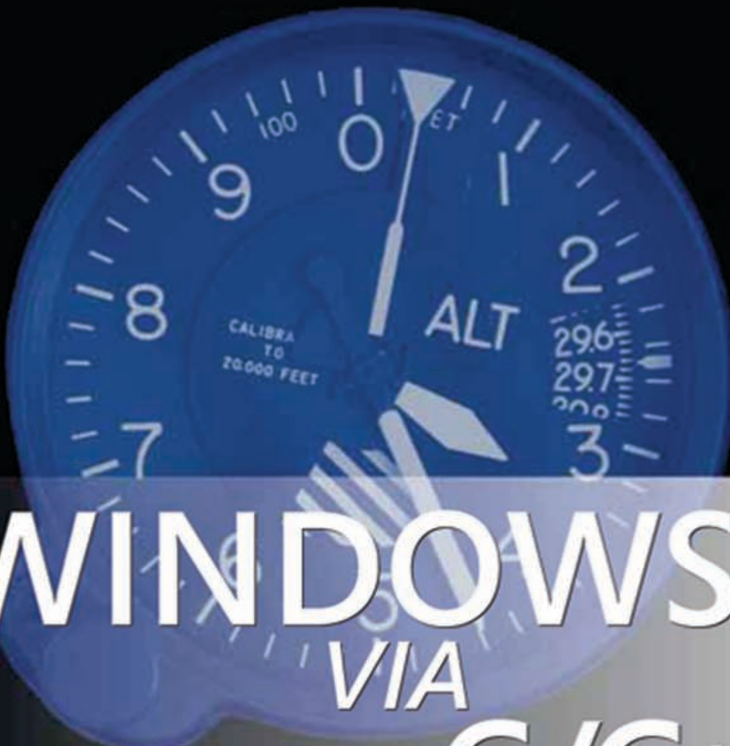
Microsoft

5

Fifth Edition



WINDOWS VIA C/C++

Jeffrey Richter
Christophe Nasarre

Wintellect
Know how.

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

Library of Congress Control Number: 2007939306

ISBN: 978-0-7356-6377-0

1 2 3 4 5 6 7 8 9 10 QGT 6 5 4 3 2 1

Printed and bound in the United States of America.

Distributed in Canada by H.B. Fenn and Company Ltd.

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

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

Acquisitions Editor: Ben Ryan
Developmental and Project Editor: Lynn Finnel
Editorial Production: Publishing.com
Technical Reviewer: Scott Seely; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Body Part No. X14-25709

Dedication

To Kristin, words cannot express how I feel about our life together.
I cherish our family and all our adventures.
I'm filled each day with love for you.

To Aidan, you have been an inspiration to me and have taught me
to play and have fun. Watching you grow up has been so rewarding
and enjoyable for me. I feel lucky to be able to
partake in your life; it has made me a better person.

To My New Baby Boy (shipping Q1 2008),
you have been wanted for so long it's hard to believe
that you're almost here. You bring completeness and balance
to our family. I look forward to playing with you,
learning who you are, and enjoying our time together.

– Jeffrey Richter

To my wife Florence, au moins cette fois c'est écrit: je t'aime Flo.

To my parents who cannot believe that learning English
with Dungeons & Dragons rules could have been so efficient.

– Christophe Nasarre

Contents at a Glance

Part V Structured Exception Handling

Part VI Appendixes

Table of Contents

Part V Structured Exception Handling

Acknowledgments

We could not have written this book without the help and technical assistance of several people. In particular, we'd like to thank the following people:

Jeffrey's Family

Jeffrey would like to thank Kristin (his wife) and Aidan (his son) for their never ending love and support.

Christophe's Family

Christophe would not have been able to write the fifth edition of this book without the love and support of Florence (his wife), the never ending curiosity of Celia (his daughter), and the purring sneak attacks of Canelle and Nougat (his cats). Now, I don't have any good excuse to not take care of you!

Technical Assistance

For writing a book like this one, personal research is not enough. We were owe great thanks to various Microsoft employees who helped us. Specifically, we'd like to thank Arun Kishan, who was able to either instantly answer weird and complicated questions or find the right person on the Windows team to provide more detailed explanations. We would also like to thank Kinshuman Kinshumann, Stephan Doll, Wedson Almeida Filho, Eric Li, Jean-Yves Poublan, Sandeep Ranade, Alan Chan, Ale Contenti, Kang Su Gatlin, Kai Hsu, Mehmet Iyigun, Ken Jung, Pavel Lebedynskiy, Paul Sliwowicz, and Landy Wang. In addition, there are those who listened to questions posted on Microsoft internal forums and shared their extensive knowledge, such as Raymond Chen, Sunggook Chue, Chris Corio, Larry Osterman, Richard Russell, Mark Russinovich, Mike Sheldon, Damien Watkins, and Junfeng Zhang. Last but not least, we would like to warmly thank John "Bugslayer" Robbins and Kenny Kerr who were kind enough to provide great feedback on chapters of this book.

Microsoft Press Editorial Team

We would like to thank Ben Ryan (acquisitions editor) for trusting a crazy French guy like Christophe, managers Lynn Finnel and Curtis Philips for their patience, Scott Seely for his constant search for technical accuracy, Roger LeBlanc for his talent in transforming Christophe's French-like English into something understandable, and Andrea Fox for her meticulous proofreading. In addition to the Redmond team, Joyanta Sen spent a lot of his personal time supporting us.

Mutual Admiration

Christophe sincerely thanks Jeffrey Richter for trusting him not to spoil the fifth edition of Jeff's book.

Jeffrey also thanks Christophe for his tireless efforts in researching, reorganizing, rewriting, and reworking the content in an attempt to reach Jeff's idea of perfection.

Introduction

Microsoft Windows is a complex operating system. It offers so many features and does so much that it's impossible for any one person to fully understand the entire system. This complexity also makes it difficult for someone to decide where to start concentrating the learning effort. Well, I always like to start at the lowest level by gaining a solid understanding of the system's basic building blocks. Once you understand the basics, it's easy to incrementally add any higher-level aspects of the system to your knowledge. So this book focuses on Windows' basic building blocks and the fundamental concepts that you must know when architecting and implementing software targeting the Windows operating system. In short, this book teaches the reader about various Windows features and how to access them via the C and C++ programming languages.

Although this book does not cover some Windows concepts—such as the Component Object Model (COM)—COM is built on top of basic building blocks such as processes, threads, memory management, DLLs, thread local storage, Unicode, and so on. If you know these basic building blocks, understanding COM is just a matter of understanding how the building blocks are used. I have great sympathy for people who attempt to jump ahead in learning COM's architecture. They have a long road ahead and are bound to have gaping holes in their knowledge, which is bound to negatively affect their code and their software development schedules.

The Microsoft .NET Framework's common language runtime (CLR) is another technology not specifically addressed in this book. (However, it is addressed in my other book: *CLR via C#,* Jeffrey Richter, Microsoft Press, 2006). However, the CLR is implemented as a COM object in a dynamic-link library (DLL) that loads in a process and uses threads to execute code that manipulates Unicode strings that are managed in memory. So again, the basic building blocks presented in this book will help developers writing managed code. In addition, by way of the CLR's Platform Invocation (P/Invoke) technology, you can call into the various Windows' APIs presented throughout this book.

So that's what this book is all about: the basic Windows building blocks that every Windows developer (at least in my opinion) should be intimately aware of. As each block is discussed, I also describe how the system uses these blocks and how your own applications can best take advantage of these blocks. In many chapters, I show you how to create building blocks of your own. These building blocks, typically implemented as generic functions or C++ classes, group a set of Windows building blocks together to create a whole that is much greater than the sum of its parts.

64-Bit Windows

Microsoft has been shipping 32-bit versions of Windows that support the *x*86 CPU architecture for many years. Today, Microsoft also offers 64-bit versions of Windows that support the *x*64 and IA-64 CPU architectures. Machines based on these 64-bit CPU architectures are fast gaining acceptance. In fact, in the very near future, it is expected that all desktop and server machines will contain 64-bit CPUs. Because of this, Microsoft has stated that Windows Server 2008 will be the last 32-bit version of Windows ever! For developers, now is the time to focus on making sure your applications run correctly on 64-bit Windows. To this end, this book includes solid coverage of what you need to know to have your applications run on 64-bit Windows (as well as 32-bit Windows).

The biggest advantage your application gets from a 64-bit address space is the ability to easily manipulate large amounts of data, because your process is no longer constrained to a 2-GB usable address space. Even if your application doesn't need all this address space, Windows itself takes advantage of the significantly larger address space (about 8 terabytes), allowing it to run faster.

