFileMapping, 370, 371–373, 379–380
- CreateGroup, 1021–1024, 1024
- CreateHardLink, 85–86
- CreateIC, 453, 454
- CreateInstance, 694, 885–887
- CreateJobObject, 105
- CreateList, 358–359
- CreateNamedPipe, **209–218, 251–253**
 - access, write-through, and overlapping parameters, 211–212
 - buffer sizes, 215
 - defined, **209–210**
 - in NamePipe demo program, 251–253
 - naming pipes, 210–211
 - overview of parameters, 216–217
 - pipe instances, 214–215
 - return values, 217–218
 - security attributes parameter, 216

- time-out period parameter, 216
- type, read mode, and wait parameter, 212–214
- CreateProcess, **194–201, 203, 204–205**
 - in AnonPipe demo program, 237–241, 243, 245
 - assigning priority classes, 203
 - blocking inheritance, 196–197
 - C runtime library equivalents, 201
 - defined, **194**
 - environment and directory parameters, 197–198
 - executable files and arguments parameters, 195–196
 - I/O handles, 200
 - letting children inherit handles, 204–205
 - in NamePipe demo program, 253–255, 262
 - parameters pointing to structures, 198–200
 - process type and priority parameters, 197
 - return values, 201
 - security attributes, 196
- CreateWindow, 34–36, 339
- CreateWindows, 167, 168–169
- Crypto API functions, **429–432**
 - CryptAcquireContext, 440
 - CryptDecrypt, 441
 - CryptEncrypt, 438–439
 - CryptEncryptMessage, 440
 - CryptExportKey, 438
 - CryptExportPublicKeyInfo, 440
 - CryptGetUserKey, 437–438
 - CryptImportKey, 440
 - Cryptographic Service Provider (CSP) functions, 430
 - encryption functions, 431
 - hashing functions, 431–432
 - installing runtime support for, 432–433
 - key functions, 430–431
 - overview of, 429
- Delete, 649
- DeleteCriticalSection, 165
- DeleteList, 363
- DispatchMessage, 37, 38–39
- DllCanUnloadNow, 891–892, 895
- DllGetClassObject, 892, 895
- DllMain, 900
- DllRegisterServer, 892, 893
- DllUnregisterServer, 892
- DoDataExchange, 627–628
- DoFieldExchange, 625–626
- DoWrite, DoRead, and DoClear, 382–384
- DrawText, 42–43
- Edit, 649
- EnableMenuItem, 130

- EndPaint, 43
- EnterCriticalSection, 165
- exported functions, 31
- FetchRecords, 696–697, 702–703
- FetchSingleTitle, 701–703
- FindOneOf, 768
- FindWindow, 380
- FormatMessage, 218, 289–290
- GetCollect, 698
- GetCursorCommitBehavior and GetCursorRollbackBehavior, 653
- GetDC, 452–454
- GetDefaultConnect, 626
- GetDefaultConnectSQL, 626–627
- GetDeviceCaps, 466, 473
- GetDocument, 641
- GetErrorMessage, 306
- GetFileAttributes, 71–72, 75
- GetGuiResources, 108
- GetLastError, 218, 229, 290, 762, 766
- GetMapMode, 481–483
- GetMessage, 36–38
- GetProfileString, 454
- GetSecurityDescriptorDacl, 420–421
- GetStatus, 71, 72–73
- GetStdHandle, 238, 245, 263
- GetSystemPowerStatus, 99–101, 101
- GetTableName, 632
- GetTextExtent, 127–128, 311
- GetVolumeInformation, 76–77, 82–83
- GetWindowsDC, 453
- global and local memory functions, **353–355**
 - overview of, 353–355
 - system limits and allocating memory, 355
- heap functions, **349–353**
 - HeapAlloc, 351, 355
 - HeapCreate, 350–352
 - HeapDestroy, 353
 - HeapFree, 351, 352–353
 - HeapRealloc, 352
 - HeapSize, 352
 - overview of, 349–350
- InitializeCriticalSection, 165
- InitializeListBox, 283–284
- InterlockedIncrement and InterlockedDecrement, 931
- Internet API functions, **752–774**
 - AutoDial functions, 755
 - connect and disconnect functions, 754–755
 - cookie functions, 761

- defined, 752–753
- error and status functions, 762–763
- file locating, reading, and writing functions, 758–759
- FindOneOf, 768
- FTP, Gopher, and HTTP functions, 759–760
- FtpGet demo program, 764–774, 765
- FtpGetFile, 760, 764, 771–773
- FtpOpenFile, 760, 774
- GetLastError, 762, 766
- Internet applications, 774
- InternetAttemptConnect, 754, 756
- InternetAutodial, 755, 757–758
- InternetConnect, 755, 764, 769–770
- InternetGetConnectedState, 755, 756–757, 758
- InternetGetLastResponseInfo, 762, 766
- InternetOpen, 755, 764, 768
- InternetOpenUrl, 762, 774
- InternetReadFile, 759, 774
- LastErrorMsg, 766
- limitations of, 753
- MFC wrapper classes, 763–764
- OnOK, 767
- opening dial-up connections, 756–758
- time/date conversion functions, 762
- URL-related functions, 761–762
- Invalidate, 311
- IsBad memory validation functions, 355–356
- IsOnFirstRecord and IsOnLastRecord, 666
- IsOpen, IsBOF, IsEOF, and IsDeleted, 651
- job object functions, 105–107
- LAN Manager API functions, 792
- LastErrorMsg, 766
- LeaveCriticalSection, 165
- LoadDealerList, 668–669
- LockServer, 885–887
- LockWorkStation, 108
- LookupAccountName, 419
- mailslot functions, 799–801
- Main_OnCommand, 173–174
- Main_OnCreate, 170–172, 177
- Main_OnInitMenu, 173
- Main_OnSize, 169–170, 172–173
- Main_OnTimer, 169–170, 173
- memory validation functions, 355–356
- MessageBeep, 511–512
- message-mapped functions, 44–45
- MoveFirst, 675
- MoveNext, 698
- named pipe functions, 799–801

- NetBIOS functions, 784–787
- NewTitleID, 667–668, 671
- OnActiveView, 645
- OnCancel, 674
- OnClose, 696
- OnDbclbResultList, 699–700
- OnEditEntry, 674
- OnEditTitle, 662–663
- OnGetRecordset, 629, 630
- OnInitDialog, 693–694, 697, 701
- OnInitialUpdate
 - in Booklist program, 668
 - for CTitleForm class, 630
 - in Popups program, 118
 - in Publisher program, 640, 641, 642
- OnKillfocusX, 133–135
- OnLButtonDown and OnRButtonDown, 119
- OnNewTitle, 670–671
- OnOK, 767
- OnPaint, 311, 451
- OnShowOrders, 644
- OnSize, 118–119
- Open, 695
- OpenFileMapping, 375, 380
- OpenJobObject, 105
- ParseCommandLine, 1018–1019
- pipe-related functions, **208–231**
 - asynchronous I/O parameter, 224
 - blocking parameter, 224
 - CallNamedPipe, 228, 229
 - CloseHandle, 227, 229, 230, 243
 - ConnectNamedPipe, 225–226, 227, 257
 - CreateFile, 218–219, 225, 229, 263–264
 - CreateNamedPipe, 209–218, 251–253
 - creating anonymous pipes with CreatePipe, 208, 237
 - for destroying pipes, 230
 - DisconnectNamedPipe, 227, 257
 - ExitProcess, 206–207, 230, 258
 - FlushFileBuffers, 227
 - GetLastError, 218, 229
 - GetNamedPipeHandleState, 220–221
 - GetNamedPipeInfo, 221–222
 - ImpersonateNamedPipeClient, 229–230
 - PeekNamedPipe, 222–223
 - ReadFile, 218, 223–224, 229, 247, 265
 - RevertToSelf, 229–230
 - SetNamedPipeHandleState, 219–220
 - for synchronizing connections, 225–227

- TransactionNamedPipe, 228–229
- WaitNamedPipe, 225, 226–227, 229, 261, 264
- WriteFile, 218, 223–224, 229, 242, 261
- PlaySound, 513–515
- PopupTips, 120–122, 120
- PostQuitMessage, 43–44
- ProcessData, 962–963, 969
- process-related functions, **194–231**
 - CreateProcess, 194–201, 203, 204–205
 - DuplicateHandle, 205, 237
 - ExitProcess, 206–207, 230, 258
 - GetCommandLine, 202, 263
 - GetCurrentProcess and GetCurrentProcessId, 202
 - GetEnvironmentVariable, 202
 - GetExitCodeProcess, 206
 - GetPriorityClass, 203
 - OpenProcess, 205–206
 - for sharing handles, 204–206
 - StartProcess, 237, 238–241
 - for synchronizing processes, 203–204
 - TerminateProcess, 207
 - WaitForInputIdle, 204
 - WaitForSingleObject and WaitForMultiple-Objects, 203, 258–259
- PutCollect, 707
- QueryInformationJobObject, 106
- RegisterClass, 32–34
- RegisterWaitForSingleObject, 107
- Registry API functions
 - defined and listed, **276–278**
 - RegCreateKey, 280–281, 285, 287
 - RegCreateKeyEx, 288
 - RegEnumKey, 284–285
 - RegOpenKeyEx, 288
 - RegQueryInfoKey, 282–283
 - RegQueryValueEx, 274, 285
 - RegSetValueEx, 274, 287–289
- ReleaseDC, 453
- ReportError, 306
- ReportStatus, 437
- Requery, 664
- ResetControls, 663
- RFX_**** function calls, 648, 655–659, 656
- Rollback, 652–653
- security API functions, **391–392, 407–417**
 - access checking functions, 413–414
 - access control entry (ACE) functions, 412–413
 - access control list (ACL) functions, 412
 - access token functions, 408–409
 - auditing functions, 417

- client impersonation functions, 409–410
- converting self-relative SDs to absolute SDs, 403
- discretionary (owner) access control list (DACL) functions, 412
- GetAclInformation, 420–421
- GetSecurityDescriptorDacl, 420–421
- IsValidAcl, 421
- local service authority (LSA) functions, 415–416
- overview of, 407–408
- privilege functions, 415
- security descriptor (SD) functions, 401, 410
- security identifier (SID) functions, 411
- Set/Get security information functions, 416
- SetFileSecurity, 423–424
- system access control list (SACL) functions, 412
- window station functions, 417
- in Windows 95/98 versus Windows 2000, 391–392
- SelectForm, 644, 645, 646
- SendCommand, 241–243, 260–262
- SendInput, 108
- SetBkColor, 125
- SetInformationJobObject, 106
- SetLockingMode, 651
- SetMapMode, 481–483
- SetThreadExecutionState, 102–104
- ShowWindow, 36
- sndPlaySound, 512–513
- SysAllocString, 1019
- SystemParametersInfo, 109–111
- TerminateJobObject, 106
- TextOut, 451
- thread-related functions, **148–165**
 - for acquiring mutexes and semaphores, 161–162
 - AfxBeginThread, 152
 - C runtime library equivalents, 158–159
 - CloseHandle, 148, 150–151, 156, 157
 - CreateEvent, 163
 - CreateMutex and CreateSemaphore, 161, 162
 - creating MFC threads with CWinThread class, 151–153
 - creating threads with CreateThread function, 148–151, 152–153, 159
 - for critical sections, 165
 - DuplicateHandle, 156, 164–165
 - ExitThread, 156–158, 159
 - GetCurrentThread and GetCurrentThreadId, 155–156
 - GetExitCodeThread, 157
 - OnKillThreadsSlow and OnKillThreadsFast methods, 181–182
 - OpenMutex, OpenSemaphore, and OpenEvent, 164–165
 - PulseEvent, 163, 255
 - ReleaseMutex and ReleaseSemaphore, 162

- ResetEvent, 162–163, 258–259
- ResumeThread, 150, 154–155
- SetEvent, 162–163
- SetThreadPriority, 153–154, 168
- for sharing and destroying mutexes, semaphores, and events, 164–165
- Sleep and SleepEx, 154–155
- StartThread, 177–178
- SuspendThread, 154–155
- for synchronizing threads, 159–165
- TerminateThread, 158
- WaitForSingleObject and WaitForMultipleObjects, 159–162
- timeGetDevCaps, 509
- TrackMouseEvent, 108
- TrackPopupMenu, 131–132
- TranslateMessage, 37, 38–39
- UnregisterWaitEx, 107
- Update, 649, 650, 674
- UpdateData, 671–673, 675, 699, 703
- UpdateEntry, 704–705
- UserHandleGrantAccess, 106
- virtual memory functions, **340–349, 355**
 - GetWriteWatch, 342
 - overview of, 340–341
 - ResetWriteWatch, 342
 - system limits and allocating memory, 355
 - VirtualAlloc, 341–345, 355, 359, 366
 - VirtualFree, 345–346
 - VirtualLock, 348–349
 - VirtualProtect, 346–347, 364–366
 - VirtualQuery, 343, 347–348
 - VirtualUnlock, 348–349
- Winsock 2 API functions, **730–752**
 - asynchronous functions, 735–738
 - database-lookup functions, 732–735
 - defined, **730**
 - FindAddr demo program, 738–743, 739, 740
 - gethostbyaddr, 732, 742
 - gethostbyname, 732, 741–742, 745–746, 747
 - HOSTS and SERVICES files, 733–735
 - network byte order and host byte order conversion functions, 731–732
 - SendMail demo program, 743–750, 744
 - SendMail2 demo program, 751
 - socket functions, 730–731
 - Winsock applications, 751–752
- Winsock 2 functions for network programming, **793–799**
 - AppleTalk protocol and Winsock 2, 794–799, 798
 - defined, **793**
 - MacSock demo program, 794–799, 798
 - new features, 793–794

WNet API functions, **787–791**
defined, **787–788**
network resources and, 788–789
WNetDemo program, 789–791, 790

G

GDI. *See* Graphics Device Interface
generic access rights, 407
Get/Set security information functions, 416
GetAclInformation function, 420–421
GetClientRect procedure, 42
GetCollect function, 698
GetCommandLine function, 202, 263
GetCurrentProcess and GetCurrentProcessId functions, 202
GetCurrentThread and GetCurrentThreadId functions, 155–156
GetCursorCommitBehavior and GetCursorRollbackBehavior functions, 653
GetDC function, 452–454
GetDefaultConnect function, 626
GetDefaultConnectSQL function, 626–627
GetDeviceCaps function, 466, 473
GetDocument function, 641
GetEnvironmentVariable function, 202
GetErrorMessage function, 306
GetExitCodeProcess function, 206
GetExitCodeThread function, 157
GetFileAttributes function, 71–72, 75
GetGuiResources function, 108
gethostbyaddr Winsock 2 API function, 732, 742
gethostbyname Winsock 2 API function, 732, 741–742, 745–746, 747
GetIdsOfNames method, 933, 947
GetLastError function, 218, 229, 290, 762, 766
GetMapMode function, 481–483
GetMessage function, 36–38
GetNamedPipeHandleState function, 220–221
GetNamedPipeInfo function, 221–222
GetPriorityClass function, 203
GetProfileString function, 454
GetSecurityDescriptorDacl function, 420–421
GetStatus function, 71, 72–73
GetStdHandle function, 238, 245, 263
GetSystemPowerStatus function, 99–101, 101
GetTableName function, 632
GetTextExtent function, 127–128, 311
GetVolumeInformation function, 76–77, 82–83
GetWindowsDC function, 453
GetWriteWatch function, 342
global and local memory functions, **353–355**
overview of, 353–355

- system limits and allocating memory, 355
- globally unique identifiers. *See* GUIDs
- Gopher functions in Internet API, 759–760
- government regulation of encryption, 425
- graphics, **492–503**
 - DirectX technology, **492–498**
 - advantages of, 492–493
 - defined, **492**
 - Direct3D component, 493, 494
 - Direct3D Immediate mode, 493
 - Direct3D Retained mode, 494
 - DirectAnimation component, 494
 - DirectDraw component, 493
 - DirectInput component, 494
 - DirectMusic component, 493
 - DirectPlay component, 494
 - DirectSetup component, 494
 - DirectShow component, 494–495
 - DirectSound component, 493
 - DirectX Transform component, 495
 - DirectX version 7.0, 495–496
 - examples, 496–498, 497
 - versus Fahrenheit, 502
 - versus OpenGL, 498
 - Pentium III processors and, 495
 - SIMD extensions and, 577
 - drawing sound waves, 568–571, 568
 - Fahrenheit technology, **502**
 - Graphics Utilities on book's CD-ROM, 450
 - OpenGL technology, **498–502**
 - advantages of, 499–501
 - defined, **499**
 - versus DirectX, 498
 - disadvantages of, 501
 - versus Fahrenheit, 502
- Graphics Device Interface (GDI), **107–111, 395, 450–489**
 - active window tracking, 109
 - AlphaBlend function, 108
 - common controls support, 108
 - DC demo program, **454–459, 466–477**
 - acquiring information device contexts, 454–459, 455
 - bitmap operations, 473
 - clipping capabilities, 473–474
 - curve capabilities, 474
 - defined, **466**
 - device dimension and resolution, 470–471
 - device palette (color) capabilities, 467–469, 467
 - driver version and device types, 469
 - GetDeviceCaps function, 466, 473

- line capabilities, 475
- polygon capabilities, 475–476
- raster capabilities, 471–473
- text capabilities, 476–477
- device contexts (DCs), **450–459**
 - accessing, 452–453
 - acquiring information device contexts, 454–459, 455
 - BeginPaint function, 452–453
 - CreateDC function, 453, 454
 - CreateIC function, 453, 454
 - defined, **450**
 - device-context handles, 450–451, 454
 - device-context handles in MFC, 451
 - EndPaint function, 452–453
 - GetDC function, 452–454
 - GetProfileString function, 454
 - GetWindowsDC function, 453
 - OnPaint function, 451
 - PAINTSTRUCT structure and, 452–453
 - querying, 451–452
 - ReleaseDC function, 453
 - TextOut function, 451
- foreground switching, 109–110
- GDI information indexes and flags, **459–477**
 - clipping capabilities, 464, 473–474
 - curve capabilities, 462, 474
 - defined, **459**
 - device color capabilities, 461, 467–469, 467
 - device dimensions, 460, 470–471
 - driver version and device types, 460, 469
 - graphics, image, and font-handling capabilities, 462–465
 - line capabilities, 463, 475
 - polygon capabilities, 463, 475–476
 - printer-related device capabilities, 462
 - raster capabilities, 464–465, 471–473
 - text capabilities, 463–464, 476–477
- GDI-defined security objects, 395
- GetGuiResources function, 108
- GUI appearance settings, 110
- handicapped preference settings, 109
- HTML resources, 108
- LockWorkStation function, 108
- mapping modes, **478–489**
 - Cartesian coordinate modes, 479–480, 480
 - default text mode, 479, 479
 - in DOS versus Windows, 478
 - GetMapMode and SetMapMode functions, 481–483
 - isotropic and anisotropic modes, 480–481, 481

- Modes demo program, 487–489, 488
 - setting window and viewport extents, 486–487
 - setting window and viewport origins, 485–486
 - viewport versus window coordinate translations, 483–487
- message-only windows, 107
- mouse operation settings, 110
- power settings, 111
- screen saver status settings, 110
- SendInput function, 108
- SystemParametersInfo function, 109–111
- task switching settings, 110
- TrackMouseEvent function, 108
- GraphWin module of ShowWave program, **522, 566–572**
 - defined, **522**
 - drawing sound waves, 568–571, 568
 - registering window classes, 566–568
- GUI. *See* Graphics Device Interface
- GUIDs (globally unique identifiers)
 - Component Object Model (COM) and, 826
 - components and, 828–829
 - creating, 839, 842
 - defined, **802–803, 826**
 - interfaces and, 826, 833, 839, 842
 - Remote Procedure Calls and, 802–803

H

- handicapped preference settings, 109
- handles
 - CloseHandle function, in thread-related operations, 148, 150–151, 156, 157
 - DuplicateHandle function
 - in process-related operations, 205, 237
 - in thread-related operations, 156, 164–165
 - GetStdHandle function, 238, 245, 263
 - handle identifiers, 57–58
 - I/O handles in CreateProcess function, 200
 - inheriting pipe handles and creating threads, 244–245
 - for processes, 148
 - process-related functions for sharing handles, 204–206
 - pseudohandles, 155–156
 - for thread objects, 142–143, 148
- hard disk drives
 - Partition Magic software, 13
 - querying drive information, 69–76, 70
 - removable hard drives
 - dual-boot management with, 19
 - NTFS file system and, 63, 65
 - volumes
 - GetVolumeInformation function, 76–77, 82–83

- volume mount points, 97
 - volume quotas, 96–97
- Windows 2000 requirements for, 8–9
- hard links, 84–86
- hardware, **7–9**
 - AGP (Accelerated Graphics Port)
 - AGP video cards and multiple monitor support, 90
 - support for, 15
 - DVD drive support, 15
 - hard disk drive requirements, 8–9
 - hardware compatibility, 6–7, 8
 - hardware compatibility list, 7
 - IDE hard drives, 8
 - memory requirements, 7
 - MMX processor support, 15
 - Multilink Channel Aggregation (MCA) technology, 15
 - multiple monitor support, 15
 - Plug-and-Play feature, 14
 - processor requirements, 7
 - SCSI hard drives, 8
 - USB (Universal Serial Bus), 14–15
 - Win32 driver model, 18
 - WinModem card support, 8
- hashing. *See also* encryption
 - defined, **428–429**
 - hashing functions, 431–432
- HCL (Hardware Compatibility List), 7
- header files
 - in creating in-process COM servers, 901
 - for CTitleForm class, 628–631
 - for CTitleSet class, 623–627
 - for ShowWave demo program, 523
 - Template.H header file, 49, 51
 - Windows.H header file
 - in “Hello, World” example, 29–30
 - in Template example, 49
- heap. *See also* memory management
 - address space and, 339
 - defined, **349**
 - heap functions, **349–353**
 - HeapAlloc, 351, 355
 - HeapCreate, 350–352
 - HeapDestroy, 353
 - HeapFree, 351, 352–353
 - HeapRealloc, 352
 - HeapSize, 352
 - overview of, 349–350
- “Hello, World” example, **28–48**. *See also* Template example
 - .DEF definition file, 45–48

- in DOS versus Windows 2000, 28–29, 30, 42, 46, 47
- exported functions, 31
- message-driven or event-driven programming, 38–39
- message-mapped functions, 44–45
- MFC foundation classes and, 30, 44
- MSG (message structure) data type, 39–41, 57
- Windows.H header file, 29–30
- WinMain procedure, **30–38**
 - arguments, 30–32
 - creating application windows, 34–36
 - displaying windows, 36
 - message handling, 36–38
 - Pascal ordering, 30–31, 56
 - registering window classes, 32–34
- WndProc procedure, 41–44
- High Performance File System (HPFS), 62–63, 66
- high-level multimedia command sets, 508–509
- host byte order conversion functions, 731–732
- HOSTS files, 733–735
- HPFS (High Performance File System), 62–63, 66
- HRESULT type return value, 839–841, 883, 893
- HTML resources, 108
- HTTP (Hypertext Transfer Protocol)
 - defined, **728**
 - HTTP functions in Internet API, 759–760
- Hungarian notation conventions, 54–55

I

- I/O handles in CreateProcess function, 200
- IADs interface
 - defined, **996–997**
 - IADs interface methods, 997–999
 - IADsUser methods, 1012–1016, *1015*
 - implementing, 997
- .ICO files, 53
- IDE hard drives, 8
- IDispatch interface, **834–836, 931–949**. *See also* COM servers; interfaces in Component Object Model
 - Automation with Active Template Library, 937–939
 - Automation data types, 939–943
 - BSTR data type, 939–940
 - C++ Automation clients, 945–948
 - CoCreateInstance function, 947
 - CoInitialize function, 948
 - CURRENCY data type, 940
 - defined, **834–836, 932**
 - dispinterfaces in Visual C++, 934–935, 936
 - dual-interface components in Visual C++, 936–937, 936
 - GetIdsOfNames method, 933, 947

- implementing with Visual C++, 932–937
- IsUserAllowed method, 947
- overview of, 931–932
- SAFEARRAY data type, 941
- type libraries and, 943–945, 944, 945
- VARIANT data type, 942–943
- Visual Basic Automation clients, 948–949
- IDL files, 876–879
- IDTs (interrupt dispatch tables), 296
- IFS (Installable File System) Kit, 95
- IIDs (interface IDs), 839, 842
- image activation tables, 339
- image capabilities of devices, 462–465
- impersonation
 - client impersonation functions, 409–410
 - defined, **405–406**
 - ImpersonateNamedPipeClient function, 229–230
- implementation rules for COM components, 843–845
- implementing
 - IADs interface, 997
 - methods in in-process COM servers, 908–909
 - properties in in-process COM servers, 905–906
- inbound pipes, 191
- incoherence in file-mapping, 375–377, 376
- inheritance. *See also* processes
 - blocking, 196–197
 - defined, **187–189**
 - inheriting pipe handles and creating threads, 244–245
 - letting children inherit handles in CreateProcess function, 204–205
 - in object-oriented programming
 - defined, **853–855**
 - inheritance using early binding, 852–853
- .INI files, 270
- InitApplication and InitInstance procedures, 50
- initial threads, 138
- initialization procedures in Threads example, 166–169
- InitializeApp procedure, 168
- InitializeCriticalSection function, 165
- InitializeListBox function, 283–284
- initializing
 - COrderForm class, 640–642
 - data structures in List demo program, 358–359
 - memory-mapped files, 379–380
 - the parent process in AnonPipe demo program, 233
 - Web browsers, 777–780, 778
- in-process COM servers, **847, 896–920**
 - adding a CoClass, 901–902, 902, 903, 910–914
 - adding methods, 906–908, 907, 915–916
 - adding properties, 903–904, 903, 904, 914

- CCoComClass, CComObjectRoot classes and, 896–897, 897
- CoCreateInstance function, 918
- CoInitialize function, 919
- CoUninitialize function, 919
- creating DLL files, 909
- creating MFC clients, 916–918, 916, 917
- versus creating out-of-process COM servers, 921–923
- creating and running the test client, 919–920
- DllMain function, 900
- editing .cpp files, 919
- header files, 901
- implementing methods, 908–909
- implementing properties, 905–906
- in-process servers defined, **847**
- overview of Active Template Library, 896–897, 897
- overview of, 897–898, 898
- source files, 899–901
- testing ATL servers, 916–920
- viewing project files, 899, 899
- Visual Basic and, 907
- insert SIMD instruction, 587
- installable file system filter drivers, 95
- Installable File System (IFS) Kit, 95
- installing
 - Active Directory Service Interfaces (ADSI) SDK, 992
 - Crypto API support, 432–433
 - dual-installing applications, 23–24
- instances of pipes, 214–215, 251–253
- integer instructions, **585–589**. *See also* SIMD
 - broadcast shuffle operation, 588–589
 - defined, **585–587**, 586
 - extract instruction, 587
 - insert instruction, 587
 - min/max instruction, 587–588
 - move byte mask to integer instruction, 588
 - multiply high unsigned instruction, 588
 - packed byte, packed word, packed doubleword, and quadword integers, 586, 586
 - rotate shuffle operation, 589
 - shuffle instruction, 588–589
 - swap shuffle operation, 589
- Intel Pentium III processors. *See* SIMD
- interfaces in Component Object Model, **825–827**, **837–846**. *See also* Graphics Device Interface (GDI); user interface
 - activating and calling, 845–846
 - asynchronous interface construction, 958–960
 - asynchronous pipe interfaces, 977–978
 - characteristics of, 833
 - versus classes, 833
 - for COM servers, 876

- COM synchronization interfaces, 955–957
- components and, 830, 832
- creating unique interface identifiers, 839, 842
- custom interfaces, 836, 880–884
- defined, **825–827, 831–833, 831**
- deriving interfaces from IUnknown interface, 831, 838–839
- designing, 837–843
- designing for remote use, 838
- dual-interface components, 836–837, 837
- GUIDs (globally unique identifiers) and, 826, 833, 839, 842
- how interfaces work, 831–832, 831
- IDispatch interface, **834–836, 931–949**
 - Automation with Active Template Library, 937–939
 - Automation data types, 939–943
 - BSTR data type, 939–940
 - C++ Automation clients, 945–948
 - CoCreateInstance function, 947
 - CoInitialize function, 948
 - CURRENCY data type, 940
 - defined, **834–836, 932**
 - dispinterfaces in Visual C++, 934–935, 936
 - dual-interface components in Visual C++, 936–937, 936
 - GetIdsOfNames method, 933, 947
 - implementing with Visual C++, 932–937
 - IsUserAllowed method, 947
 - overview of, 931–932
 - SAFEARRAY data type, 941
 - type libraries and, 943–945, 944, 945
 - VARIANT data type, 942–943
 - Visual Basic Automation clients, 948–949
- immutability of, 832, 833, 839
- interface IDs (IIDs), 839, 842
- interface inheritance, 853–855
- interface inheritance using early binding, 852–853
- interface pointers, 831
- interface rules, 837–841
- interface semantics, 843
- interface signatures or syntax, 842–843
- IObjectControl interface, 872
- ISynchronize interface, 955–956
- ISynchronizeContainer interface, 956
- ISynchronizeHandle interface, 956–957
- IUnknown interface
 - and creating COM servers, 881–884
 - defined, **834**
 - deriving interfaces from, 831, 838–839
 - implement IUnknown rule, 844
- MIDL (Microsoft Interface Definition Language) and, 836, 876

- versus objects, 833
- persistence of, 829
- planning, 829, 832
- querying for object interfaces, 851–852
- returning HRESULT type values, 839–841, 883, 893
- string parameters as Unicode strings, 841
- type libraries and, 837
- virtual function tables, vtables, or VTBLs and, 831, 988
- Visual Basic, Visual C++ and, 832