Here is a quick look at what you need to know about 64-bit Windows:

- The 64-bit Windows kernel is a port of the 32-bit Windows kernel. This means that all the details and intricacies that you've learned about 32-bit Windows still apply in the 64-bit world. In fact, Microsoft has modified the 32-bit Windows source code so that it can be compiled to produce a 32-bit or a 64-bit system. They have just one source-code base, so new features and bug fixes are simultaneously applied to both systems.
- Because the kernels use the same code and underlying concepts, the Windows API is identical on both platforms. This means that you do not have to redesign or reimplement your application to work on 64-bit Windows. You can simply make slight modifications to your source code and then rebuild.
- For backward compatibility, 64-bit Windows can execute 32-bit applications. However, your application's performance will improve if the application is built as a true 64-bit application.
- Because it is so easy to port 32-bit code, there are already device drivers, tools, and applications available for 64-bit Windows. Unfortunately, Visual Studio is a native 32-bit application and Microsoft seems to be in no hurry to port it to be a native 64-bit application. However, the good news is that 32-bit Visual Studio does run quite well on 64-bit Windows; it just has a limited address space for its own data structures. And Visual Studio does allow you to debug a 64-bit application.
- There is little new for you to learn. You'll be happy to know that most data types remain 32 bits wide. These include **int**s, **DWORD**s, **LONG**s, **BOOL**s, and so on. In fact, you mostly just need to worry about pointers and handles, since they are now 64-bit values.

Because Microsoft offers so much information on how to modify your existing source code to be 64-bit ready, I will not go into those details in this book. However, I thought about 64-bit Windows as I wrote each chapter. Where appropriate, I have included information specific to 64-bit Windows. In addition, I have compiled and tested all the sample applications in this book for 64-bit Windows. So, if you follow the sample applications in this book and do as I've done, you should have no trouble creating a single source-code base that you can easily compile for 32-bit or 64-bit Windows.

What's New in the Fifth Edition

In the past, this book has been titled *Advanced Windows NT*, *Advanced Windows*, and *Programming Applications for Microsoft Windows*. In keeping with tradition, this edition of the book has gotten a new title: *Windows via C/C++*. This new title indicates that the book is for C and C++ programmers wanting to understand Windows. This new edition covers more than 170 new functions and Windows features that have been introduced in Windows XP, Windows Vista, and Windows Server 2008.

Some chapters have been completely rewritten—such as Chapter 11, which explains how the new thread pool API should be used. Existing chapters have been greatly enhanced to present new features. For example, Chapter 4 now includes coverage of User Account Control and Chapter 8 now covers new synchronization mechanisms (Interlocked Singly-Linked List, Slim Reader-Writer Locks, and condition variables).

I also give much more coverage of how the C/C++ run-time library interacts with the operating system—particularly on enhancing security as well as exception handling. Last but not least, two new chapters have been added to explain how I/O operations work and to dig into the new Windows Error Reporting system that changes the way you must think about application error reporting and application recovery.

In addition to the new organization and greater depth, I added a ton of new content. Here is a partial list of enhancements made for this edition:

New Windows Vista and Windows Server 2008 features Of course, the book would not be a true revision unless it covered new features offered in Windows XP, Windows Vista, Windows Server 2008, and the C/C++ run-time library. This edition has new information on the secure string functions, the kernel object changes (such as namespaces and boundary descriptors), thread and process attribute lists, thread and I/O priority scheduling, synchronous I/O cancellation, vectored exception handling, and more.

64-bit Windows support The text addresses 64-bit Windows-specific issues; all sample applications have been built and tested on 64-bit Windows.

Use of C++ The sample applications use C++ and require fewer lines of code, and their logic is easier to follow and understand.

Reusable code Whenever possible, I created the source code to be generic and reusable. This should allow you to take individual functions or entire C++ classes and drop them into your own applications with little or no modification. The use of C++ made reusability much easier.

The ProcessInfo utility This particular sample application from the earlier editions has been enhanced to show the process owner, command line, and UAC-related details.

The LockCop utility This sample application is new. It shows which processes are running on the system. Once you select a process, this utility lists the threads of the process and, for each, on which kind of synchronization mechanism it is blocked—with deadlocks explicitly pointed out.

API hooking I present updated C++ classes that make it trivial to hook APIs in one or all modules of a process. My code even traps run-time calls to **`LoadLibrary`** and **`GetProcAddress`** so that your API hooks are enforced.

Structured exception handling improvements I have rewritten and reorganized much of the structured exception handling material. I have more information on unhandled exceptions, and I've added coverage on customizing Windows Error Reporting to fulfill your needs.

Code Samples and System Requirements

The sample applications presented throughout this book can be downloaded from the book's companion content Web page at

http://www.Wintellect.com/Books.aspx

To build the applications, you'll need Visual Studio 2005 (or later), the Microsoft Platform SDK for Windows Vista and Windows Server 2008 (which comes with some versions of Visual Studio). In addition, to run the applications, you'll need a computer (or virtual machine) with Windows Vista (or later) installed.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to an Errata document downloadable at the following Web site:

http://www.Wintellect.com/Books.aspx

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the site just mentioned, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Windows via C/C*++ Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 2

Working with Characters and Strings

In this chapter:

With Microsoft Windows becoming more and more popular around the world, it is increasingly important that we, as developers, target the various international markets. It was once common for U.S. versions of software to ship as much as six months prior to the shipping of international versions. But increasing international support for the operating system is making it easier to produce applications for international markets and therefore is reducing the time lag between distribution of the U.S. and international versions of our software.

Windows has always offered support to help developers localize their applications. An application can get country-specific information from various functions and can examine Control Panel settings to determine the user's preferences. Windows even supports different fonts for our applications. Last but not least, in Windows Vista, Unicode 5.0 is now supported. (Read "Extend The Global Reach Of Your Applications With Unicode 5.0" at *http://msdn.microsoft.com/msdnmag/issues/07/01/Unicode/default.aspx* for a high-level presentation of Unicode 5.0.)

Buffer overrun errors (which are typical when manipulating character strings) have become a vector for security attacks against applications and even against parts of the operating system. In previous years, Microsoft put forth a lot of internal and external efforts to raise the security bar in the Windows world. The second part of this chapter presents new functions provided by Microsoft in the C run-time library. You should use these new functions to protect your code against buffer overruns when manipulating strings.

I decided to present this chapter early in the book because I highly recommend that your application always use Unicode strings and that you always manipulate these strings via the new secure string functions. As you'll see, issues regarding the secure use of Unicode strings are discussed in just about every chapter and in all the sample applications presented in this book. If you have a code base that is non-Unicode, you'll be best served by moving that code base to Unicode, as this will improve your application's execution performance as well as prepare it for localization. It will also help when interoperating with COM and the .NET Framework.

Character Encodings

The real problem with localization has always been manipulating different character sets. For years, most of us have been coding text strings as a series of single-byte characters with a zero at the end. This is second nature to us. When we call **`strlen`**, it returns the number of characters in a zero-terminated array of ANSI single-byte characters.

The problem is that some languages and writing systems (Japanese kanji being a classic example) have so many symbols in their character sets that a single byte, which offers no more than 256 different symbols at best, is just not enough. So double-byte character sets (DBCSs) were created to support these languages and writing systems. In a double-byte character set, each character in a string consists of either 1 or 2 bytes. With kanji, for example, if the first character is between 0x81 and 0x9F or between 0xE0 and 0xFC, you must look at the next byte to determine the full character in the string. Working with double-byte character sets is a programmer's nightmare because some characters are 1 byte wide and some are 2 bytes wide. Fortunately, you can forget about DBCS and take advantage of the support of Unicode strings supported by Windows functions and the C run-time library functions.

Unicode is a standard founded by Apple and Xerox in 1988. In 1991, a consortium was created to develop and promote Unicode. The consortium consists of companies such as Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys, and Xerox. (A complete and updated list of consortium members is available at *http://www.Unicode.org*.) This group of companies is responsible for maintaining the Unicode standard. The full description of Unicode can be found in *The Unicode Standard*, published by Addison-Wesley. (This book is available through *http://www.Unicode.org*.)

In Windows Vista, each Unicode character is encoded using UTF-16 (where *UTF* is an acronym for *Unicode Transformation Format*). UTF-16 encodes each character as 2 bytes (or 16 bits). In this book, when we talk about Unicode, we are always referring to UTF-16 encoding unless we state otherwise. Windows uses UTF-16 because characters from most languages used throughout the world can easily be represented via a 16-bit value, allowing programs to easily traverse a string and calculate its length. However, 16-bits is not enough to represent all characters from certain languages. For these languages, UTF-16 supports surrogates, which are a way of using 32 bits (or 4 bytes) to represent a single character. Because few applications need to represent the characters of these languages, UTF-16 is a good compromise between saving space and providing ease of coding. Note that the .NET Framework always encodes all characters and strings using UTF-16, so using UTF-16 in your Windows application will improve performance and reduce memory consumption if you need to pass characters or strings between native and managed code.