InterlockedIncrement and InterlockedDecrement functions, 931

Internet, **603, 605, 720–780**. *See also* Web sites

- ActiveX Web controls, **775–780**
 - defined, **775**
 - initializing Web browsers, 777–780, 778
 - WebView demo program, 775–780, 776
- cookies
 - defined, **761**
 - Internet API cookie functions, 761
- FTP (File Transfer Protocol)
 - defined, **728**
 - FTP functions in Internet API, 759–760
 - FtpGet demo program, 764–774, 765
 - FtpGetFile function, 760, 764, 771–773
 - FtpOpenFile function, 760, 774
- HTTP (Hypertext Transfer Protocol)
 - defined, **728**
 - HTTP functions in Internet API, 759–760

Internet advantages of ActiveX Data Objects and OLE DB, 603

Internet API functions, **752–774**

- AutoDial functions, 755
- connect and disconnect functions, 754–755
- cookie functions, 761
- defined, **752–753**
- error and status functions, 762–763
- file locating, reading, and writing functions, 758–759
- FindOneOf, 768
- FTP, Gopher, and HTTP functions, 759–760
- FtpGet demo program, 764–774, 765
- FtpGetFile, 760, 764, 771–773
- FtpOpenFile, 760, 774
- GetLastError, 762, 766
- Internet applications, 774
- InternetAttemptConnect, 754, 756
- InternetAutodial, 755, 757–758
- InternetConnect, 755, 764, 769–770
- InternetGetConnectedState, 755, 756–757, 758
- InternetGetLastResponseInfo, 762, 766
- InternetOpen, 755, 764, 768
- InternetOpenUrl, 762, 774

- InternetReadFile, 759, 774
- LastErrorMsg, 766
- limitations of, 753
- MFC wrapper classes, 763–764
- OnOK, 767
- opening dial-up connections, 756–758
- time/date conversion functions, 762
- URL-related functions, 761–762
- Internet protocols, 727–728
- IP addresses
 - defined, **726–727**
 - FindAddr demo program, 738–743, 739, 740
- IP (Internet Protocol), 727
- Microsoft Internet support, **720–726**
 - Internet connectivity in operating systems, 721
 - Internet servers, 721–722
 - Internet-related programming tools, 722
 - Internet-related SDKs and APIs, 723–725
 - other Internet-related technologies, 725
 - Web design and management tools, 721
- POP3 (Post Office Protocol), 728
- proxy servers, 769
- Remote Terminal Access protocol, 728
- SMTP (Simple Mail Transfer Protocol), 727
- sockets
 - defined, **728–729**
 - socket functions in Winsock 2 API, 730–731
 - TCP versus UDP sockets, 729
 - Winsock defined, **729**
- TCP (Transmission Control Protocol)
 - defined, **727**
 - TCP sockets, 729
- Telnet protocol, 728
- UDP (User Datagram Protocol)
 - defined, **727**
 - UDP sockets, 729
- Universal Data Access and, 605
- Winsock 2 API functions, **730–752**
 - asynchronous functions, 735–738
 - database-lookup functions, 732–735
 - defined, **730**
 - FindAddr demo program, 738–743, 739, 740
 - gethostbyaddr, 732, 742
 - gethostbyname, 732, 741–742, 745–746, 747
 - HOSTS and SERVICES files, 733–735
 - network byte order and host byte order conversion functions, 731–732
 - SendMail demo program, 743–750, 744
 - SendMail2 demo program, 751

- socket functions, 730–731
 - Winsock applications, 751–752
 - Winsock defined, **728–729**
- interpreting error messages, 218, 289–290
- interrupt dispatch tables (IDTs), 296
- interrupt objects, 296
- interrupt request levels (IRQLs), 295
- interrupt service routines (ISRs), 295–296
- interrupts. *See* exception handling
- introductory programs. *See* “Hello, World” example; Template example
- Invalidate function, 311
- IObjectControl interface, 872
- IP addresses
 - defined, **726–727**
 - FindAddr demo program, 738–743, 739, 740
- IP (Internet Protocol), 727
- IPipeByte interface, 972–977
- IRQLs (interrupt request levels), 295
- IRQs. *See* exception handling
- IsBad memory validation functions, 355–356
- IsOnFirstRecord and IsOnLastRecord functions, 666
- IsOpen, IsBOF, IsEOF, and IsDeleted functions, 651
- isotropic mapping mode, 480–481, 481
- ISRs (interrupt service routines), 295–296
- IsUserAllowed method, 947
- IsValidAcl function, 421
- IUnknown interface. *See also* interfaces
 - and creating COM servers, 881–884
 - defined, **834**
 - deriving interfaces from, 831, 838–839
 - implement IUnknown rule, 844

J

- job object functions, 105–107

K

- Katmai New Instructions, 576
- kernel, **104–107**
 - defined, **104**
 - job object functions, 105–107
 - security objects and, 394
 - trap handler, 294–296
- keyboards, capturing input for pop-up message windows, 125–126
- keys
 - in databases, 611
 - in encryption
 - creating random keys, 437–440, 439
 - CryptExportKey function, 438

- CryptExportPublicKeyInfo function, 440
- CryptGenKey function, 437–438
- CryptGetUserKey function, 437–438
- CryptImportKey function, 440
- key functions, 430–431
- public-key encryption, 427, 428
- symmetric-key encryption, 428
- in Registry
 - defined, **273–275**, 276
 - opening, 280–281
 - querying keys and values, 281–287
 - setting keys and values, 287–289

L

- LAN Manager API functions, 792
- LastErrorMsg function, 766
- late binding, 852
- launching child processes, 253–255
- LDAP (Lightweight Directory Access Protocol), 988, 989–990
- LeaveCriticalSection function, 165
- life cycles
 - of pipes, 191
 - of processes, 189
- Lightweight Directory Access Protocol (LDAP), 988, 989–990
- lightweight handlers (LWHs), **981–983**. *See also* COM in Windows 2000
 - custom lightweight handlers, 983, 983
 - defined, **981**
 - standard lightweight handlers, 981–982, 982
 - Web site about, 983
- LILO (Linux Loader) software, 20
- line capabilities of devices, 463, 475
- linking files
 - hard links, 84–86
 - in NTFS file system, 84–86
- list box entries, selecting in ADO programming, 699–700
- List demo program, **356–369**. *See also* memory management
 - adding items, 359–361
 - CommitMemFilter function, 360–361
 - CreateList function, 358–359
 - defined, **356–357**, 357
 - DeleteList function, 363
 - deleting items, 361–364
 - initializing data structures, 358–359
 - setting list size and structure, 357–358
 - VirtualAlloc function, 359, 366
 - VirtualProtect function, 364–366
 - write-protecting list pages, 364–369
- load balancing, **870–871**

- application clusters and, 871
 - COM+ load balancing, 870–871
 - component or load-balancing routers, 871
 - defined, **870**
 - Windows Load Balancing (WLB), 870–871
- load and store MXCSR register, 592
- LoadDealerList function, 668–669
- local memory functions, **353–355**
 - overview of, 353–355
 - system limits and allocating memory, 355
- local service authority (LSA) functions, 415–416
- locating file functions in Internet API, 758–759
- locking
 - memory, 348–349
 - records in Booklist program, 651
- LockServer function, 885–887
- LockWorkStation function, 108
- logical SIMD instructions, 584
- long filenames in NTFS file system
 - converting to 8.3 format, 64
 - guidelines for compatibility between file systems, 78
 - NetWare and, 79
 - overview of, 63–64
- LookupAccountName function, 419
- low-level multimedia command sets, 508–509
- LSA (local service authority) functions, 415–416
- LWHs. *See* lightweight handlers

M

- macros
 - exception-handling macros, 304
 - message-map macros, 45
- mailslots
 - mailslot functions, 799–801
 - versus pipes, 230–231
- Main_OnCommand function, 173–174
- Main_OnCreate function, 170–172, 177
- Main_OnInitMenu function, 173
- Main_OnSize function, 169–170, 172–173
- Main_OnTimer function, 169–170, 173
- mapping. *See* file-mapping objects
- mapping modes, **478–489**. *See also* Graphics Device Interface
 - Cartesian coordinate modes, 479–480, 480
 - default text mode, 479, 479
 - in DOS versus Windows, 478
 - GetMapMode and SetMapMode functions, 481–483
 - isotropic and anisotropic modes, 480–481, 481
 - Modes demo program, 487–489, 488

- setting window and viewport extents, 486–487
 - setting window and viewport origins, 485–486
 - viewport versus window coordinate translations, 483–487
- marshaling services for COM servers, 930
- marshalling code
 - defined, **803–804**
 - Mathlib.ACF marshalling interface handle file, 810–811
- Master File Tables (MFTs), 65–66
- Mathlib.ACF marshalling interface handle file, 810–811
- Mathlib.CPP function library, 806
- Mathlib.H header file, 806–808, 811–812
- Mathlib.IDL function interface definition file, 808–810
- Mathlib_c.C and Mathlib_s.C files, 811–812
- MCA (Multilink Channel Aggregation) technology, 15
- MCI (Media Control Interface). *See also* multimedia
 - overview of functions, **508–509, 515–517**
 - ShowWave program MCI module, **522, 523–533**
 - defined, **522**
 - error handling, 533
 - opening and closing devices, 524–527
 - overview of, 523–524
 - playing sounds, 529–530
 - recording sounds, 531–532
 - saving sounds, 532
 - setting time formats, 527–528
 - stopping sounds, 530
- MDAC (Microsoft Data Access Components), 598
- memory
 - buffers
 - buffer sizes in CreateNamedPipe function, 215
 - FlushFileBuffers function, 227
 - Translation Look-aside Buffers (TLBs), 333, 943
 - move masked byte to memory SIMD instruction, 590
 - Windows 2000 requirements for, 7
- memory management, **328–385**
 - address space, **338–339, 338, 349–350**
 - allocating memory
 - allocating heap, 351–352, 355
 - defined, **336–337**
 - GlobalAlloc and LocalAlloc functions, 354, 355
 - heap and, 350
 - system limits and, 355
 - VirtualAlloc function, 341–345, 355, 359, 366
 - Browser demo program, **377–385**. *See also* file-mapping objects
 - ChangeAccess function, 384–385
 - CreateFile function, 380
 - CreateFileMapping function, 379–380
 - creating and using mapped files, 378–379
 - defined, **377–378, 378**

- DoWrite, DoRead, and DoClear functions, 382–384
- editing shared objects, 383–385
- FindWindow function, 380
- initializing mapped files, 379–380
- OpenFileMapping function, 380
- running the dialog box, 380–382
- C runtime equivalents for memory functions, 356
- file-mapping objects, **339–340**, **369–385**. *See also* Browser demo program
 - CreateFile function, 370, 380
 - CreateFileMapping function, 370, 371–373, 379–380
 - creating, **371–373**
 - defined, **369**
 - mapping views, 374
 - MapViewOfFile function, 370, 374
 - memory-mapped files defined, **339–340**, **369–370**, 371
 - OpenFileMapping function, 375, 380
 - preserving coherence, 375–377, 376
 - sharing, 375–377, 376
 - uses of copy-on-write protection, 377
- freeing memory, 345–346, 354
- global and local memory functions, **353–355**
 - overview of, 353–355
 - system limits and allocating memory, 355
- heap functions, **349–353**
 - heap defined, **349**
 - HeapAlloc, 351, 355
 - HeapCreate, 350–352
 - HeapDestroy, 353
 - HeapFree, 351, 352–353
 - HeapRealloc, 352
 - HeapSize, 352
 - overview of, 349–350
- history of, 328–329
- image activation tables, 339
- IsBad functions, 355–356
- List demo program, **356–369**
 - adding items, 359–361
 - CommitMemFilter function, 360–361
 - CreateList function, 358–359
 - defined, **356–357**, 357
 - DeleteList function, 363
 - deleting items, 361–364
 - initializing data structures, 358–359
 - setting list size and structure, 357–358
 - VirtualAlloc function, 359, 366
 - VirtualProtect function, 364–366
 - write-protecting list pages, 364–369
- locking and unlocking memory, 348–349