There are other UTF standards for representing characters, including the following ones:

UTF-8 UTF-8 encodes some characters as 1 byte, some characters as 2 bytes, some characters as 3 bytes, and some characters as 4 bytes. Characters with a value below 0x0080 are compressed to 1 byte, which works very well for characters used in the United States. Characters between 0x0080 and 0x07FF are converted to 2 bytes, which works well for European and Middle Eastern languages. Characters of 0x0800 and above are converted to 3 bytes, which works well for East Asian languages. Finally, surrogate pairs are written out as 4 bytes. UTF-8 is an extremely popular encoding format, but it's less efficient than UTF-16 if you encode many characters with values of 0x0800 or above.

UTF-32 UTF-32 encodes every character as 4 bytes. This encoding is useful when you want to write a simple algorithm to traverse characters (used in any language) and you don't want to have to deal with characters taking a variable number of bytes. For example, with UTF-32, you do not need to think about surrogates because every character is 4 bytes. Obviously, UTF-32 is not an efficient encoding format in terms of memory usage. Therefore, it's rarely used for saving or transmitting strings to a file or network. This encoding format is typically used inside the program itself.

Currently, Unicode code points[1] are defined for the Arabic, Chinese bopomofo, Cyrillic (Russian), Greek, Hebrew, Japanese kana, Korean hangul, and Latin (English) alphabets—called scripts—and more. Each version of Unicode brings new characters in existing scripts and even new scripts such as Phoenician (an ancient Mediterranean alphabet). A large number of punctuation marks, mathematical symbols, technical symbols, arrows, dingbats, diacritics, and other characters are also included in the character sets.

These 65,536 characters are divided into regions. Table 2-1 shows some of the regions and the characters that are assigned to them.

Table 2-1 Unicode Character Sets and Alphabets

16-Bit Code	Characters	16-Bit Code	Alphabet/Scripts
0000–007F	ASCII	0300–036F	Generic diacritical marks
0080–00FF	Latin1 characters	0400–04FF	Cyrillic
0100–017F	European Latin	0530–058F	Armenian
0180–01FF	Extended Latin	0590–05FF	Hebrew
0250–02AF	Standard phonetic	0600–06FF	Arabic
02B0–02FF	Modified letters	0900–097F	Devanagari

ANSI and Unicode Character and String Data Types

I'm sure you're aware that the C language uses the **char** data type to represent an 8-bit ANSI character. By default, when you declare a literal string in your source code, the C compiler turns the string's characters into an array of 8-bit **char** data types:

```
// An 8-bit character
char c = 'A';

// An array of 99 8-bit characters and an 8-bit terminating zero.
char szBuffer[100] = "A String";
```

Microsoft's C/C++ compiler defines a built-in data type, **wchar_t**, which represents a 16-bit Unicode (UTF-16) character. Because earlier versions of Microsoft's compiler did not offer this built-in data type, the compiler defines this data type only when the **/Zc:wchar_t** compiler switch is specified. By default, when you create a C++ project in Microsoft Visual Studio, this compiler switch is specified. We recommend that you always specify this compiler switch, as it is better to work with Unicode characters by way of the built-in primitive type understood intrinsically by the compiler.

1 A code point is the position of a symbol in a character set.

Note Prior to the built-in compiler support, a C header file defined a **wchar_t** data type as follows:

```
typedef unsigned short wchar_t;
```

Here is how you declare a Unicode character and string:

```
// A 16-bit character
wchar_t c = L'A';

// An array up to 99 16-bit characters and a 16-bit terminating zero.
wchar_t szBuffer[100] = L"A String";
```

An uppercase **L** before a literal string informs the compiler that the string should be compiled as a Unicode string. When the compiler places the string in the program's data section, it encodes each character using UTF16, interspersing zero bytes between every ASCII character in this simple case.

The Windows team at Microsoft wants to define its own data types to isolate itself a little bit from the C language. And so, the Windows header file, WinNT.h, defines the following data types:

```
typedef char     CHAR;   // An 8-bit character

typedef wchar_t WCHAR;   // A 16-bit character
```

Furthermore, the WinNT.h header file defines a bunch of convenience data types for working with pointers to characters and pointers to strings:

```
// Pointer to 8-bit character(s)
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR

// Pointer to 16-bit character(s)
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR *PCWSTR;
```

Note If you take a look at WinNT.h, you'll find the following definition:

```
typedef __nullterminated WCHAR *NWPSTR, *LPWSTR, *PWSTR;
```

The **__nullterminated** prefix is a *header annotation* that describes how types are expected to be used as function parameters and return values. In the Enterprise version of Visual Studio, you can set the Code Analysis option in the project properties. This adds the **/analyze** switch to the command line of the compiler that detects when your code calls functions in a way that breaks the semantic defined by the annotations. Notice that only Enterprise versions of the compiler support this **/analyze** switch. To keep the code more readable in this book, the header annotations are removed. You should read the "Header Annotations" documentation on MSDN at *http://msdn2.microsoft.com/En-US/library/aa383701.aspx* for more details about the header annotations language.

In your own source code, it doesn't matter which data type you use, but I'd recommend you try to be consistent to improve maintainability in your code. Personally, as a Windows programmer, I always use the Windows data types because the data types match up with the MSDN documentation, making things easier for everyone reading the code.

It is possible to write your source code so that it can be compiled using ANSI or Unicode characters and strings. In the WinNT.h header file, the following types and macros are defined:

```
#ifdef UNICODE

typedef WCHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) quote         // r_winnt

#define __TEXT(quote) L##quote

#else

typedef CHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST CHAR *PCTSTR;
#define __TEXT(quote) quote

#endif

#define   TEXT(quote) __TEXT(quote)
```

These types and macros (plus a few less commonly used ones that I do not show here) are used to create source code that can be compiled using either ANSI or Unicode chacters and strings, for example:

```
// If UNICODE defined, a 16-bit character; else an 8-bit character
TCHAR c = TEXT('A');

// If UNICODE defined, an array of 16-bit characters; else 8-bit characters
TCHAR szBuffer[100] = TEXT("A String");
```

Unicode and ANSI Functions in Windows

Since Windows NT, all Windows versions are built from the ground up using Unicode. That is, all the core functions for creating windows, displaying text, performing string manipulations, and so forth require Unicode strings. If you call any Windows function passing it an ANSI string (a string of 1-byte characters), the function first converts the string to Unicode and then passes the Unicode string to the operating system. If you are expecting ANSI strings back from a function, the system converts the Unicode string to an ANSI string before returning to your application. All these conversions occur invisibly to you. Of course, there is time and memory overhead involved for the system to carry out all these string conversions.

When Windows exposes a function that takes a string as a parameter, two versions of the same function are usually provided—for example, a **`CreateWindowEx`** that accepts Unicode strings and a second **CreateWindowEx** that accepts ANSI strings. This is true, but the two functions are actually prototyped as follows:

```
HWND WINAPI CreateWindowExW(
   DWORD dwExStyle,
   PCWSTR pClassName,    // A Unicode string
   PCWSTR pWindowName,   // A Unicode string
   DWORD dwStyle,
   int X,
   int Y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU hMenu,
   HINSTANCE hInstance,
   PVOID pParam);

HWND WINAPI CreateWindowExA(
   DWORD dwExStyle,
   PCSTR pClassName,    // An ANSI string
   PCSTR pWindowName,   // An ANSI string
   DWORD dwStyle,
   int X,
   int Y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU hMenu,
   HINSTANCE hInstance,
   PVOID pParam);
```

CreateWindowExW is the version that accepts Unicode strings. The uppercase *W* at the end of the function name stands for *wide*. Unicode characters are 16 bits wide, so they are frequently referred to as wide characters. The uppercase *A* at the end of **CreateWindowExA** indicates that the function accepts ANSI character strings.