- memory management rule for components, 844

- protecting memory, 346–347

- querying memory information, 347–348

- reserving memory

 - defined, **336–337**

 - heap and, 349–350

 - uses for reserved memory, 345

- validation functions, **355–356**

- virtual memory functions, **340–349, 355**

 - GetWriteWatch, 342

 - overview of, 340–341

 - ResetWriteWatch, 342

 - system limits and allocating memory, 355

 - VirtualAlloc, 341–345, 355, 359, 366

 - VirtualFree, 345–346

 - VirtualLock, 348–349

 - VirtualProtect, 346–347, 364–366

 - VirtualQuery, 343, 347–348

 - VirtualUnlock, 348–349

- Virtual Memory Manager (VMM), **329–337**

 - allocating memory, 336–337, 350

 - committed pages, 333

 - committing memory, 336–337, 349–350

 - defined, **329–330**

 - demand paging, 335

 - free pages, 333

 - page directories, 333, 334

 - page frame databases and page tables, 332–333

 - page frame states, 333–335

 - page frames, 332

 - page interrupts or page faults, 331–332, 334

 - page policies, 335–336

 - pages and processes, 335

 - paging, 331–332

 - pointers and movable memory, 330–331

 - process working sets, 335

 - reserved pages, 333

 - reserving memory, 336–337, 345, 349–350

 - selectors, 330–331

 - Translation Look-aside Buffers (TLBs), 333, 943

 - virtual address descriptors (VADs), 336–337, 337

 - virtual memory and, 329–330

 - virtual memory APIs, 337

 - in Windows 2000 versus Windows 95/98, 329

 - zeroed pages, 335–336

- menus, pop-up or floating menus, 128–132, 129

- message boxes versus pop-up tip windows, 116

- message windows

 - message-only windows, 107

pop-up message windows, **120–128**

calculating window size, 126–128

capturing mouse and keyboard input, 125–126

creating CTipWnd class, 122–125

PopupTips function, 120–122, 120

ShutDown method, 125–126

MessageBeep function, 511–512

message-driven programming, 38–39

message-map macros, 45

message-mapped functions, 44–45

messages

FormatMessage function, 218, 289–290

GetErrorMessage function, 306

GetLastError function, 218, 229, 290

interpreting error messages, 218, 289–290

message handling in “Hello, World” example, 36–38

message pipes, 192

Plug-and-Play event messages, **206–209**

defined, **306–307**

how Plug-and-Play messages function, 307

listed, 308–309

responding to system messages, 233–236, 234

retrieving error messages, 218, 229, 290

threads and message queues, 141

transactional message queuing, 864, 864

windows and, 40

methods. *See also* functions

adding to in-process COM servers, 906–908, 907, 915–916

adding to out-of-process COM servers, 923

AddNew in ADO, 710–712

AddRef, 884, 893

CalcWndSize, 123

CreatePopupMenu, 130

CTipWnd::Create, 124–125

CWnd::CreateEx, 124

DrawText, 125

GetIdsOfNames, 933, 947

IADs interface methods, 997–999

IADsUser methods, 1012–1016, 1015

implementing in in-process COM servers, 908–909

IsUserAllowed, 947

OnKillThreadsSlow and OnKillThreadsFast, 181–182

PopupMenu, 129–130

QueryInterface, 884, 888

Release, 884, 888, 893

SetCapture, 125

ShutDown, 125–126

SuspendThread, 142

- thread methods, 141–142, 142
- MFC (Microsoft Foundation Class). *See also* classes
 - creating MFC clients for testing in-process COM servers, 916–918, 916, 917
 - creating MFC threads with CWinThread class, 151–153
 - “Hello, World” example and, 30, 44
 - MFC exception handling
 - defined, **297**
 - exception classes, 305–306, 312–316
 - MFC wrapper classes, 763–764
 - SQL (Structured Query Language) and, 612–613
 - Template example and, 48
- MFTs (Master File Tables), 65–66
- microprocessors. *See* processors
- Microsoft Active Directory. *See* Active Directory
- Microsoft Data Access Components (MDAC), 598
- Microsoft Data Access SDK, 599
- Microsoft Distributed Transaction Coordinator (MS DTC), 859–860, 861
- Microsoft Foundation Class. *See* MFC
- Microsoft Interface Definition Language (MIDL), 836, 876
- Microsoft Internet support, **720–726**. *See also* Internet
 - Internet connectivity in operating systems, 721
 - Internet servers, 721–722
 - Internet-related programming tools, 722
 - Internet-related SDKs and APIs, 723–725
 - other Internet-related technologies, 725
 - Web design and management tools, 721
- Microsoft Message Queue (MSMQ), 857–858, 863, 870, 988
- Microsoft Transaction Server (MTS), 857–858, 872
- Microsoft Windows 3.x
 - NTFS file system and, 64
 - Registry versus Windows 3.x initialization files, 270
- Microsoft Windows 95/98 versus Windows 2000
 - encryption, 392
 - memory management, 329
 - security, 388–392
 - security API functions, 391–392
- Microsoft Windows 2000. *See also* COM in Windows 2000
 - 64-bit version, 6, 328
 - cross-system application development, 17–18
 - DLL file protection, 16–17
 - dual-boot systems, **12, 18–24**
 - Be operating system multiboot manager, 19
 - and dual-installing applications, 23–24
 - file systems and, 12
 - hardware boot management with removable hard drives, 19
 - LILO (Linux Loader) software, 20
 - network names and, 23
 - NT Loader (NTLDR) software, 19, 20–22
 - overview of, 18–19

- PowerQuest Boot Magic software, 19–20
- setting up, 21–22
- V Communications System Commander software, 20
- FAT16 versus FAT32 versus NTFS file systems, 9–13
- flags, 74
- hardware, **7–9**
 - AGP (Accelerated Graphics Port) support, 15, 90
 - DVD drive support, 15
 - hard disk drive requirements, 8–9
 - hardware compatibility, 6–7, 8
 - hardware compatibility list, 7
 - IDE hard drives, 8
 - memory requirements, 7
 - MMX processor support, 15
 - Multilink Channel Aggregation (MCA) technology, 15
 - multiple monitor support, 15
 - Plug-and-Play feature, 14
 - processor requirements, 7
 - SCSI hard drives, 8
 - USB (Universal Serial Bus), 14–15
 - Win32 driver model, 18
 - WinModem card support, 8
- kernel, **104–107**
 - defined, **104**
 - job object functions, 105–107
 - security objects and, 394
 - thread pooling, 107
 - trap handler, 294–296
- user interface and Graphics Device Interface (GDI), **107–111**
 - active window tracking, 109
 - Alphablend function, 108
 - common controls support, 108
 - foreground switching, 109–110
 - GetGuiResources function, 108
 - GUI appearance settings, 110
 - handicapped preference settings, 109
 - HTML resources, 108
 - LockWorkStation function, 108
 - message-only windows, 107
 - mouse operation settings, 110
 - power settings, 111
 - screen saver status settings, 110
 - SendInput function, 108
 - SystemParametersInfo function, 109–111
 - task switching settings, 110
 - TrackMouseEvent function, 108
- versions of, 6, 15–16
- versus Windows 95/98

- encryption, 392
- memory management, 329
- security, 388–392
- security API functions, 391–392
- Windows 2000 Advanced Server, 15–16
- Windows 2000 Datacenter, 15–16
- Windows 2000 Professional, 15–16
- Windows 2000 Server, 15–16
- versus Windows NT, 11, 13–18
- Microsoft Windows NT
 - AutoChk, ChkDsk, and ScanDisk utilities, 63
 - versus Windows 2000, 11, 13–18
- middle tier technologies, 857–858
- MIDI format, 510
- MIDL (Microsoft Interface Definition Language), 836, 876
- min/max SIMD instructions, 580, 587–588
- mixing sounds, 559–561
- MMIO (multimedia I/O). *See also* multimedia
 - MMIO functions, **509, 517–522**
 - chunk operations, 520–521
 - defined, **509**
 - file operations, 519–520
 - PCMWAVEFORMAT structure, 521–522
 - RIFF (Resource Interchange File Format) and, 509, 517–518, 518
 - ShowWave program MMIO module, **522, 533–543**
 - defined, **522**
 - error handling, 543
 - overview of, 533–534
 - reading .WAV files, 534–538
 - writing .WAV files, 539–543
- MMX processors. *See also* SIMD
 - Windows 2000 support for, 15
- modems, WinModem card support, 8
- Modes demo program, 487–489, 488
- modification procedures in Threads example, 174–177
- monitors. *See* multiple monitor support
- mouse
 - capturing input for pop-up message windows, 125–126
 - OnLButtonDown and OnRButtonDown functions, 119
 - TrackMouseEvent function, 108
- move byte mask to integer SIMD instruction, 588
- move mask from floating point to integer SIMD instruction, 583
- move masked byte to memory SIMD instruction, 590
- MoveFirst function, 675
- MoveMemory command, 356
- MoveNext function, 698
- MS DTC (Microsoft Distributed Transaction Coordinator), 859–860, 861
- MSG message structure, 39–41, 57
- MSMQ (Microsoft Message Queue), 857–858, 863,

- 870, 988
- MTAs (multithreaded apartments), 927–929, 928
- MTS (Microsoft Transaction Server), 857–858, 872
- Multilink Channel Aggregation (MCA) technology, 15
- multimedia, **506–522**. *See also* ShowWave demo program; sound
 - Media Control Interface (MCI) functions, **508–509**, **515–517**
 - multimedia devices
 - device elements, 526
 - opening and closing, 524–527
 - simple versus compound devices, 525
 - Windows 2000 support for, 506–507
 - multimedia I/O (MMIO) functions, **509**, **517–522**
 - chunk operations, 520–521
 - defined, **509**
 - file operations, 519–520
 - PCMWAVEFORMAT structure, 521–522
 - RIFF (Resource Interchange File Format) and, 509, 517–518, 518
 - multimedia services, **508–510**
 - low-level and high-level command sets, 508–509
 - multimedia animation, 510
 - multimedia timer, 509–510
 - timeGetDevCaps function, 509
- multiple data stream support, 79–81, 80
- multiple monitor support, **15**, **90–94**
 - AGP (Accelerated Graphics Port) video cards and, 90
 - compatibility issues, 93–94
 - configuring, 91, 92
 - and displaying multiple applications, 93
 - overview of, 15
 - as remote displays, 93
 - resolution and, 91
 - rules applied to primary and secondary monitors, 92–93
 - screen capture utilities and, 93
 - uses for, 91–94, 92
- multiple threads. *See also* threads
 - MultiThreads demo program, 180–182, 181
 - overview of, 139
 - Threads example, **166–180**
 - CreateWindows function, 167, 168–169
 - defined, **166**, 166
 - DoThread procedure, 174–177
 - DrawProc procedure, 178–179
 - initialization procedures, 166–169
 - InitializeApp procedure, 168
 - Main_OnCommand function, 173–174
 - Main_OnCreate function, 170–172, 177
 - Main_OnInitMenu function, 173
 - Main_OnSize function, 169–170, 172–173

- Main_OnTimer function, 169–170, 173
 - modification procedures, 174–177
 - SetThreadPriority function, 168
 - StartThread function, 177–178
 - thread procedures, 177–179
 - viewing in Process Viewer, 179–180, 180
 - window and message handler procedures, 169–174
- multiply high unsigned SIMD instruction, 588
- multiprocessors, threads and, 139–140
- multitasking
 - versus multithreading, 186–187
 - permissive multitasking, 187
 - preemptive multitasking, 167, 187
- multithreading
 - versus multitasking, 186–187
 - multithreaded apartments (MTAs), 927–929, 928
 - multithreaded Win32 applications, 926–927
- multi-value keys, 611
- mutex objects
 - acquiring, 161–162
 - CreateMutex function, 161, 162
 - defined, **146–147**
 - OpenMutex function, 164–165
 - processes and, 147
 - ReleaseMutex function, 162
 - sharing and destroying, 164–165

N

- named pipes. *See also* pipes
 - creating, **249–251**
 - creating named pipe instances, 251–253
 - defined, **193–194**
 - GetNamedPipeHandleState function, 220–221
 - GetNamedPipeInfo function, 221–222
 - ImpersonateNamedPipeClient function, 229–230
 - named pipe functions, 799–801
 - NamePipe demo program, **231–232, 247–265**
 - child process, **262–265**
 - connecting with clients, 257–260
 - ConnectNamedPipe function, 257
 - CreateFile function, 263–264
 - CreateNamedPipe function, 251–253
 - CreateProcess function, 253–255, 262
 - creating named pipe instances, 251–253
 - creating named pipes, 249–251
 - creating threads and connecting to pipes, 263–265
 - defined, **231–232, 247–248, 248**
 - DisconnectNamedPipe function, 257

- ExitProcess function, 258
- GetCommandLine function, 263
- GetStdHandle function, 263
- launching child processes, 253–255
- parent process, **249–262**
- PulseEvent function, 255
- ReadFile function, 265
- ResetEvent function, 258–259
- SendCommand function, 260–262
- synchronizing threads, 255–257
- WaitForSingleObject function, 258–259
- WaitNamedPipe function, 261, 264
- WriteFile function, 261
- writing to pipes, 260–262
- PeekNamedPipe function, 222–223
- SetNamedPipeHandleState function, 219–220
- TransactionNamedPipe function, 228–229
- WaitNamedPipe function, 225, 226–227, 229, 261, 264
- in Windows 95/98 versus Windows NT/2000, 193
- naming
 - pipes, 210–211
 - variables, 54–55
- native structured storage, 97
- NEAR data type, 39, 57
- nested exception handler example, 312–316, 316
- NetBIOS functions, **784–787**
- NetWare, NTFS long filenames and, 79
- network byte order conversion functions, 731–732
- network names, dual-boot systems and, 23
- network programming, **784–819**
 - LAN Manager API functions, **792**
 - named pipe and mailslot functions, **799–801**
 - NetBIOS functions, **784–787**
 - Remote Procedure Call project example, **805–818**
 - Mathlib.ACF marshalling interface handle file, 810–811
 - Mathlib.CPP function library, 806
 - Mathlib.H header file, 806–808, 811–812
 - Mathlib.IDL function interface definition file, 808–810
 - Mathlib_c.C and Mathlib_s.C files, 811–812
 - overview of, 805–806
 - RPC client and server testing, 818–819, 818
 - RPCClient demo program, 812–814
 - RPCServ demo program, 814–818
 - Remote Procedure Calls (RPCs), **801–805**
 - DCOM (Distributed Component Object Model) and, 801–802
 - defined, **801–802**
 - GUIDs (globally unique identifiers), 802–803
 - interface classes, 803
 - marshalling code, 803–804

- RPC network connections, 805
 - UUID (universally unique identifiers), 802–803
- Winsock 2 functions for network programming, **793–799**
 - AppleTalk protocol and Winsock 2, 794–799, 798 defined, **793**
 - MacSock demo program, 794–799, 798
 - new features, 793–794
- WNet API functions, **787–791**
 - defined, **787–788**
 - network resources and, 788–789
 - WNetDemo program, 789–791, 790
- NewTitleID function, 667–668, 671
- nonblocking pipes, 192
- nontemporal cached data, 589–590
- Novell NetWare, NTFS long filenames and, 79
- NT Loader (NTLDR) software, 19, 20–22
- NTFS file system, **9–13, 62–86, 94–97**. *See also* FAT file systems; file systems
 - defined, **62–63**
 - DOS, Windows 3.x applications and, 64
 - versus FAT16 and FAT32, 9–13, 62–63
 - file attributes, 68–69
 - floppy disks, removable media and, 63, 65
 - limitations of, 64–65
 - linking files, 84–86
 - long filenames
 - converting to 8.3 format, 64
 - guidelines for compatibility between file systems, 78
 - NetWare and, 79
 - overview of, 63–64
 - Master File Tables (MFTs), 65–66
 - new features in Windows 2000, **94–97**
 - Change Journal, 94
 - Encrypting File System (EFS) driver, 96
 - Installable File System (IFS) Kit, 95
 - native structured storage, 97
 - reparse points, 94–95
 - spare file support, 95–96
 - volume or disk quotas, 96–97
 - volume mount points, 97
 - writing installable file system filter drivers, 95
 - object-oriented programming and, 62–63
 - querying file/drive information, **69–78**
 - NTFS_Ops demo program, 69–76, 70
 - querying file systems, 76–78, 76
 - security and, 395, 396
 - streams, **79–84**
 - creating, 81–82, 82
 - creating multistream examples, 80

- deleting, 84
- multiple data stream support, 79–81, 80
- NTFS_Streams demo program, 81–84, 82
- system crashes and recovery and, 63

O

- object pooling, 872
- object threading models, 929–930
- object-oriented programming, **848–855**
 - abstraction, 850
 - Component Object Model and, 848–849
 - encapsulation, 849–850
 - interface inheritance, 853–855
 - interface inheritance using early binding, 852–853
 - late binding, 852
 - polymorphism, 851–853
- ODBC. *See* Open Database Connectivity
- OLE DB, **600–602**, **680–681**. *See also* ActiveX Data Objects (ADO); Universal Data Access (UDA)
 - ActiveX Data Objects and, 602
 - advantages of, 601–602
 - availability of, 600
 - cross-platform support with, 604–605, 604
 - data consumers, data providers, and service components, 600, 682
 - defined, **600–601**, 601, **680–681**
 - Internet advantages of, 603
 - ODBC (Open Database Connectivity) and, 600–601, 680–681
- OLE/COM View utility, 944, 944
- OnActiveView function, 645
- OnCancel function, 674
- OnClose function, 696
- OnDbclbResultList function, 699–700
- OnEditEntry function, 674
- OnEditTitle function, 662–663
- OnGetRecordset function, 629, 630
- OnInitDialog function, 693–694, 697, 701
- OnInitialUpdate function
 - in Booklist program, 668
 - for CTitleForm class, 630
 - in Popups program, 118
 - in Publisher program, 640, 641, 642
- OnKillfocusX functions, 133–135
- OnKillThreadsSlow and OnKillThreadsFast methods, 181–182
- OnLButtonDown and OnRButtonDown functions, 119
- OnNewTitle function, 670–671
- OnNow ACPI specification, 104
- OnOK function, 767
- OnPaint function, 311, 451
- OnShowOrders function, 644

- OnSize function, 118–119
- Open Database Connectivity (ODBC). *See also* databases
 - OLE DB and, 600–601, 680–681
 - registering databases with, 615–618, 616, 617
- Open function, 695
- OpenFileMapping function, 375, 380
- OpenGL technology, **498–502**. *See also* DirectX technology
 - advantages of, 499–501
 - defined, **499**
 - versus DirectX technology, 498
 - disadvantages of, 501
 - versus Fahrenheit technology, 502
- opening
 - data files in ShowWave demo program, 557
 - data source connections, 693–696
 - dial-up connections, 756–758
 - multimedia devices, 524–527
 - Registry keys, 280–281
- OpenJobObject function, 105
- OpenMutex, OpenSemaphore, and OpenEvent functions, 164–165
- OpenProcess function, 205–206
- order of execution in exception handling, 302–304
- outbound pipes, 191
- out-of-process COM servers, **847, 896–897, 920–925**
 - adding CoClasses, properties, and methods, 923
 - versus creating in-process COM servers, 921–923
 - out-of-process servers defined, **847**
 - overview of Active Template Library, 896–897, 897
 - overview of, 920, 921
 - SecurityMgrExe.cpp file, 922–923
 - StdAfx.h file, 922
 - testing out-of-process servers, 923–925, 924
- overlapping parameters in CreateNamedPipe function, 211–212

P

- packed byte, packed word, and packed doubleword integers, 586, 586
- packed floating-point SIMD instructions, 578
- padding bytes, 428
- paging, **331–336, 348–349**. *See also* memory management
 - committed pages, 333
 - defined, **331–332**
 - demand paging, 335
 - free pages, 333
 - locking and unlocking pages, 348–349
 - page directories, 333, 334
 - page frame databases and page tables, 332–333
 - page frame states, 333–335
 - page frames, 332

- page interrupts or page faults, 331–332, 334
- page policies, 335–336
- processes and, 335
- reserved pages, 333
- zeroed pages, 335–336
- PAINTSTRUCT structure, 452–453
- Parameters collection in ADO, 685–686
- parent pipes, 236–241
- parent processes
 - in AnonPipe demo program, 232–243
 - defined, **187–189**
 - in NamePipe demo program, 249–262
- ParseCommandLine function, 1018–1019
- Partition Magic software, 13
- Pascal ordering, 30–31, 56
- PCMWAVEFORMAT structure, 521–522
- PeekNamedPipe function, 222–223
- Pentium III processors. *See* SIMD
- permissive multitasking, 187
- per-user security model, 393–394, 394
- pipes, **190–194, 208–265, 972–978**. *See also* processes; threads
 - AnonPipe demo program, **195, 231–247**
 - child process, **243–247**
 - CloseHandle function, 243
 - CreatePipe function, 237
 - CreateProcess function, 237–241, 243, 245
 - creating parent and child pipes, 236–241
 - defined, **231–232, 232**
 - DuplicateHandle function, 237
 - GetStdHandle function, 238, 245
 - inheriting pipe handles and creating threads, 244–245
 - initializing the parent process, 233
 - parent process, **232–243**
 - ReadFile function, 247
 - reading from pipes, 245–247
 - responding to system messages, 233–236, 234
 - SendCommand function, 241–243
 - StartProcess function, 237, 238–241
 - warning about testing, 195
 - WriteFile function, 242
 - writing to pipes, 241–243
 - anonymous pipes
 - creating with CreatePipe function, 208, 237
 - defined, **193–194**
 - blocking and nonblocking pipes, 192
 - byte and message pipes, 192
 - COM pipes, **972–978**
 - asynchronous pipe interfaces and read-ahead, 977–978
 - COM pipe interfaces, 972–977

- defined, **972**
- IPipeByte interface, 972–977
- CreateNamedPipe function, **209–218, 251–253**
 - access, write-through, and overlapping parameters, 211–212
 - buffer sizes, 215
 - defined, **209–210**
 - in NamePipe demo program, 251–253
 - naming pipes, 210–211
 - overview of parameters, 216–217
 - pipe instances, 214–215
 - return values, 217–218
 - security attributes parameter, 216
 - time-out period parameter, 216
 - type, read mode, and wait parameter, 212–214
- defined, **190, 972**
- inbound, outbound, and duplex pipes, 191
- instances of, 214–215, 251–253
- life cycle of, 191
- mailslots
 - mailslot functions, 799–801
 - versus pipes, 230–231
- named pipes
 - creating, **249–251, 799–801**
 - creating named pipe instances, 251–253
 - defined, **193–194**
 - GetNamedPipeHandleState function, 220–221
 - GetNamedPipeInfo function, 221–222
 - ImpersonateNamedPipeClient function, 229–230
 - named pipe functions, 799–801
 - PeekNamedPipe function, 222–223
 - SetNamedPipeHandleState function, 219–220
 - TransactionNamedPipe function, 228–229
 - WaitNamedPipe function, 225, 226–227, 229, 261, 264
 - in Windows 95/98 versus Windows NT/2000, 193
- NamePipe demo program, **231–232, 247–265**
 - child process, **262–265**
 - connecting with clients, 257–260
 - ConnectNamedPipe function, 257
 - CreateFile function, 263–264
 - CreateNamedPipe function, 251–253
 - CreateProcess function, 253–255, 262
 - creating named pipe instances, 251–253
 - creating named pipes, 249–251
 - creating threads and connecting to pipes, 263–265
 - defined, **231–232, 247–248, 248**
 - DisconnectNamedPipe function, 257
 - ExitProcess function, 258
 - GetCommandLine function, 263

- GetStdHandle function, 263
- launching child processes, 253–255
- parent process, **249–262**
- PulseEvent function, 255
- ReadFile function, 265
- ResetEvent function, 258–259
- SendCommand function, 260–262
- synchronizing threads, 255–257
- WaitForSingleObject function, 258–259
- WaitNamedPipe function, 261, 264
- WriteFile function, 261
- writing to pipes, 260–262
- naming, 210–211
- pipe-related functions, **208–231**
 - asynchronous I/O parameter, 224
 - blocking parameter, 224
 - CallNamedPipe, 228, 229
 - CloseHandle, 227, 229, 230, 243
 - ConnectNamedPipe, 225–226, 227, 257
 - CreateFile, 218–219, 225, 229, 263–264
 - CreateNamedPipe, 209–218, 251–253
 - creating anonymous pipes with CreatePipe, 208, 237
 - for destroying pipes, 230
 - DisconnectNamedPipe, 227, 257
 - ExitProcess, 206–207, 230, 258
 - FlushFileBuffers, 227
 - GetLastError, 218, 229
 - GetNamedPipeHandleState, 220–221
 - GetNamedPipeInfo, 221–222
 - ImpersonateNamedPipeClient, 229–230
 - PeekNamedPipe, 222–223
 - ReadFile, 218, 223–224, 229, 247, 265
 - RevertToSelf, 229–230
 - SetNamedPipeHandleState, 219–220
 - for synchronizing connections, 225–227
 - TransactionNamedPipe, 228–229
 - WaitNamedPipe, 225, 226–227, 229, 261, 264
 - WriteFile, 218, 223–224, 229, 242, 261
- processes and, 186
- processes communicating through pipes, 231–232
- planning
 - COM interfaces, 829, 832
 - security, 392–393
- playing sounds
 - with MessageBeep function, 511–512
 - with PlaySound function, 513–515
 - in ShowWave demo program, 529–530, 549–550, 563–565
 - with sndPlaySound function, 512–513
- Plug-and-Play feature

defined, **14**

Plug-and-Play event messages, **206–209**

defined, **306–307**

how Plug-and-Play messages function, 307

listed, 308–309

pointers

interface pointers, 831

movable memory and, 330–331

polygon capabilities of devices, 463, 475–476

polymorphism, 851–853

pooling

objects, 872

threads, 107

POP3 (Post Office Protocol), 728

populating

data fields in ADO, 700–704, 703

data from another table, 668

pop-up tip windows, **116–118**. *See also* windows

defined, **116–118**

versus message boxes or dialog boxes, 116

tooltips, 117

Popups demo program, **118–135**. *See also* windows

CheckMenuItem and CheckMenuRadioItem functions, 131

CreatePopupMenu method, 130

defined, **118–119**

EnableMenuItem function, 130

OnInitialUpdate function, 118

OnKillfocusX functions, 133–135

OnLButtonDown and OnRButtonDown functions, 119

OnSize function, 118–119

pop-up or floating menus, 128–132, 129

pop-up message windows, **120–128**

calculating window size, 126–128

capturing mouse and keyboard input, 125–126

creating CTipWnd class, 122–125

PopupTips function, 120–122, 120

ShutDown method, 125–126

PopupMenu method, 129–130

pop-ups in dialog boxes, 132–135, 132, 133

TrackPopupMenu function, 131–132

ports

AGP (Accelerated Graphics Port)

AGP video cards and multiple monitor support, 90

support for, 15

PostQuitMessage function, 43–44

power management, **97–104, 111, 309**

Advanced Power Management (APM) system, 309

creating power-aware applications, 98–99

- creating “power-friendly” applications, 101–102
- OnNow ACPI specification, 104
- overview of, 97–98
- power event types, 98–99
- power settings, 111
- querying power status with GetSystemPowerStatus function, 99–101, 101
- SetThreadExecutionState function and, 102–104

PowerQuest

- Boot Magic software, 19–20
- Partition Magic software, 13

PPTP (Point to Point Tunneling Protocol), 389

predefined constants, 55–56

preemptive multitasking, 167, 187

prefetch SIMD instructions, 590

prefixes for constants, 56

preparing Visual C++ to use Active Directory Service Interfaces, 992–993, 992, 993

preserving coherence in file-mapping, 375–377, 376

preventing problems with switching application views, 645–647

primary keys, 611

primary monitors, 92–93

primary single-threaded apartments (STAs), 929

primary threads, 138

printer-related device capabilities, 462

priority parameters for CreateProcess function, 197

priority of threads

- process, base, and dynamic priority, 143–145, 144, 295
- SetThreadPriority function, 153–154, 168

privileges. *See also* security

- privilege functions, 415
- versus rights, 401

procedures

- GetClientRect, 42

- InitApplication and InitInstance, 50

- in Threads example

 - DoThread procedure, 174–177

 - DrawProc procedure, 178–179

 - initialization procedures, 166–169

 - InitializeApp procedure, 168

 - modification procedures, 174–177

 - thread procedures, 177–179

 - window and message handler procedures, 169–174

- WinMain procedure

 - PeekMessage loops, 167

 - in ShowWave demo program, 522, 544

 - in Template example, 49–50

- WinMain procedure in “Hello, World” example, **30–38**

 - arguments, 30–32

 - creating application windows, 34–36

 - displaying windows, 36

- message handling, 36–38
- Pascal ordering, 30–31, 56
- registering window classes, 32–34
- WndProc procedure in “Hello, World” example, 41–44
- process priority of threads, 143–145, 144
- Process Viewer, viewing Threads example in, 179–180, 180
- process working sets, 335
- ProcessData function, 962–963, 969
- processes, **186–266**. *See also* pipes; threads
 - background processes, 139
 - child processes
 - in AnonPipe demo program, 243–247
 - defined, **139, 187–189**
 - launching, 253–255
 - in NamePipe demo program, 262–265
 - communicating through pipes, 231–232
 - CreateProcess function, **194–201, 203, 204–205**
 - in AnonPipe demo program, 237–241, 243, 245
 - assigning priority classes, 203
 - blocking inheritance, 196–197
 - C runtime library equivalents, 201
 - defined, **194**
 - environment and directory parameters, 197–198
 - executable files and arguments parameters, 195–196
 - I/O handles, 200
 - letting children inherit handles, 204–205
 - in NamePipe demo program, 253–255, 262
 - parameters pointing to structures, 198–200
 - process type and priority parameters, 197
 - return values, 201
 - security attributes, 196
 - defined, **186–187**
 - handles for, 148
 - inheritance
 - blocking, 196–197
 - defined, **187–189**
 - inheriting pipe handles and creating threads, 244–245
 - letting children inherit handles in CreateProcess function, 204–205
 - life cycle of, 189
 - and multitasking versus multithreading, 186–187
 - mutexes, semaphores, events, critical sections and, 147
 - parent processes
 - in AnonPipe demo program, 232–243
 - defined, **187–189**
 - in NamePipe demo program, 249–262
 - pipes and, 186
 - process-related functions, **194–231**
 - CreateProcess, 194–201, 203, 204–205

- DuplicateHandle, 205, 237
 - ExitProcess, 206–207, 230, 258
 - GetCommandLine, 202, 263
 - GetCurrentProcess and GetCurrentProcessId, 202
 - GetEnvironmentVariable, 202
 - GetExitCodeProcess, 206
 - GetPriorityClass, 203
 - OpenProcess, 205–206
 - for sharing handles, 204–206
 - StartProcess, 237, 238–241
 - for synchronizing processes, 203–204
 - TerminateProcess, 207
 - WaitForInputIdle, 204
 - WaitForSingleObject and WaitForMultiple-Objects, 203, 258–259
- sibling processes, 187–189
- versus threads, 138–139, 186–187
- when to create, 140–141
- processors. *See also* SIMD
 - Compaq Alpha processors, 7
 - DirectX and Pentium III processors, 495
 - MMX processor support, 15
 - threads and multiprocessors, 139–140
 - Windows 2000 requirements for, 7
- project settings
 - adding libraries to ADSI project Link settings, 994, 994
 - in Crypto demo program, 434–435, 434
- properties
 - adding to in-process COM servers, 903–904, 903, 904, 914
 - adding to out-of-process COM servers, 923
 - implementing in in-process COM servers, 905–906
 - Properties collection in ADO, 686
- protected-mode FAT file system (VFAT), 68
- protecting memory, 346–347
- protocols
 - AppleTalk, 794–799, 798
 - FTP (File Transfer Protocol)
 - defined, **728**
 - FTP functions in Internet API, 759–760
 - FtpGet demo program, 764–774, 765
 - FtpGetFile function, 760, 764, 771–773
 - FtpOpenFile function, 760, 774
 - HTTP (Hypertext Transfer Protocol)
 - defined, **728**
 - HTTP functions in Internet API, 759–760
 - Internet protocols, 727–728
 - IP (Internet Protocol), 727
 - LDAP (Lightweight Directory Access Protocol), 988, 989–990
 - POP3 (Post Office Protocol), 728
 - PPTP (Point to Point Tunneling Protocol), 389