But usually we just include a call to **CreateWindowEx** in our code and don't directly call either **CreateWindowExW** or **CreateWindowExA**. In WinUser.h, **CreateWindowEx** is actually a macro defined as

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif
```

Whether or not **UNICODE** is defined when you compile your source code module determines which version of **CreateWindowEx** is called. When you create a new project with Visual Studio, it defines **UNICODE** by default. So, by default, any calls you make to **CreateWindowEx** expand the macro to call **CreateWindowExW**—the Unicode version of **CreateWindowEx**.

Under Windows Vista, Microsoft's source code for **CreateWindowExA** is simply a translation layer that allocates memory to convert ANSI strings to Unicode strings; the code then calls **CreateWindowExW**, passing the converted strings. When **CreateWindowExW** returns, **CreateWindowExA** frees its memory buffers and returns the window handle to you. So, for functions that fill buffers with strings, the system must convert from Unicode to non-Unicode equivalents before your application can process the string. Because the system must perform all these conversions, your application requires more memory and runs slower. You can make your application perform more efficiently by developing your application using Unicode from the start. Also, Windows has been known to have some bugs in these translation functions, so avoiding them also eliminates some potential bugs.

If you're creating dynamic-link libraries (DLLs) that other software developers will use, consider using this technique: supply two exported functions in the DLL—an ANSI version and a Unicode version. In the ANSI version, simply allocate memory, perform the necessary string conversions, and call the Unicode version of the function. I'll demonstrate this process later in this chapter in "Exporting ANSI and Unicode DLL Functions" on page 29.

Certain functions in the Windows API, such as **WinExec** and **OpenFile**, exist solely for backward compatibility with 16-bit Windows programs that supported only ANSI strings. These methods should be avoided by today's programs. You should replace any calls to **WinExec** and **OpenFile** with calls to the **CreateProcess** and **CreateFile** functions. Internally, the old functions call the new functions anyway. The big problem with the old functions is that they don't accept Unicode strings and they typically offer fewer features. When you call these functions, you must pass ANSI strings. On Windows Vista, most non-obsolete functions have both Unicode and ANSI versions. However, Microsoft has started to get into the habit of producing some functions offering only Unicode versions—for example, **ReadDirectoryChangesW** and **CreateProcessWithLogonW**.

When Microsoft was porting COM from 16-bit Windows to Win32, an executive decision was made that all COM interface methods requiring a string would accept only Unicode strings. This was a great decision because COM is typically used to allow different components to talk to each other and Unicode is the richest way to pass strings around. Using Unicode throughout your application makes interacting with COM easier too.

Finally, when the resource compiler compiles all your resources, the output file is a binary representation of the resources. String values in your resources (string tables, dialog box templates, menus, and so on) are always written as Unicode strings. Under Windows Vista, the system performs internal conversions if your application doesn't define the **UNICODE** macro. For example, if **UNICODE** is not defined when you compile your source module, a call to **LoadString** will actually call the **LoadStringA** function. **LoadStringA** will then read the Unicode string from your resources and convert the string to ANSI. The ANSI representation of the string will be returned from the function to your application.

Unicode and ANSI Functions in the C Run-Time Library

Like the Windows functions, the C run-time library offers one set of functions to manipulate ANSI characters and strings and another set of functions to manipulate Unicode characters and strings. However, unlike Windows, the ANSI functions do the work; they do not translate the strings to Unicode and then call the Unicode version of the functions internally. And, of course, the Unicode versions do the work themselves too; they do not internally call the ANSI versions.

An example of a C run-time function that returns the length of an ANSI string is **strlen**, and an example of an equivalent C run-time function that returns the length of a Unicode string is **wcslen**.

Both of these functions are prototyped in String.h. To write source code that can be compiled for either ANSI or Unicode, you must also include TChar.h, which defines the following macro:

```
#ifdef _UNICODE
#define _tcslen     wcslen
#else
#define _tcslen     strlen
#endif
```

Now, in your code, you should call **`_tcslen`**. If **`_UNICODE`** is defined, it expands to **`wcslen`**; otherwise, it expands to **`strlen`**. By default, when you create a new C++ project in Visual Studio, **`_UNICODE`** is defined (just like **`UNICODE`** is defined). The C run-time library always prefixes identifiers that are not part of the C++ standard with underscores, while the Windows team does not do this. So, in your applications you'll want to make sure that both **`UNICODE`** and **`_UNICODE`** are defined or that neither is defined. Appendix A, "The Build Environment," will describe the details of the CmnHdr.h header file used by all the code samples of this book to avoid this kind of problem.

Secure String Functions in the C Run-Time Library

Any function that modifies a string exposes a potential danger: if the destination string buffer is not large enough to contain the resulting string, memory corruption occurs. Here is an example:

```
// The following puts 4 characters in a
// 3-character buffer, resulting in memory corruption
WCHAR szBuffer[3] = L"";
wcscpy(szBuffer, L"abc"); // The terminating 0 is a character too!
```

The problem with the **`strcpy`** and **`wcscpy`** functions (and most other string manipulation functions) is that they do not accept an argument specifying the maximum size of the buffer, and therefore, the function doesn't know that it is corrupting memory. Because the function doesn't know that it is corrupting memory, it can't report an error back to your code, and therefore, you have no way of knowing that memory was corrupted. And, of course, it would be best if the function just failed without corrupting any memory at all.

This kind of misbehavior has been heavily exploited by malware in the past. Microsoft is now providing a set of new functions that replace the unsafe string manipulation functions (such as **`wcscat`**, which was shown earlier) provided by the C run-time library that many of us have grown to know and love over the years. To write safe code, you should no longer use any of the familiar C run-time functions that modify a string. (Functions such as **`strlen`**, **`wcslen`**, and **`_tcslen`** are OK, however, because they do not attempt to modify the string passed to them even though they assume that the string is **`0`** terminated, which might not be the case.) Instead, you should take advantage of the new secure string functions defined by Microsoft's StrSafe.h file.

> **Note** Internally, Microsoft has retrofitted its ATL and MFC class libraries to use the new safe string functions, and therefore, if you use these libraries, rebuilding your application to the new versions is all you have to do to make your application more secure.

Because this book is not dedicated to C/C++ programming, for a detailed usage of this library, you should take a look at the following sources of information:

- The MSDN Magazine article "Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries" by Martyn Lovell, located at *http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx*
- The Martyn Lovell video presentation on Channel9, located at *http://channel9.msdn.com/Showpost.aspx?postid=186406*
- The secure strings topic on MSDN Online, located at *http://msdn2.microsoft.com/en-us/library/ms647466.aspx*
- The list of all C run-time secured replacement functions on MSDN Online, which you can find at *http://msdn2.microsoft.com/en-us/library/wd3wzwts(VS.80).aspx*

However, it is worth discussing a couple of details in this chapter. I'll start by describing the patterns employed by the new functions. Next, I'll mention the pitfalls you might encounter if you are following the migration path from legacy functions to their corresponding secure versions, like using **_tcscpy_s** instead of **_tcscpy**. Then I'll show you in which case it might be more interesting to call the new **StringC*** functions instead.

Introducing the New Secure String Functions

When you include StrSafe.h, String.h is also included and the existing string manipulation functions of the C run-time library, such as those behind the **_tcscpy** macro, are flagged with obsolete warnings during compilation. Note that the inclusion of StrSafe.h must appear after all other include files in your source code. I recommend that you use the compilation warnings to explicitly replace all the occurrences of the deprecated functions by their safer substitutes—thinking each time about possible buffer overflow and, if it is not possible to recover, how to gracefully terminate the application.

Each existing function, like **_tcscpy** or **_tcscat**, has a corresponding new function that starts with the same name that ends with the **_s** (for *secure*) suffix. All these new functions share common characteristics that require explanation. Let's start by examining their prototypes in the following code snippet, which shows the side-by-side definitions of two usual string functions:

```
PTSTR   _tcscpy  (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscpy_s(PTSTR strDestination, size_t numberOfCharacters,
   PCTSTR strSource);

PTSTR   _tcscat  (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscat_s(PTSTR strDestination, size_t numberOfcharacters,
   PCTSTR strSource);
```

When a writable buffer is passed as a parameter, its size must also be provided. This value is expected in the character count, which is easily computed by using the **_countof** macro (defined in stdlib.h) on your buffer.

All of the secure (**_s**) functions validate their arguments as the first thing they do. Checks are performed to make sure that pointers are not **NULL**, that integers are within a valid range, that enumeration values are valid, and that buffers are large enough to hold the resulting data. If any of these checks fail, the functions set the thread-local C run-time variable **errno** and the function returns an **errno_t** value to indicate success or failure. However, these functions don't actually return; instead, in a debug build, they display a user-unfriendly assertion dialog box similar to that shown in Figure 2-1. Then your application is terminated. The release builds directly auto-terminate.

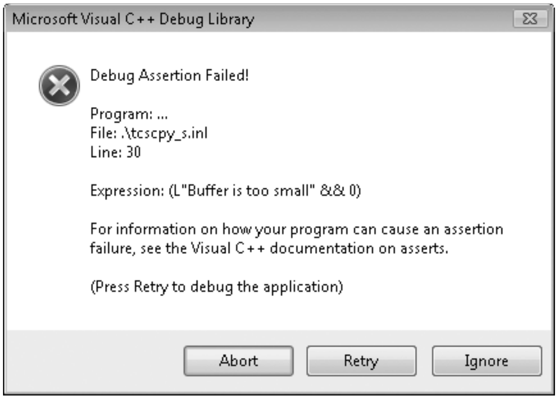


Figure 2-1 Assertion dialog box displayed when an error occurs

The C run time actually allows you to provide a function of your own, which it will call when it detects an invalid parameter. Then, in this function, you can log the failure, attach a debugger, or do whatever you like. To enable this, you must first define a function that matches the following prototype:

```
void InvalidParameterHandler(PCTSTR expression, PCTSTR function,
   PCTSTR file, unsigned int line, uintptr_t /*pReserved*/);
```

The **expression** parameter describes the failed expectation in the C run-time implementation code, such as **(L"Buffer is too small" && 0)**. As you can see, this is not very user friendly and should not be shown to the end user. This comment also applies to the next three parameters because **function**, **file**, and **line** describe the function name, the source code file, and the source code line number where the error occurred, respectively.

> **Note** All these arguments will have a value of **NULL** if **DEBUG** is not defined. So this handler is valuable for logging errors only when testing debug builds. In a release build, you could replace the assertion dialog box with a more user-friendly message explaining that an unexpected error occurred that requires the application to shut down—maybe with specific logging behavior or an application restart. If its memory state is corrupted, your application execution should stop. However, it is recommended that you wait for the **errno_t** check to decide whether the error is recoverable or not.

The next step is to register this handler by calling **_set_invalid_parameter_handler**. However, this step is not enough because the assertion dialog box will still appear. You need to call **_CrtSetReportMode(_CRT_ASSERT, 0);** at the beginning of your application, disabling all assertion dialog boxes that could be triggered by the C run time.

Now, when you call one of the legacy replacement functions defined in String.h, you are able to check the returned **errno_t** value to understand what happened. Only the value **S_OK** means that the call was successful. The other possible return values found in errno.h, such as **EINVAL**, are for invalid arguments such as **NULL** pointers.

Let's take an example of a string that is copied into a buffer that is too small for one character:

```
TCHAR szBefore[5] = {
   TEXT('B'), TEXT('B'), TEXT('B'), TEXT('B'), '\0'
};

TCHAR szBuffer[10] = {
   TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'),
   TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), '\0'
};

TCHAR szAfter[5] = {
   TEXT('A'), TEXT('A'), TEXT('A'), TEXT('A'), '\0'
};

errno_t result = _tcscpy_s(szBuffer, _countof(szBuffer),
   TEXT("0123456789"));
```

Just before the call to **_tcscpy_s**, each variable has the content shown in Figure 2-2.

Figure 2-2 Variable state before the *_tcscpy_s* call

Because the "1234567890" string to be copied into **szBuffer** has exactly the same 10-character size as the buffer, there is not enough room to copy the terminating **'\0'** character. You might expect that the value of **result** is now **STRUNCATE** and the last character **'9'** has not been copied, but this is not the case. **ERANGE** is returned, and the state of each variable is shown in Figure 2-3.

Figure 2-3 Variable state after the *_tcscpy_s* call

There is one side effect that you don't see unless you take a look at the memory behind **szBuffer**, as shown in Figure 2-4.

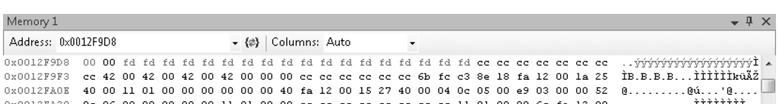


Figure 2-4 Content of *szBuffer* memory after a failed call

The first character of **szBuffer** has been set to **'\0'**, and all other bytes now contain the value **0xfd**. So the resulting string has been truncated to an empty string and the remaining bytes of the buffer have been set to a filler value (**0xfd**).

> **Note** If you wonder why the memory after all the variables have been defined is filled up with the **0xcc** value in Figure 2-4, the answer is in the result of the compiler implementation of the run-time checks (**/RTCs**, **/RTCu**, or **/RTC1**) that automatically detect buffer overrun at run time. If you compile your code without these **/RTCx** flags, the memory view will show all **sz*** variables side by side. But remember that your builds should always be compiled with these run-time checks to detect any remaining buffer overrun early in the development cycle.

How to Get More Control When Performing String Operations

In addition to the new secure string functions, the C run-time library has some new functions that provide more control when performing string manipulations. For example, you can control the filler values or how truncation is performed. Naturally, the C run time offers both ANSI (A) versions of the functions as well as Unicode (W) versions of the functions. Here are the prototypes for some of these functions (and many more exist that are not shown here):

```
HRESULT StringCchCat(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCatEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchCopy(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCopyEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchPrintf(PTSTR pszDest, size_t cchDest,
   PCTSTR pszFormat, ...);
HRESULT StringCchPrintfEx(PTSTR pszDest, size_t cchDest,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags,
   PCTSTR pszFormat,...);
```

You'll notice that all the methods shown have "Cch" in their name. This stands for *Count of characters*, and you'll typically use the **_countof** macro to get this value. There is also a set of functions that have "Cb" in their name, such as **StringCbCat(Ex)**, **StringCbCopy(Ex)**, and **StringCbPrintf(Ex)**. These functions expect that the size argument is in count of bytes instead of count of characters. You'll typically use the **sizeof** operator to get this value.

All these functions return an **HRESULT** with one of the values shown in Table 2-2.

Table 2-2 HRESULT Values for Safe String Functions

HRESULT Value	Description
S_OK	Success. The destination buffer contains the source string and is terminated by **'\0'**.
STRSAFE_E_INVALID_PARAMETER	Failure. The **NULL** value has been passed as a parameter.
STRSAFE_E_INSUFFICIENT_BUFFER	Failure. The given destination buffer was too small to contain the entire source string.

Unlike the secure (**_s** suffixed) functions, when a buffer is too small, these functions do perform truncation. You can detect such a situation when **STRSAFE_E_INSUFFICIENT_BUFFER** is returned. As you can see in StrSafe.h, the value of this code is **0x8007007a** and is treated as a failure by

SUCCEEDED/FAILED macros. However, in that case, the part of the source buffer that could fit into the destination writable buffer has been copied and the last available character is set to **'\0'**. So, in the previous example, **szBuffer** would contain the string "012345678" if **StringCchCopy** is used instead of **_tcscpy_s**. Notice that the truncation feature might or might not be what you need, depending on what you are trying to achieve, and this is why it is treated as a failure (by default). For example, in the case of a path that you are building by concatenating different pieces of information, a truncated result is unusable. If you are building a message for user feedback, this could be acceptable. It's up to you to decide how to handle a truncated result.

Last but not least, you'll notice that an extended (**Ex**) version exists for many of the functions shown earlier. These extended versions take three additional parameters, which are described in Table 2-3.

Table 2-3 Extended Version Parameters

Parameters and Values	Description
size_t* pcchRemaining	Pointer to a variable that indicates the number of unused characters in the destination buffer. The copied terminating **'\0'** character is not counted. For example, if one character is copied into a buffer that is 10 characters wide, 9 is returned even though you won't be able to use more than 8 characters without truncation. If **pcchRemaining** is **NULL**, the count is not returned.
LPTSTR* ppszDestEnd	If **ppszDestEnd** is non-**NULL**, it points to the terminating **'\0'** character at the end of the string contained by the destination buffer.
DWORD dwFlags	One or more of the following values separated by '|'.
STRSAFE_FILL_BEHIND_NULL	If the function succeeds, the low byte of **dwFlags** is used to fill the rest of the destination buffer, just after the terminating **'\0'** character. (See the comment about **STRSAFE_FILL_BYTE** just after this table for more details.)
STRSAFE_IGNORE_NULLS	Treats **NULL** string pointers like empty strings **(TEXT(""))**.
STRSAFE_FILL_ON_FAILURE	If the function fails, the low byte of **dwFlags** is used to fill the entire destination buffer except the first **'\0'** character used to set an empty string result. (See the comment about **STRSAFE_FILL_BYTE** just after this table for more details.) In the case of a **STRSAFE_E_INSUFFICIENT_BUFFER** failure, any character in the string being returned is replaced by the filler byte value.
STRSAFE_NULL_ON_FAILURE	If the function fails, the first character of the destination buffer is set to **'\0'** to define an empty string **(TEXT(""))**. In the case of a **STRSAFE_E_INSUFFICIENT_BUFFER** failure, any truncated string is overwritten.
STRSAFE_NO_TRUNCATION	As in the case of **STRSAFE_NULL_ON_FAILURE**, if the function fails, the destination buffer is set to an empty string **(TEXT(""))**. In the case of a **STRSAFE_E_INSUFFICIENT_BUFFER** failure, any truncated string is overwritten.

> **Note** Even if **STRSAFE_NO_TRUNCATION** is used as a flag, the characters of the source string are still copied, up to the last available character of the destination buffer. Then both the first and the last characters of the destination buffer are set to **'\0'**. This is not really important except if, for security purposes, you don't want to keep garbage data.

There is a last detail to mention that is related to the remark that you read at the bottom of page 21. In Figure 2-4, the **0xfd** value is used to replace all the characters after the **'\0'**, up to the end of the destination buffer. With the **Ex** version of these functions, you can choose whether you want this expensive filling operation (especially if the destination buffer is large) to occur and with which byte value. If you add **STRSAFE_FILL_BEHIND_NULL** to **dwFlag**, the remaining characters are set to **'\0'**. When you replace **STRSAFE_FILL_BEHIND_NULL** with the **STRSAFE_FILL_BYTE** macro, the given byte value is used to fill up the remaining values of the destination buffer.

Windows String Functions

Windows also offers various functions for manipulating strings. Many of these functions, such as **lstrcat** and **lstrcpy**, are now deprecated because they do not detect buffer overrun problems. Also, the ShlwApi.h file defines a number of handy string functions that format operating system–related numeric values, such as **StrFormatKBSize** and **StrFormatByteSize**. See *http://msdn2.microsoft.com/en-us/library/ms538658.aspx* for a description of shell string handling functions.

It is common to want to compare strings for equality or for sorting. The best functions to use for this are **CompareString(Ex)** and **CompareStringOrdinal**. You use **CompareString(Ex)** to compare strings that will be presented to the user in a linguistically correct manner. Here is the prototype of the **CompareString** function:

```
int CompareString(
   LCID locale,
   DWORD dwCmdFlags,
   PCTSTR pString1,
   int cch1,
   PCTSTR pString2,
   int cch2);
```

This function compares two strings. The first parameter to **CompareString** specifies a locale ID (LCID), a 32-bit value that identifies a particular language. **CompareString** uses this LCID to compare the two strings by checking the meaning of the characters as they apply to a particular language. A linguistically correct comparison produces results much more meaningful to an end user. However, this type of comparison is slower than doing an ordinal comparison. You can get the locale ID of the calling thread by calling the Windows **GetThreadLocale** function:

```
LCID GetThreadLocale();
```

The second parameter of **CompareString** identifies flags that modify the method used by the function to compare the two strings. Table 2-4 shows the possible flags.

Table 2-4 Flags Used by the CompareString Function

Flag	Meaning
NORM_IGNORECASE **LINGUISTIC_IGNORECASE**	Ignore case difference.
NORM_IGNOREKANATYPE	Do not differentiate between hiragana and katakana characters.
NORM_IGNORENONSPACE **LINGUISTIC_IGNOREDIACRITIC**	Ignore nonspacing characters.
NORM_IGNORESYMBOLS	Ignore symbols.
NORM_IGNOREWIDTH	Do not differentiate between a single-byte character and the same character as a double-byte character.
SORT_STRINGSORT	Treat punctuation the same as symbols.

The remaining four parameters of **CompareString** specify the two strings and their respective lengths in characters (not in bytes). If you pass negative values for the **cch1** parameter, the function assumes that the **pString1** string is zero-terminated and calculates the length of the string. This also is true for the **cch2** parameter with respect to the **pString2** string. If you need more advanced linguistic options, you should take a look at the **CompareStringEx** functions.

To compare strings that are used for programmatic strings (such as pathnames, registry keys/values, XML elements/attributes, and so on), use **CompareStringOrdinal**:

```
int CompareStringOrdinal(
  PCWSTR pString1,
  int cchCount1,
  PCWSTR pString2,
  int cchCount2,
  BOOL bIgnoreCase);
```

This function performs a code-point comparison without regard to the locale, and therefore it is fast. And because programmatic strings are not typically shown to an end user, this function makes the most sense. Notice that only Unicode strings are expected by this function.

The **CompareString** and **CompareStringOrdinal** functions' return values are unlike the return values you get back from the C run-time library's ***cmp** string comparison functions. **CompareString**(**Ordinal**) returns **0** to indicate failure, **CSTR_LESS_THAN** (defined as **1**) to indicate that **pString1** is less than **pString2**, **CSTR_EQUAL** (defined as **2**) to indicate that **pString1** is equal to **pString2**, and **CSTR_GREATER_THAN** (defined as **3**) to indicate that **pString1** is greater than **pString2**. To make things slightly more convenient, if the functions succeed, you can subtract **2** from the return value to make the result consistent with the result of the C run-time library functions (**-1**, **0**, and **+1**).

Why You Should Use Unicode

When developing an application, we highly recommend that you use Unicode characters and strings. Here are some of the reasons why:

- Unicode makes it easy for you to localize your application to world markets.
- Unicode allows you to distribute a single binary (.exe or DLL) file that supports all languages.
- Unicode improves the efficiency of your application because the code performs faster and uses less memory. Windows internally does everything with Unicode characters and strings, so when you pass an ANSI character or string, Windows must allocate memory and convert the ANSI character or string to its Unicode equivalent.
- Using Unicode ensures that your application can easily call all nondeprecated Windows functions, as some Windows functions offer versions that operate only on Unicode characters and strings.
- Using Unicode ensures that your code easily integrates with COM (which requires the use of Unicode characters and strings).
- Using Unicode ensures that your code easily integrates with the .NET Framework (which also requires the use of Unicode characters and strings).
- Using Unicode ensures that your code easily manipulates your own resources (where strings are always persisted as Unicode).

How We Recommend Working with Characters and Strings

Based on what you've read in this chapter, the first part of this section summarizes what you should always keep in mind when developing your code. The second part of the section provides tips and tricks for better Unicode and ANSI string manipulations. It's a good idea to start converting your application to be Unicode-ready even if you don't plan to use Unicode right away. Here are the basic guidelines you should follow:

- Start thinking of text strings as arrays of characters, not as arrays of **chars** or arrays of bytes.
- Use generic data types (such as **TCHAR/PTSTR**) for text characters and strings.
- Use explicit data types (such as **BYTE** and **PBYTE**) for bytes, byte pointers, and data buffers.
- Use the **TEXT** or **_T** macro for literal characters and strings, but avoid mixing both for the sake of consistency and for better readability.
- Perform global replaces. (For example, replace **PSTR** with **PTSTR**.)
- Modify string arithmetic problems. For example, functions usually expect you to pass a buffer's size in characters, not bytes. This means you should pass **_countof(szBuffer)** instead of **sizeof(szBuffer)**. Also, if you need to allocate a block of memory for a string and you have the number of characters in the string, remember that you allocate memory in bytes. This means that you must call **malloc(nCharacters * sizeof(TCHAR))** and not call **malloc(nCharacters)**. Of all the guidelines I've just listed, this is the most difficult one to remember, and the compiler offers no warnings or errors if you make a mistake. This is a good opportunity to define your own macros, such as the following:

```
#define chmalloc(nCharacters)  (TCHAR*)malloc(nCharacters * sizeof(TCHAR)).
```

- Avoid **printf** family functions, especially by using **%s** and **%S** field types to convert ANSI to Unicode strings and vice versa. Use **MultiByteToWideChar** and **WideCharToMultiByte** instead, as shown in "Translating Strings Between Unicode and ANSI" below.
- Always specify both **UNICODE** and **_UNICODE** symbols or neither of them.

In terms of string manipulation functions, here are the basic guidelines that you should follow:

- Always work with safe string manipulation functions such as those suffixed with **_s** or prefixed with **StringCch**. Use the latter for explicit truncation handling, but prefer the former otherwise.
- Don't use the unsafe C run-time library string manipulation functions. (See the previous recommendation.) In a more general way, don't use or implement any buffer manipulation routine that would not take the size of the destination buffer as a parameter. The C run-time library provides a replacement for buffer manipulation functions such as **memcpy_s**, **memmove_s**, **wmemcpy_s**, or **wmemmove_s**. All these methods are available when the **__STDC_WANT_SECURE_LIB__** symbol is defined, which is the case by default in CrtDefs.h. So don't undefine **__STDC_WANT_SECURE_LIB__**.
- Take advantage of the **/GS** (*http://msdn2.microsoft.com/en-us/library/aa290051(VS.71).aspx*) and **/RTCs** compiler flags to automatically detect buffer overruns.
- Don't use Kernel32 methods for string manipulation such as **lstrcat** and **lstrcpy**.
- There are two kinds of strings that we compare in our code. Programmatic strings are file names, paths, XML elements and attributes, and registry keys/values. For these, use **CompareStringOrdinal**, as it is very fast and does not take the user's locale into account. This is good because these strings remain the same no matter where your application is running in the world. User strings are typically strings that appear in the user interface. For these, call **CompareString(Ex)**, as it takes the locale into account when comparing strings.

You don't have a choice: as a professional developer, you can't write code based on unsafe buffer manipulation functions. And this is the reason why all the code in this book relies on these safer functions from the C run-time library.

Translating Strings Between Unicode and ANSI

You use the Windows function **MultiByteToWideChar** to convert multibyte-character strings to wide-character strings. **MultiByteToWideChar** is shown here:

```
int MultiByteToWideChar(
   UINT uCodePage,
   DWORD dwFlags,
   PCSTR pMultiByteStr,
   int cbMultiByte,
   PWSTR pWideCharStr,
   int cchWideChar);
```

The **uCodePage** parameter identifies a code page number that is associated with the multibyte string. The **dwFlags** parameter allows you to specify an additional control that affects characters with diacritical marks such as accents. Usually the flags aren't used, and **0** is passed in the **dwFlags** parameter (For more details about the possible values for this flag, read the MSDN online help at *http://msdn2.microsoft.com/en-us/library/ms776413.aspx*.) The **pMultiByteStr** parameter specifies the string to be converted, and the **cbMultiByte** parameter indicates the length (in bytes) of the string. The function automatically determines the length of the source string if you pass −1 for the **cbMultiByte** parameter.

The Unicode version of the string resulting from the conversion is written to the buffer located in memory at the address specified by the **pWideCharStr** parameter. You must specify the maximum size of this buffer (in characters) in the **cchWideChar** parameter. If you call **MultiByteToWideChar**, passing **0** for the **cchWideChar** parameter, the function doesn't perform the conversion and instead returns the number of wide characters (including the terminating **'\0'** character) that the buffer must provide for the conversion to succeed. Typically, you convert a multibyte-character string to its Unicode equivalent by performing the following steps:

1. Call **MultiByteToWideChar**, passing **NULL** for the **pWideCharStr** parameter and **0** for the **cchWideChar** parameter and −1 for the **cbMultiByte** parameter.
2. Allocate a block of memory large enough to hold the converted Unicode string. This size is computed based on the value returned by the previous call to **MultiByteToWideChar** multiplied by **sizeof(wchar_t)**.
3. Call **MultiByteToWideChar** again, this time passing the address of the buffer as the **pWideCharStr** parameter and passing the size computed based on the value returned by the first call to **MultiByteToWideChar** multiplied by **sizeof(wchar_t)** as the **cchWideChar** parameter.
4. Use the converted string.
5. Free the memory block occupying the Unicode string.

The function **WideCharToMultiByte** converts a wide-character string to its multibyte-string equivalent, as shown here:

```
int WideCharToMultiByte(
   UINT uCodePage,
   DWORD dwFlags,
   PCWSTR pWideCharStr,
   int cchWideChar,
   PSTR pMultiByteStr,
   int cbMultiByte,
   PCSTR pDefaultChar,
   PBOOL pfUsedDefaultChar);
```

This function is similar to the **MultiByteToWideChar** function. Again, the **uCodePage** parameter identifies the code page to be associated with the newly converted string. The **dwFlags** parameter allows you to specify additional control over the conversion. The flags affect characters with diacritical marks and characters that the system is unable to convert. Usually, you won't need this degree of control over the conversion, and you'll pass **0** for the **dwFlags** parameter.

The **pWideCharStr** parameter specifies the address in memory of the string to be converted, and the **cchWideChar** parameter indicates the length (in characters) of this string. The function determines the length of the source string if you pass −1 for the **cchWideChar** parameter.

The multibyte version of the string resulting from the conversion is written to the buffer indicated by the **pMultiByteStr** parameter. You must specify the maximum size of this buffer (in bytes) in the **cbMultiByte** parameter. Passing **0** as the **cbMultiByte** parameter of the **WideCharToMultiByte** function causes the function to return the size required by the destination buffer. You'll typically convert a wide-character string to a multibyte-character string using a sequence of events similar to those discussed when converting a multibyte string to a wide-character string, except that the return value is directly the number of bytes required for the conversion to succeed.

You'll notice that the **WideCharToMultiByte** function accepts two parameters more than the **MultiByteToWideChar** function: **pDefaultChar** and **pfUsedDefaultChar**. These parameters are used by the **WideCharToMultiByte** function only if it comes across a wide character that doesn't have a representation in the code page identified by the **uCodePage** parameter. If the wide character cannot be converted, the function uses the character pointed to by the **pDefaultChar** parameter. If this parameter is **NULL**, which is most common, the function uses a system default character. This default character is usually a question mark. This is dangerous for filenames because the question mark is a wildcard character.

The **pfUsedDefaultChar** parameter points to a Boolean variable that the function sets to **TRUE** if at least one character in the wide-character string could not be converted to its multibyte equivalent. The function sets the variable to **FALSE** if all the characters convert successfully. You can test this variable after the function returns to check whether the wide-character string was converted successfully. Again, you usually pass **NULL** for this parameter.

For a more complete description of how to use these functions, please refer to the Platform SDK documentation.

Exporting ANSI and Unicode DLL Functions

You could use these two functions to easily create both Unicode and ANSI versions of functions. For example, you might have a dynamic-link library containing a function that reverses all the characters in a string. You could write the Unicode version of the function as shown here:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength) {

   // Get a pointer to the last character in the string.
   PWSTR pEndOfStr = pWideCharStr + wcsnlen_s(pWideCharStr , cchLength) - 1;
   wchar_t cCharT;
   // Repeat until we reach the center character in the string.
   while (pWideCharStr < pEndOfStr) {
      // Save a character in a temporary variable.
      cCharT = *pWideCharStr;

      // Put the last character in the first character.
      *pWideCharStr = *pEndOfStr;

      // Put the temporary character in the last character.
      *pEndOfStr = cCharT;

      // Move in one character from the left.
      pWideCharStr++;

      // Move in one character from the right.
      pEndOfStr--;
   }

   // The string is reversed; return success.
   return(TRUE);
}
```

And you could write the ANSI version of the function so that it doesn't perform the actual work of reversing the string at all. Instead, you could write the ANSI version so that it converts the ANSI string to Unicode, passes the Unicode string to the **StringReverseW** function, and then converts the reversed string back to ANSI. The function would look like this:

```
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength) {
   PWSTR pWideCharStr;
   int nLenOfWideCharStr;
   BOOL fOk = FALSE;

   // Calculate the number of characters needed to hold
   // the wide-character version of the string.
   nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
      pMultiByteStr, cchLength, NULL, 0);

   // Allocate memory from the process' default heap to
   // accommodate the size of the wide-character string.
   // Don't forget that MultiByteToWideChar returns the
   // number of characters, not the number of bytes, so
   // you must multiply by the size of a wide character.
   pWideCharStr = (PWSTR)HeapAlloc(GetProcessHeap(), 0,
      nLenOfWideCharStr * sizeof(wchar_t));

   if (pWideCharStr == NULL)
      return(fOk);

   // Convert the multibyte string to a wide-character string.
   MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, cchLength,
      pWideCharStr, nLenOfWideCharStr);

   // Call the wide-character version of this
   // function to do the actual work.
   fOk = StringReverseW(pWideCharStr, cchLength);

   if (fOk) {
      // Convert the wide-character string back
      // to a multibyte string.
      WideCharToMultiByte(CP_ACP, 0, pWideCharStr, cchLength,
         pMultiByteStr, (int)strlen(pMultiByteStr), NULL, NULL);
   }

   // Free the memory containing the wide-character string.
   HeapFree(GetProcessHeap(), 0, pWideCharStr);

   return(fOk);
}
```

Finally, in the header file that you distribute with the dynamic-link library, you prototype the two functions as follows:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength);
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength);

#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

Determining If Text Is ANSI or Unicode

The Windows Notepad application allows you to open both Unicode and ANSI files as well as create them. In fact, Figure 2-5 shows Notepad's File Save As dialog box. Notice the different ways that you can save a text file.