SMTP (Simple Mail Transfer Protocol), 727

SSL (Secure Socket Layer), 389

TCP (Transmission Control Protocol)

defined, **727**

TCP sockets, 729

Telnet protocol, 728

UDP (User Datagram Protocol)

defined, **727**

UDP sockets, 729

proxy servers, 769

pseudohandles, 155–156

public-key encryption, 427, 428

Publisher demo program, **618–647**. *See also* Booklist demo program; databases

adding table access, **632–636**

CDealerSet and CDealerForm classes, 636

creating classes for record views, 634–635, 635

creating procedure for, 632–634, 633, 634

changing classes after creation, **636–647**

CheckOrders function, 645–646

Class Info tab in Class Wizard and, 636, 636

customizing COrderSet and COrderForm classes, 637–642

defining Filter parameter, 638–640

GetDocument function, 641

initializing COrderForm class, 640–642

OnActiveView function, 645

OnInitialUpdate function, 640, 641, 642

OnShowOrders function, 644

overview of, 636–637

preventing problems with switching application views, 645–647

SelectForm function, 644, 645, 646

switching application views, 642–645

creating, **618–623**, 618, 623

CTitleForm class, **628–632**

GetTableName function, 632

OnGetRecordset function, 629, 630

OnInitialUpdate function, 630, 632

setting sort order, 630–631

setting window captions, 631–632

CTitleSet class, **623–628**

DoDataExchange function, 627–628

DoFieldExchange function, 625–626

GetDefaultConnect function, 626

GetDefaultConnectSQL function, 626–627

header file, 623–627

transferring data from databases to recordsets, 627–628

PulseEvent function, 163, 255

PutCollect function, 707

QC. *See* queued components

quadword integers, 586, 586

querying

- device contexts, 451–452
- file/drive information, **69–78**
 - NTFS_Ops demo program, 69–76, 70
 - querying file systems, 76–78, 76
- memory information, 347–348
- for object interfaces, 851–852
- power status, 99–101, 101
- QueryInformationJobObject function, 106
- Registry keys and values, 281–287

QueryInterface method, 884, 888

queued components (QC), **863–870**

- architecture of, 865, 865
- asynchronous calling mechanism, 957
- client-side dead letter queues, 869
- defined, **863**
- failure recovery, 867–869
- limitations of real-time architectures, 863–864
- Microsoft Message Queue (MSMQ) and, 863, 870
- QC client components, 867
- QC server components, 866
- runtime architecture, 865–867
- security, 869–870
- server-side retry queues, 867–869
- transactional message queuing, 864, 864

R

Rapid Application Development (RAD) tools, 824

raster capabilities of devices, 464–465, 471–473

read mode parameters for CreateNamedPipe function, 212–214

read-ahead, 977–978

ReadFile function, 218, 223–224, 229, 247, 265

reading

- error messages, 218, 289–290

- from pipes, 245–247

- reading file functions in Internet API, 758–759

- .WAV files, 534–538

ready threads, 145

real-time architectures, 863–864

recording sounds in ShowWave demo program, 531–532, 549–550, 565–566

records

- in ADO programming, **704–713**

- appending, 708–712

- deleting, 712–713

- FetchRecords function, 696–697, 702–703

- updating, 704–708

- in Booklist program
 - adding, 647–648, 666–670
 - appending, 650
 - canceling Edit or Add Record operations, 674–675
 - creating record IDs, 667–668
 - editing, 649
 - locking, 651
 - updating, 647–648
- creating classes for record views, 634–635, 635
- in SQL
 - restricting record selection, 613–614
 - sorting, 614–615
- recordsets. *See also* Booklist demo program
 - CRecordset update transactions, **648–651**
 - AddNew function, 648, 650
 - appending records, 650
 - CancelUpdate function, 649
 - CanUpdate, CanAppend, and CanRestart functions, 650
 - checking the legality of operations, 650–651
 - CRecordset class, 648–649
 - Delete function, 649
 - Edit function, 649
 - editing records, 649
 - IsOpen, IsBOF, IsEOF, and IsDeleted functions, 651
 - locking records, 651
 - SetLockingMode function, 651
 - Update function, 649, 650
 - OnGetRecordset function, 629, 630
 - Recordset objects in ADO, 685
 - transferring data from databases to, 627–628
- recovery, NTFS file system and, 63
- Red Book audio format (Compact Disc-Digital Audio format), 510
- reference counting rules for components, 845
- registering
 - databases with Open Database Connectivity, 615–618, 616, 617
 - window classes
 - in “Hello, World” example, 32–34
 - in ShowWave demo program, 566–568
- RegisterWaitForSingleObject function, 107
- Registry, **270–291**
 - API functions
 - defined and listed, **276–278**
 - RegCreateKey, 280–281, 285, 287
 - RegCreateKeyEx, 288
 - RegEnumKey, 284–285
 - RegOpenKeyEx, 288
 - RegQueryInfoKey, 282–283
 - RegQueryValueEx, 274, 285
 - RegSetValueEx, 274, 287–289

- data types, 274–275
- defined, **270–271**
- versus DOS and Windows 3.x initialization files, 270
- editing, 273
- keys

- defined, **273–275**, 276
 - opening, 280–281
 - querying keys and values, 281–287
 - setting keys and values, 287–289

- Registry Editors, 271–273, 271

- Reg_Ops demo program, **279–291**

- defined, **279**, 279
 - FormatMessage function, 289–290
 - GetLastError function, 290
 - InitializeListBox function, 283–284
 - interpreting system error messages, 289–290
 - opening keys, 280–281
 - preparing to run, 279–280
 - querying keys and values, 281–287
 - RegCreateKey function, 280–281, 285, 287
 - RegCreateKeyEx function, 288
 - RegEnumKey function, 284–285
 - RegOpenKeyEx function, 288
 - RegQueryInfoKey function, 282–283
 - RegQueryValueEx function, 274, 285
 - RegSetValueEx function, 274, 287–289
 - running, 290–291
 - setting keys and values, 287–289

- structure of, 273–275, 276

- relational databases, 611

- Release method, 884, 888, 893

- ReleaseDC function, 453

- ReleaseMutex and ReleaseSemaphore functions, 162

- remote displays, multiple monitors as, 93

- Remote Procedure Calls (RPCs), **801–818**. *See also* network programming

- DCOM (Distributed Component Object Model) and, 801–802

- defined, **801–802**

- GUIDs (globally unique identifiers), 802–803

- interface classes, 803

- marshalling code, 803–804

- RPC network connections, 805

- RPC project example, **805–818**

- Mathlib.ACF marshalling interface handle file, 810–811

- Mathlib.CPP function library, 806

- Mathlib.H header file, 806–808, 811–812

- Mathlib.IDL function interface definition file, 808–810

- Mathlib_c.C and Mathlib_s.C files, 811–812

- overview of, 805–806

- RPC client and server testing, 818–819, *818*
- RPCClient demo program, 812–814
- RPCServ demo program, 814–818
- UUIDs (universally unique identifiers), 802–803
- Remote Terminal Access protocol, 728
- remote use, designing COM interfaces for, 838
- removable hard drives
 - dual-boot management with, 19
 - NTFS file system and, 63, 65
- reparse points, 94–95
- ReportError function, 306
- ReportStatus function, 437
- Requery function, 664
- reserved pages, 333
- reserving memory
 - defined, **336–337**
 - heap and, 349–350
 - uses for reserved memory, 345
- ResetControls function, 663
- ResetEvent function, 162–163, 258–259
- resetting
 - controls in Booklist program, 665–666
 - ShowWave demo program, 554–555
- ResetWriteWatch function, 342
- resolution on multiple monitors, 91
- resource exception handler example, 318–322, *321*
- Resource Interchange File Format (RIFF), 509, 517–518, *518*
- resource script for ShowWave demo program, 523
- resources, threads and resource allocation, 139
- responding
 - to commands in ShowWave demo program, 544–546
 - to system messages, 233–236, *234*
- restricting record selections in SQL, 613–614
- ResumeThread function, 150, 154–155
- retrieving
 - error messages, 218, 229, 290
 - filenames in ShowWave demo program, 555–556
 - security descriptors, 419–420
 - security identifiers, 418–419
- return values
 - for CreateNamedPipe function, 217–218
 - for CreateProcess function, 201
 - HRESULT type return value, 839–841, 883, 893
- returning HRESULT type values, 839–841, 883, 893
- RevertToSelf function, 229–230
- RFX_xxxx function calls, 648, 655–659, *656*
- RIFF (Resource Interchange File Format), 509, 517–518, *518*
- rights. *See also* security
 - access checking functions, 413–414

- adding, 406–407
- checking, 406
- defined, **406–407**
- generic rights, 407
- versus privileges, 401
- specific rights, 407
- standard rights, 407
- Rollback function, 652–653
- rotate SIMD shuffle operation, 581, 589
- routers, component or load-balancing routers, 871
- RPCs. *See* Remote Procedure Calls
- rules
 - for component implementation, 843–845
 - interface rules, 837–841
- running
 - Reg_Ops demo program, 290–291
 - test clients for in-process COM servers, 919–920
- running threads, 145

S

- SACLs (system access control lists)
 - overview of, 394, 401, 402
 - SACL functions, 412
- SAFEARRAY data type, 941
- sampling rates, 511
- saving data in ShowWave demo program, 532, 558–559
- scalar floating-point SIMD instructions, 579
- ScanDisk utility, 63
- scheduling threads. *See also* threads
 - how scheduling happens, 145
 - process, base, and dynamic priority, 143–145, 144, 295
 - SetThreadPriority function, 153–154, 168
- SCM (Service Control Manager), 846
- screen capture utilities, 93
- screen saver status settings, 110
- scripts
 - resource script for ShowWave demo program, 523
 - Template.RC resource script, 52–54, 53
- scrolling wave images in ShowWave demo program, 546–549, 546
- SCSI hard drives, 8
- SDs (security descriptors). *See also* FileUser demo program
 - access control entries (ACEs) and, 401, 402, 403
 - access control lists (ACLs) and, 401, 402, 403
 - converting self-relative SDs to absolute SDs, 403
 - defined, **393–394**, 394, **401**
 - GetSecurityDescriptorDacl function, 420–421
 - retrieving, 419–420
 - SD functions, 401, 410

self-relative versus absolute SDs, 401–403, 402

setting, 423–424

secondary monitors, 92–93

secondary single-threaded apartments (STAs), 929

Secure Channels support, 389

Secure Socket Layer (SSL) protocol, 389

security, **196, 216, 388–424**. *See also* encryption

access control entries (ACEs)

access control lists (ACLs) and, 401, 402, 403, 404

ACE functions, 412–413

in FileUser demo program, 421–423

access control lists (ACLs)

access control entries (ACEs) and, 401, 402, 403, 404

ACL functions, 412

defined, **404**

discretionary (owner) access control lists (DACs), 394, 401, 402, 404, 412, 420–423

GetAclInformation function, 420–421

IsValidAcl function, 421

security descriptors (SDs) and, 401, 402, 403, 404, 423

system access control lists (SACLs), 394, 401, 402, 412

access tokens

access token functions, 408–409

defined, **400**

and adding encryption support to applications, 442–444

anti-virus software, 429

attacks, 443–444

Authenticode support, 389

built-in Windows NT/2000 security versus application security, 395–396

client impersonation

client impersonation functions, 409–410

defined, **405–406**

ImpersonateNamedPipeClient function, 229–230

client/server model and, 396–398

FAT file systems and, 395

FileUser demo program, **418–424**. *See also* SDs (security descriptors)

adding access control entries, 421–423

Check function, 419

defined, **418**

discretionary (owner) access control list operations, 420–423

GetAclInformation function, 420–421

GetFileSecurity function, 419–420

GetSecurityDescriptorDacl function, 420–421

IsValidAcl function, 421

LookupAccountName function, 419

retrieving security descriptors, 419–420

retrieving security identifiers, 418–419

SetFileSecurity function, 423–424

setting security descriptors, 423–424

NTFS file system and, 395, 396

- per-user security model, 393–394, 394
- planning, 392–393
- PPTP (Point to Point Tunneling Protocol) and, 389
- privileges
 - privilege functions, 415
 - versus rights, 401
- queued component (QC) security, 869–870
- rights
 - access checking functions, 413–414
 - adding, 406–407
 - checking, 406
 - defined, **406–407**
 - generic rights, 407
 - versus privileges, 401
 - specific rights, 407
 - standard rights, 407
- Secure Channels support, 389
- Secure Socket Layer (SSL) protocol, 389
- security API functions, **391–392, 407–417**
 - access checking functions, 413–414
 - access control entry (ACE) functions, 412–413
 - access control list (ACL) functions, 412
 - access token functions, 408–409
 - auditing functions, 417
 - client impersonation functions, 409–410
 - converting self-relative SDs to absolute SDs, 403
 - discretionary (owner) access control list (DACL) functions, 412
 - GetAclInformation, 420–421
 - GetSecurityDescriptorDacl, 420–421
 - IsValidAcl, 421
 - local service authority (LSA) functions, 415–416
 - overview of, 407–408
 - privilege functions, 415
 - security descriptor (SD) functions, 401, 410
 - security identifier (SID) functions, 411
 - Set/Get security information functions, 416
 - SetFileSecurity, 423–424
 - system access control list (SACL) functions, 412
 - window station functions, 417
 - in Windows 95/98 versus Windows 2000, 391–392
- security attributes
 - for CreateNamedPipe function, 216
 - for CreateProcess function, 196
- security descriptors (SDs). *See also* FileUser demo program
 - access control entries (ACEs) and, 401, 402, 403
 - access control lists (ACLs) and, 401, 402, 403
 - converting self-relative SDs to absolute SDs, 403
 - defined, **393–394, 394, 401**

- GetSecurityDescriptorDacl function, 420–421
- retrieving, 419–420
- SD functions, 401, 410
- self-relative versus absolute SDs, 401–403, 402
- setting, 423–424
- security identifiers (SIDs)
 - defined, **403–404**
 - retrieving, 418–419
 - SID functions, 411
- security objects
 - application-defined objects, 395
 - GDI-defined objects, 395
 - kernel objects, 394
 - system objects, 394
 - user-defined objects, 395
 - users and, 393–394, 394
- security structures, 400–404
- SECURITY_ATTRIBUTES structure, 398–400
- smart card support, 389
- in Windows 95/98 versus Windows 2000, **388–392**
- SecurityMgrExe.cpp file, 922–923
- SelectForm function, 644, 645, 646
- selecting list box entries in ADO programming, 699–700
- selectors, 330–331
- self-relative security descriptors (SDs), 401–403, 402
- semantics of interfaces, 843
- semaphore objects
 - acquiring, 161–162
 - CreateSemaphore function, 161, 162
 - defined, **146–147**
 - OpenSemaphore function, 164–165
 - processes and, 147
 - ReleaseSemaphore function, 162
 - sharing and destroying, 164–165
- SendCommand function
 - in AnonPipe demo program, 241–243
 - in NamePipe demo program, 260–262
- SendInput function, 108
- SendMail demo program, 743–750, 744
- SendMail2 demo program, 751
- sequencing operations in Booklist demo program, 661–662, 661
- servers. *See also* COM servers
 - in Asynchronous COM
 - asynchronous server modifications, 970–971
 - asynchronous server processing modes, 964–966, 965
 - asynchronous server processing runtime events, 966–967
 - Microsoft Internet servers, 721–722
 - Microsoft Transaction Server (MTS), 857–858, 872
 - proxy servers, 769

- QC server components, 866
 - RPC server testing, 818–819, *818*
 - RPCServ demo program, 814–818
 - security and, 396–398
 - server-side retry queues, 867–869
- service components in OLE DB, 600
- Service Control Manager (SCM), 846
- SERVICES files, 733–735
- Set/Get security information functions, 416
- SetBkColor function, 125
- SetCapture method, 125
- SetEvent function, 162–163
- SetFileSecurity function, 423–424
- SetInformationJobObject function, 106
- SetLockingMode function, 651
- SetMapMode function, 481–483
- SetNamedPipeHandleState function, 219–220
- SetThreadExecutionState function, 102–104
- SetThreadPriority function, 153–154, 168
- setting
 - Registry keys and values, 287–289
 - security descriptors, 423–424
 - sort order in databases, 630–631
 - time formats in ShowWave demo program, 527–528
 - window captions, 631–632
 - window and viewport extents, 486–487
 - window and viewport origins, 485–486
- setting up dual-boot systems, 21–22
- sharing
 - file-mapping objects, 375–377, 376
 - mutexes, semaphores, and events, 164–165
- ShowWave demo program, **522–572**. *See also* multimedia
 - defined, **522–523**, 523
 - GraphWin module, **522**, **566–572**
 - defined, **522**
 - drawing sound waves, 568–571, 568
 - registering window classes, 566–568
 - header files and resource script, 523
 - improvements, 571–572
 - MCI module, **522**, **523–533**
 - defined, **522**
 - error handling, 533
 - opening and closing devices, 524–527
 - overview of, 523–524
 - playing sounds, 529–530
 - recording sounds, 531–532
 - saving sounds, 532
 - setting time formats, 527–528

- stopping sounds, 530
- MMIO module, **522, 533–543**
 - defined, **522**
 - error handling, 543
 - overview of, 533–534
 - reading .WAV files, 534–538
 - writing .WAV files, 539–543
- ShowWave module, **522, 544–566**
 - changing volume, 561–563, 562
 - closing the program, 550–553
 - defined, **522, 544**
 - displaying information in static controls, 553–554
 - mixing sounds, 559–561
 - opening data files, 557
 - playing sounds, 563–565
 - recording sounds, 565–566
 - resetting the program, 554–555
 - responding to commands, 544–546
 - retrieving filenames, 555–556
 - saving data, 558–559
 - scrolling wave images, 546–549, 546
 - stopping play or record operations, 549–550
- WinMain module, **522, 544**
- ShowWindow function, 36
- shuffle SIMD instruction, 580–581, 588–589
- ShutDown method, 125–126
- sibling processes, 187–189
- SIDs (security identifiers)
 - defined, **403–404**
 - retrieving, 418–419
 - SID functions, 411
- signatures
 - digital signatures, 427
 - interface signatures, 842–843
- SIMD (Single Instruction, Multiple Data) extensions, **576–593**
 - cacheability control instructions, **589–591**
 - defined, **589**
 - move masked byte to memory instruction, 590
 - nontemporal, temporal, and spatially cached data, 589–590
 - prefetch instructions, 590
 - store fence instruction, 591
 - streaming store instructions, 589–590
 - defined, **576–577**
- DirectX technology and, 577
- floating-point instructions, **578–585**
 - arithmetic instructions, 579–580
 - broadcast shuffle operation, 581
 - compare and set EFLAGS instructions, 584–585
 - compare and set mask instructions, 583

- conversion operation instructions, 585
- data movement instructions, 582–583
- defined, **578–579**, 578
- fast approximation operations, 580
- logical instructions, 584
- min/max comparisons, 580
- move mask from floating point to integer instruction, 583
- packed operations, 578
- rotate shuffle operation, 581
- scalar operations, 579
- shuffle instruction, 580–581
- square root instructions, 580
- swap shuffle operation, 581
- unpack instruction, 581–582
- integer instructions, **585–589**
 - broadcast shuffle operation, 588–589
 - defined, **585–587**, 586
 - extract instruction, 587
 - insert instruction, 587
 - min/max instruction, 587–588
 - move byte mask to integer instruction, 588
 - multiply high unsigned instruction, 588
 - packed byte, packed word, packed doubleword, and quadword integers, 586, 586
 - rotate shuffle operation, 589
 - shuffle instruction, 588–589
 - swap shuffle operation, 589
- Katmai New Instructions, 576
- State Management operations, **591–592**
 - control status register (MXCSR), 591–592
 - load and store MXCSR register, 592
 - state save and restore instructions, 592
- support for, 577
- types of, 576–577
- Web sites about, 592–593
- simple exception handler example, 310–312, 310
- Simple Mail Transfer Protocol (SMTP), 727
- simple multimedia devices, 525
- Single Instruction, Multiple Data extensions. *See* SIMD
- single-threaded apartments (STAs)
 - defined, **927–928**, 928
 - primary versus secondary STAs, 929
- 64-bit version of Windows 2000, 6, 328
- size of devices, 460, 470–471
- Sleep and SleepEx functions, 154–155
- smart card support, 389
- SMTP (Simple Mail Transfer Protocol), 727
- sndPlaySound function, 512–513
- sockets. *See also* Internet

defined, **728–729**

socket functions in Winsock 2 API, 730–731

TCP versus UDP sockets, 729

Winsock 2 API functions, **730–752**

asynchronous functions, 735–738

database-lookup functions, 732–735

defined, **730**

FindAddr demo program, 738–743, 739, 740

gethostbyaddr, 732, 742

gethostbyname, 732, 741–742, 745–746, 747

HOSTS and SERVICES files, 733–735

network byte order and host byte order conversion functions, 731–732

SendMail demo program, 743–750, 744

SendMail2 demo program, 751

socket functions, 730–731

Winsock applications, 751–752

Winsock 2 functions for network programming, **793–799**

AppleTalk protocol and Winsock 2, 794–799, 798

defined, **793**

MacSock demo program, 794–799, 798

new features, 793–794

Winsock defined, **729**

software. *See also* utilities

LILO (Linux Loader), 20

NT Loader (NTLDR), 19, 20–22

PowerQuest

Boot Magic, 19–20

Partition Magic software, 13

V Communications System Commander, 20

Visual Studio

Enterprise Setup dialog box, 687–689, 687, 688

OLE/COM View utility, 944, 944

sort order in databases, 630–631

sorting records in SQL, 614–615

sound, **510–515**. *See also* multimedia; ShowWave demo program

Compact Disc-Digital Audio (Red Book audio) format, 510

MIDI format, 510

playing sounds, **511–515, 529–530, 549–550, 563–565**

with MessageBeep function, 511–512

with PlaySound function, 513–515

in ShowWave demo program, 529–530, 549–550, 563–565

with sndPlaySound function, 512–513

recording sounds in ShowWave demo program, 531–532, 549–550, 565–566

sampling rates, 511

in ShowWave demo program

changing volume, 561–563, 562

drawing sound waves, 568–571, 568

mixing sounds, 559–561

opening data files, 557

- playing sounds, 529–530, 563–565
- reading .WAV files, 534–538
- recording sounds, 531–532, 565–566
- scrolling wave images, 546–549, 546
- stopping play or record operations, 549–550
- stopping sounds, 530
- writing .WAV files, 539–543
- .WAV files
 - defined, **510–511**
 - reading, 534–538
 - writing, 539–543
- sparse file support, 95–96
- spatially cached data, 589–590
- specific access rights, 407
- SQL (Structured Query Language), **612–615**. *See also* databases
 - defined, **612**
 - GetDefaultConnectSQL function, 626–627
 - MFC (Microsoft Foundation Class) and, 612–613
 - restricting record selections, 613–614
 - sorting records, 614–615
 - SQL statements in Booklist program, 669–670
 - tracing SQL operations, 659–660
- square root SIMD instructions, 580
- SSL (Secure Socket Layer) protocol, 389
- stacks, address space and, 339
- standard access rights, 407
- standard lightweight handlers, 981–982, 982
- standby threads, 145
- StartProcess function, 237, 238–241
- StartThread function, 177–178
- STAs (single-threaded apartments)
 - defined, **927–928**, 928
 - primary versus secondary STAs, 929
- State Management operations, **591–592**. *See also* SIMD
 - control status register (MXCSR), 591–592
 - load and store MXCSR register, 592
 - state save and restore instructions, 592
- states of threads, 145
- status functions in Internet API, 762–763
- StdAfx.h file, 922
- stopping
 - play or record operations, 549–550
 - ShowWave demo program, 550–553
 - sounds in ShowWave demo program, 530
- stopping in ShowWave demo program, sounds, 530
- Storage+ technology, 856–857, 858
- store fence SIMD instruction, 591
- streaming store SIMD instructions, 589–590

- streams, **79–84**
 - creating, 81–82, 82
 - creating multistream examples, 80
 - deleting, 84
 - multiple data stream support, 79–81, 80
 - NTFS_Streams demo program, 81–84, 82
 - stream encryption, 428
- string parameters as Unicode strings, 841
- structured exception handling. *See* exception handling
- Structured Query Language. *See* SQL
- structures
 - CreateProcess parameters pointing to, 198–200
 - data structures
 - defined, **57**
 - initializing in List demo program, 358–359
 - MSG (message structure), 39–41, 57
 - PAINTSTRUCT structure, 452–453
 - PCMWAVEFORMAT structure, 521–522
 - security structures, 400–404
 - SECURITY_ATTRIBUTES structure, 398–400
- subprocesses, 139, 187–189
- support for transient environments, 681
- Supports tests, 708, 713–714
- SuspendThread function, 154–155
- SuspendThread method, 142
- swap SIMD shuffle operation, 581, 589
- switching
 - application views, 642–645
 - preventing problems with, 645–647
 - foreground switching, 109–110
 - task switching settings, 110
- symmetric-key encryption, 428
- synchronization mechanisms in COM, **954–957**. *See also* COM in Windows 2000
 - COM synchronization API, 954–955
 - COM synchronization interfaces, 955–957
 - CoWaitForMultipleHandles API function, 954–955
 - history of, 954
 - ISynchronize interface, 955–956
 - ISynchronizeContainer interface, 956
 - ISynchronizeHandle interface, 956–957
- synchronizing
 - pipe connections, 225–227
 - processes, 203–204
- synchronizing threads, **143–147**. *See also* threads
 - commands for, 159–165
 - critical section objects
 - critical section commands, 165
 - defined, **146**
 - processes and, 147

defined, **143**

event objects

CreateEvent function, 163

defined, **146–147**

OpenEvent function, 164–165

processes and, 147

ResetEvent function, 162–163, 258–259

SetEvent function, 162–163

sharing and destroying, 164–165

how scheduling happens, 145

how synchronization happens, 146–147

mutex objects

acquiring, 161–162

CreateMutex function, 161, 162

defined, **146–147**

OpenMutex function, 164–165

processes and, 147

ReleaseMutex function, 162

sharing and destroying, 164–165

in NamePipe demo program, 255–257

semaphore objects

acquiring, 161–162

CreateSemaphore function, 161, 162

defined, **146–147**

OpenSemaphore function, 164–165

processes and, 147

ReleaseSemaphore function, 162

sharing and destroying, 164–165

synchronization objects, 146

thread priority

process, base, and dynamic priority, 143–145, *144*, 295

SetThreadPriority function, 153–154, 168

thread states, 145

syntax, interface syntax, 842–843

SysAllocString function, 1019

system access control lists (SACLs)

overview of, 394, 401, *402*

SACL functions, 412

System Commander software, 20

system crashes and recovery, 63

system file protection, 16–17

system security objects, 394

SystemParametersInfo function, 109–111

T

tables. *See also* databases

adding table access to Publisher program, **632–636**

CDealerSet and CDealerForm classes, 636

- creating classes for record views, 634–635, 635
 - creating procedure for, 632–634, 633, 634
 - defined, **611–612**, 611
 - GetTableName function, 632
 - populating data from another table, 668
- task switching settings, 110
- TCP (Transmission Control Protocol)
 - defined, **727**
 - TCP sockets, 729
- Telnet protocol, 728
- Template example, **48–54**. *See also* “Hello, World” example
 - InitApplication and InitInstance procedures, 50
 - MFC foundation classes and, 48
 - overview of, 48–49
 - Template.C source code, 49–50
 - Template.H header file, 49, 51
 - Template.I file, 50
 - Template.RC resource script, 52–54, 53
 - Windows.H header file, 49
 - WinMain procedure, 49–50
- temporal cached data, 589–590
- terminated threads, 145
- TerminateJobObject function, 106
- TerminateProcess function, 207
- TerminateThread function, 158
- test clients for in-process COM servers, 919–920
- testing
 - in-process COM servers, 916–920
 - out-of-process COM servers, 923–925, 924
 - RPC clients and servers, 818–819, 818
- text
 - default text mapping mode, 479, 479
 - text capabilities of devices, 463–464, 476–477
 - TextOut function, 451
- Third Party API, 990
- threads, **102–104**, **107**, **138–182**. *See also* exception handling; pipes; processes
 - calling, 929–930
 - creating
 - and connecting to pipes, 263–265
 - with CreateThread function, 148–151, 152–153, 159
 - MFC threads with CWinThread class, 151–153
 - defined, **138**
 - inheriting pipe handles and creating threads, 244–245
 - initial or primary threads, 138
 - message queues and, 141
 - multiple threads
 - MultiThreads demo program, 180–182, 181
 - overview of, 139
 - multiprocessors and, 139–140

- multitasking versus multithreading, 186–187
- versus processes, 138–139, 186–187
- ready threads, 145
- resource allocation and, 139
- running threads, 145
- scheduling and synchronizing, **143–147**
 - acquiring mutexes and semaphores, 161–162
 - CreateEvent function, 163
 - CreateMutex and CreateSemaphore functions, 161, 162
 - critical section functions, 165
 - critical section objects, 146
 - defined, **143**
 - event objects defined, **146–147**
 - functions for synchronizing threads, 159–165
 - how scheduling happens, 145
 - how synchronization happens, 146–147
 - mutex objects defined, **146–147**
 - OpenMutex, OpenSemaphore, and OpenEvent functions, 164–165
 - process, base, and dynamic priority, 143–145, 144, 295
 - ResetEvent function, 162–163, 258–259
 - semaphore objects defined, **146–147**
 - SetEvent function, 162–163
 - SetThreadPriority function, 153–154, 168
 - sharing and destroying mutexes, semaphores, and events, 164–165
 - synchronization objects defined, **146**
 - synchronizing threads in NamePipe demo program, 255–257
 - thread states, 145
- SetThreadExecutionState function, 102–104
- standby threads, 145
- terminated threads, 145
- thread objects, **141–143**
 - defined, **141–142**, 142
 - handles for, 142–143, 148
 - SuspendThread method, 142
 - thread methods and attributes, 141–142, 142
 - Win32 APIs and, 142
- thread pooling, 107
- threaded COM servers, **926–931**
 - apartments and Active Template Library, 930–931
 - apartments defined, **927–928**, 928
 - CoInitialize and CoInitializeEx functions, 929
 - execution contexts, 897
 - InterlockedIncrement and InterlockedDecrement functions, 931
 - marshaling services, 930
 - multithreaded apartments (MTAs), 927–929, 928
 - multithreaded Win32 applications, 926–927
 - object threading models and calling threads, 929–930
 - primary versus secondary single-threaded apartments (STAs), 929

- single-threaded apartments (STAs), 927–928, 928
- threading COM components, 927–931, 928
- thread-related functions, **148–165**
 - for acquiring mutexes and semaphores, 161–162
 - AfxBeginThread, 152
 - C runtime library equivalents, 158–159
 - CloseHandle, 148, 150–151, 156, 157
 - CreateEvent, 163
 - CreateMutex and CreateSemaphore, 161, 162
 - creating MFC threads with CWinThread class, 151–153
 - creating threads with CreateThread function, 148–151, 152–153, 159
 - for critical sections, 165
 - DuplicateHandle, 156, 164–165
 - ExitThread, 156–158, 159
 - GetCurrentThread and GetCurrentThreadId, 155–156
 - GetExitCodeThread, 157
 - OnKillThreadsSlow and OnKillThreadsFast methods, 181–182
 - OpenMutex, OpenSemaphore, and OpenEvent, 164–165
 - processes and, 147
 - PulseEvent, 163, 255
 - ReleaseMutex and ReleaseSemaphore, 162
 - ResetEvent, 162–163, 258–259
 - ResumeThread, 150, 154–155
 - SetEvent, 162–163
 - SetThreadPriority, 153–154, 168
 - for sharing and destroying mutexes, semaphores, and events, 164–165
 - Sleep and SleepEx, 154–155
 - StartThread, 177–178
 - SuspendThread, 154–155
 - for synchronizing threads, 159–165
 - TerminateThread, 158
 - WaitForSingleObject and WaitForMultipleObjects, 159–162
- Threads multiple threads example, **166–180**
 - CreateWindows function, 167, 168–169
 - defined, **166**, 166
 - DoThread procedure, 174–177
 - DrawProc procedure, 178–179
 - initialization procedures, 166–169
 - InitializeApp procedure, 168
 - Main_OnCommand function, 173–174
 - Main_OnCreate function, 170–172, 177
 - Main_OnInitMenu function, 173
 - Main_OnSize function, 169–170, 172–173
 - Main_OnTimer function, 169–170, 173
 - modification procedures, 174–177
 - SetThreadPriority function, 168
 - StartThread function, 177–178
 - thread procedures, 177–179
 - viewing in Process Viewer, 179–180, 180

- window and message handler procedures, 169–174
 - transition threads, 145
 - user-interface threads, 152
 - waiting threads, 145
 - when to create, 140–141
 - window creation and, 141
 - working threads, 152
- time. *See also* dates
 - date/time stamps, 73
 - multimedia timer, 509–510
 - setting time formats in ShowWave demo program, 527–528
 - time/date conversion functions in Internet API, 762
 - timeGetDevCaps function, 509
 - time-out period parameter for CreateNamedPipe function, 216
- tip windows. *See* pop-up tip windows
- TLBs (Translation Look-aside Buffers), 333, 943
- tooltips, 117
- tracing SQL operations, 659–660
- tracking active windows, 109
- TrackMouseEvent function, 108
- TrackPopupMenu function, 131–132
- transactions
 - in Booklist demo program, 652–653
 - in COM+ model, 861–863, 862
 - transactional message queuing, 864, 864
 - TransactionNamedPipe function, 228–229
- transferring data from databases to recordsets, 627–628
- transient environment support, 681
- transition threads, 145
- TranslateMessage function, 37, 38–39
- Translation Look-aside Buffers (TLBs), 333, 943
- Transmission Control Protocol (TCP), 727
- traps and the trap handler, 294–296
- type libraries, 837, 876, 943–945, 944, 945
- type parameters
 - for CreateNamedPipe function, 212–214
 - for CreateProcess function, 197

U

UDP (User Datagram Protocol)

- defined, **727**

- UDP sockets, 729

Unicode strings, string parameters as, 841

Universal Data Access (UDA), **598–606**. *See also* ActiveX Data Objects (ADO); COM (Component Object Model)

- Component Object Model (COM) and, 599–600

- defined, **598**

- the Internet and, 605

- Microsoft Data Access Components (MDAC), 598
- Microsoft Data Access SDK, 599
- OLE DB, **600–602, 680–681**
 - ActiveX Data Objects and, 602
 - advantages of, 601–602
 - availability of, 600
 - cross-platform support with, 604–605, 604
 - data consumers, data providers, and service components, 600, 682
 - defined, **600–601, 601, 680–681**
 - Internet advantages of, 603
 - ODBC (Open Database Connectivity) and, 600–601, 680–681
- universally unique identifiers (UUIDs), 802–803
- unlocking memory, 348–349
- unpack SIMD instruction, 581–582
- UnregisterWaitEx function, 107
- updating
 - records
 - in ADO programming, 704–708
 - in Booklist program, 647–648
 - Update function, 649, 650, 674
 - UpdateData function, 671–673, 675, 699, 703
 - UpdateEntry function, 704–705
- URL-related functions in Internet API, 761–762
- USB (Universal Serial Bus), 14–15
- User Datagram Protocol (UDP), 727
- user exception handler example, 322–324, 323
- user interface, **107–111**
 - active window tracking, 109
 - Alphablend function, 108
 - common controls support, 108
 - foreground switching, 109–110
 - GetGuiResources function, 108
 - GUI appearance settings, 110
 - handicapped preference settings, 109
 - HTML resources, 108
 - LockWorkStation function, 108
 - message-only windows, 107
 - mouse operation settings, 110
 - power settings, 111
 - screen saver status settings, 110
 - SendInput function, 108
 - SystemParametersInfo function, 109–111
 - task switching settings, 110
 - TrackMouseEvent function, 108
 - user interface tier technologies, 857
- user-defined security objects, 395
- UserHandleGrantAccess function, 106
- user-interface threads, 152
- users, security objects and, 393–394, 394

utilities. *See also* software

AutoChk, 63

ChkDsk, 63

Graphics Utilities on book's CD-ROM, 450

OLE/COM View utility, 944, 944

ScanDisk, 63

screen capture utilities, 93

UUIDs (universally unique identifiers), 802–803

V

V Communications System Commander software, 20

VADs (virtual address descriptors), 336–337, 337

validation functions for memory, **355–356**

variables

GetEnvironmentVariable function, 202

Hungarian notation conventions, 54–55

naming, 54–55

VARIANT data type, 942–943

versions of Windows 2000, 6, 15–16

VFAT (protected-mode FAT) file system, 68

viewing

project files for in-process COM servers, 899, 899

Threads example in Process Viewer, 179–180, 180

viewports

setting viewport extents, 486–487

setting viewport origins, 485–486

viewport versus window coordinate translations, 483–487

virtual function tables (VTBLs), 831, 988

virtual memory functions, **340–349**, 355. *See also* memory management

GetWriteWatch, 342

overview of, 340–341

ResetWriteWatch, 342

system limits and allocating memory, 355

VirtualAlloc, 341–345, 355, 359, 366

VirtualFree, 345–346

VirtualLock, 348–349

VirtualProtect, 346–347, 364–366

VirtualQuery, 343, 347–348

VirtualUnlock, 348–349

Virtual Memory Manager (VMM), **329–337**. *See also* memory management

allocating memory, 336–337, 350

committed pages, 333

committing memory, 336–337, 349–350

defined, **329–330**

demand paging, 335

free pages, 333

page directories, 333, 334

page frame databases and page tables, 332–333

- page frame states, 333–335
- page frames, 332
- page interrupts or page faults, 331–332, 334
- page policies, 335–336
- pages and processes, 335
- paging, 331–332
- pointers and movable memory, 330–331
- process working sets, 335
- reserved pages, 333
- reserving memory, 336–337, 345, 349–350
- selectors, 330–331
- Translation Look-aside Buffers (TLBs), 333, 943
- virtual address descriptors (VADs), 336–337, 337
- virtual memory and, 329–330
- virtual memory APIs, 337
- in Windows 2000 versus Windows 95/98, 329
- zeroed pages, 335–336
- virus software, 429
- Visual Basic
 - application development and, 824
 - Component Object Model and, 832, 848
 - and creating in-process COM servers, 907
 - GUIDs (globally unique identifiers) and, 826
 - IDispatch Automation clients, 948–949
 - interface inheritance, 853–855
 - interface inheritance using early binding, 852–853
 - object pooling, 872
 - as an object-oriented language, 849
 - querying for object interfaces, 851–852
 - QueryInterface function, 851–852
- Visual C++. *See also* C++ language
 - Asynchronous COM and, 960
 - Component Object Model and, 832, 848
 - dispinterfaces, 934–935, 936
 - dual-interface components, 936–937, 936
 - exception handling, 297–298, 301–302, 301
 - GUIDs (globally unique identifiers) and, 826
 - implementing IDispatch interface, 932–937
 - preparing to use Active Directory Service Interfaces, 992–993, 992, 993
 - querying for object interfaces, 851–852
- Visual Studio
 - Enterprise Setup dialog box, 687–689, 687, 688
 - OLE/COM View utility, 944, 944
- volume in ShowWave demo program, 561–563, 562
- volumes
 - GetVolumeInformation function, 76–77, 82–83
 - volume mount points, 97
 - volume quotas, 96–97
- VTBLs (virtual function tables), 831, 988

W

- wait parameter for CreateNamedPipe function, 212–214
- WaitForInputIdle function, 204
- WaitForSingleObject and WaitForMultipleObjects functions
 - in process-related operations, 203, 258–259
 - RegisterWaitForSingleObject function, 107
 - in thread-related operations, 159–162
- waiting threads, 145
- WaitNamedPipe function, 225, 226–227, 229, 261, 264
- .WAV files. *See also* ShowWave demo program; sound
 - defined, **510–511**
 - reading, 534–538
 - writing, 539–543
- Web design and management tools, 721
- Web sites. *See also* Internet
 - Installable File System (IFS) Kit, 95
 - about lightweight handlers, 983
 - Microsoft Data Access SDK, 599
 - about SIMD extensions, 592–593
 - URL-related functions in Internet API, 761–762
- WebView demo program, 775–780, 776
- Win32 APIs
 - defined, **142**
 - thread methods and, 142
- Win32 driver model, 18
- window station functions, 417
- Windows. *See* Microsoft Windows
- windows
 - active window tracking, 109
 - client or display windows, 40
 - CreateWindow function, 34–36, 339
 - CreateWindows function, 167, 168–169
 - FindWindow function, 380
 - in “Hello, World” example
 - creating application windows, 34–36
 - displaying, 36
 - registering window classes, 32–34
 - message-only windows, 107
 - messages and, 40
 - pop-up message windows, **120–128**
 - calculating window size, 126–128
 - capturing mouse and keyboard input, 125–126
 - creating CTipWnd class, 122–125
 - PopupTips function, 120–122, 120
 - ShutDown method, 125–126
 - pop-up tip windows, **116–118**
 - defined, **116–118**

- versus message boxes or dialog boxes, 116
 - tooltips, 117
- Popups demo program, **118–135**
 - CheckMenuItem and CheckMenuRadioItem functions, 131
 - CreatePopupMenu method, 130
 - defined, **118–119**
 - EnableMenuItem function, 130
 - OnInitialUpdate function, 118
 - OnKillfocusX functions, 133–135
 - OnLButtonDown and OnRButtonDown functions, 119
 - OnSize function, 118–119
 - pop-up or floating menus, 128–132, 129
 - PopupMenu method, 129–130
 - pop-ups in dialog boxes, 132–135, 132, 133
 - TrackPopupMenu function, 131–132
- registering window classes
 - in “Hello, World” example, 32–34
 - in ShowWave demo program, 566–568
- setting window captions, 631–632
- threads and window creation, 141
- versus viewports
 - setting window and viewport extents, 486–487
 - setting window and viewport origins, 485–486
 - viewport versus window coordinate translations, 483–487
- Windows DNA (Distributed interNet Applications Architecture), 856–858
- Windows.H header file
 - in “Hello, World” example, 29–30
 - in Template example, 49
- Windows Load Balancing (WLB), 870–871
- WinHello example. *See* “Hello, World” example
- WinMain procedure
 - in “Hello, World” example, **30–38**
 - arguments, 30–32
 - creating application windows, 34–36
 - displaying windows, 36
 - message handling, 36–38
 - Pascal ordering, 30–31, 56
 - registering window classes, 32–34
 - PeekMessage loops, 167
 - in ShowWave demo program, 522, 544
 - in Template example, 49–50
- WinModem card support, 8
- Winsock. *See also* Internet; sockets
 - defined, **728–729**
 - Winsock 2 API functions, **730–752**
 - asynchronous functions, 735–738
 - database-lookup functions, 732–735
 - defined, **730**
 - FindAddr demo program, 738–743, 739, 740

- gethostbyaddr, 732, 742
- gethostbyname, 732, 741–742, 745–746, 747
- HOSTS and SERVICES files, 733–735
- network byte order and host byte order conversion functions, 731–732
- SendMail demo program, 743–750, 744
- SendMail2 demo program, 751
- socket functions, 730–731
- Winsock applications, 751–752
- Winsock 2 functions for network programming, **793–799**
 - AppleTalk protocol and Winsock 2, 794–799, 798
 - defined, **793**
 - MacSock demo program, 794–799, 798
 - new features, 793–794
- WLB (Windows Load Balancing), 870–871
- WndProc procedure in “Hello, World” example, 41–44
- WNet API functions, **787–791**
 - defined, **787–788**
 - network resources and, 788–789
 - WNetDemo program, 789–791, 790
- working threads, 152
- WriteFile function, 218, 223–224, 229, 242, 261
- write-protecting list pages, 364–369
- write-through parameters in CreateNamedPipe function, 211–212
- writing
 - copy-on-write protection, 377
 - installable file system filter drivers, 95
 - to pipes, 241–243, 260–262
 - .WAV files, 539–543
 - writing file functions in Internet API, 758–759

X

XORing (exclusive-OR-ing), 442

Z

zeroed pages, 335–336

ZeroMemory command, 356

Previous	Table of Contents	Next

[HOME](#) [SUBSCRIBE](#) [SEARCH](#) [FAQ](#) [SITEMAP](#) [CONTACT US](#)

[Products](#) | [Contact Us](#) | [About Us](#) | [Privacy](#) | [Ad Info](#) | [Home](#)

Use of this site is subject to certain [Terms & Conditions](#), Copyright © 1996-2000 EarthWeb Inc. All rights reserved. Reproduction whole or in part in any form or medium without express written [permission](#) of EarthWeb is prohibited. Read EarthWeb's [privacy](#) statement.