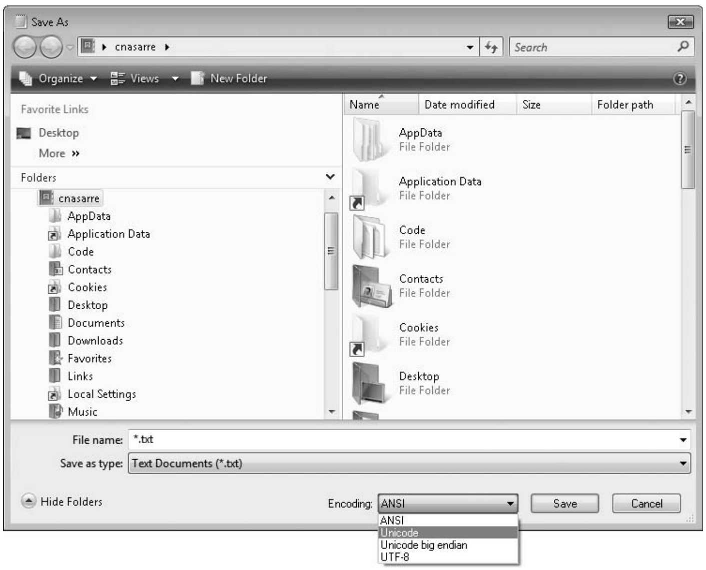


Figure 2-5 The Windows Vista Notepad File Save As dialog box

For many applications that open text files and process them, such as compilers, it would be convenient if, after opening a file, the application could determine whether the text file contained ANSI characters or Unicode characters. The **`IsTextUnicode`** function exported by AdvApi32.dll and declared in WinBase.h can help make this distinction:

```
BOOL IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

The problem with text files is that there are no hard and fast rules as to their content. This makes it extremely difficult to determine whether the file contains ANSI or Unicode characters. **`IsTextUnicode`** uses a series of statistical and deterministic methods to guess at the content of the buffer. Because this is not an exact science, it is possible that **`IsTextUnicode`** will return an incorrect result.

The first parameter, **pvBuffer**, identifies the address of a buffer that you want to test. The data is a void pointer because you don't know whether you have an array of ANSI characters or an array of Unicode characters.

The second parameter, **cb**, specifies the number of bytes that **pvBuffer** points to. Again, because you don't know what's in the buffer, **cb** is a count of bytes rather than a count of characters. Note that you do not have to specify the entire length of the buffer. Of course, the more bytes **IsTextUnicode** can test, the more accurate a response you're likely to get.

The third parameter, **pResult**, is the address of an integer that you must initialize before calling **IsTextUnicode**. You initialize this integer to indicate which tests you want **IsTextUnicode** to perform. You can also pass **NULL** for this parameter, in which case **IsTextUnicode** will perform every test it can. (See the Platform SDK documentation for more details.)

If **IsTextUnicode** thinks that the buffer contains Unicode text, **TRUE** is returned; otherwise, **FALSE** is returned. If specific tests were requested in the integer pointed to by the **pResult** parameter, the function sets the bits in the integer before returning to reflect the results of each test.

The FileRev sample application presented in Chapter 17, "Memory-Mapped Files," demonstrates the use of the **IsTextUnicode** function.

Chapter 10

Synchronous and Asynchronous Device I/O

In this chapter:

I can't stress enough the importance of this chapter, which covers the Microsoft Windows technologies that enable you to design high-performance, scalable, responsive, and robust applications. A scalable application handles a large number of concurrent operations as efficiently as it handles a small number of concurrent operations. For a service application, typically these operations are processing client requests that arrive at unpredictable times and require an unpredictable amount of processing power. These requests usually arrive from I/O devices such as network adapters; processing the requests frequently requires additional I/O devices such as disk files.

In Microsoft Windows applications, threads are the best facility available to help you partition work. Each thread is assigned to a processor, which allows a multiprocessor machine to execute multiple operations simultaneously, increasing throughput. When a thread issues a synchronous device I/O request, the thread is temporarily suspended until the device completes the I/O request. This suspension hurts performance because the thread is unable to do useful work, such as initiate another client's request for processing. So, in short, you want to keep your threads doing useful work all the time and avoid having them block.

To help keep threads busy, you need to make your threads communicate with one another about the operations they will perform. Microsoft has spent years researching and testing in this area and has developed a finely tuned mechanism to create this communication. This mechanism, called the *I/O completion port*, can help you create high-performance, scalable applications. By using the I/O completion port, you can make your application's threads achieve phenomenal throughput by reading and writing to devices without waiting for the devices to respond.

The I/O completion port was originally designed to handle device I/O, but over the years, Microsoft has architected more and more operating system facilities that fit seamlessly into the I/O completion port model. One example is the job kernel object, which monitors its processes and sends event notifications to an I/O completion port. The Job Lab sample application detailed in Chapter 5, "Jobs," demonstrates how I/O completion ports and job objects work together.

Throughout my many years as a Windows developer, I have found more and more uses for the I/O completion port, and I feel that every Windows developer must fully understand how the I/O completion port works. Even though I present the I/O completion port in this chapter about device I/O, be aware that the I/O completion port doesn't have to be used with device I/O at all—simply put, it is an awesome interthread communication mechanism with an infinite number of uses.

From this fanfare, you can probably tell that I'm a huge fan of the I/O completion port. My hope is that by the end of this chapter, you will be too. But instead of jumping right into the details of the I/O completion port, I'm going to explain what Windows originally offered developers for device I/O. This will give you a much greater appreciation for the I/O completion port. In "I/O Completion Ports" on page 320 I'll discuss the I/O completion port.

Opening and Closing Devices

One of the strengths of Windows is the sheer number of devices that it supports. In the context of this discussion, I define a device to be anything that allows communication. Table 10-1 lists some devices and their most common uses.

Table 10-1 Various Devices and Their Common Uses

Device	Most Common Use
File	Persistent storage of arbitrary data
Directory	Attribute and file compression settings
Logical disk drive	Drive formatting
Physical disk drive	Partition table access
Serial port	Data transmission over a phone line
Parallel port	Data transmission to a printer
Mailslot	One-to-many transmission of data, usually over a network to a machine running Windows
Named pipe	One-to-one transmission of data, usually over a network to a machine running Windows
Anonymous pipe	One-to-one transmission of data on a single machine (never over the network)
Socket	Datagram or stream transmission of data, usually over a network to any machine supporting sockets (The machine need not be running Windows.)
Console	A text window screen buffer

This chapter discusses how an application's threads communicate with these devices without waiting for the devices to respond. Windows tries to hide device differences from the software developer as much as possible. That is, once you open a device, the Windows functions that allow you to read and write data to the device are the same no matter what device you are communicating with. Although only a few functions are available for reading and writing data regardless of the device, devices are certainly different from one another. For example, it makes sense to set a baud rate for a serial port, but a baud rate has no meaning when using a named pipe to communicate over a network (or over the local machine). Devices are subtly different from one another, and I will not attempt to address all their nuances. However, I will spend some time addressing files because files are so common. To perform any type of I/O, you must first open the desired device and get a handle to it. The way you get the handle to a device depends on the particular device. Table 10-2 lists various devices and the functions you should call to open them.

Table 10-2 Functions for Opening Various Devices

Device	Function Used to Open the Device
File	**CreateFile** (**pszName** is pathname or UNC pathname).
Directory	**CreateFile** (**pszName** is directory name or UNC directory name). Windows allows you to open a directory if you specify the **FILE_FLAG_BACKUP_SEMANTICS** flag in the call to **CreateFile**. Opening the directory allows you to change the directory's attributes (to normal, hidden, and so on) and its time stamp.
Logical disk drive	**CreateFile** (**pszName** is "\\.*x*:"). Windows allows you to open a logical drive if you specify a string in the form of "\\.*x*:" where *x* is a drive letter. For example, to open drive A, you specify "\\.\A:". Opening a drive allows you to format the drive or determine the media size of the drive.
Physical disk drive	**CreateFile** (**pszName** is "\\.\PHYSICALDRIVE*x*"). Windows allows you to open a physical drive if you specify a string in the form of "\\.\PHYSICALDRIVE*x*" where *x* is a physical drive number. For example, to read or write to physical sectors on the user's first physical hard disk, you specify "\\.\PHYSICALDRIVE0". Opening a physical drive allows you to access the hard drive's partition tables directly. Opening the physical drive is potentially dangerous; an incorrect write to the drive could make the disk's contents inaccessible by the operating system's file system.
Serial port	**CreateFile** (**pszName** is "COM*x*").
Parallel port	**CreateFile** (**pszName** is "LPT*x*").
Mailslot server	**CreateMailslot** (**pszName** is "\\.\mailslot*mailslotname*").
Mailslot client	**CreateFile** (**pszName** is "*servername*\mailslot*mailslotname*").
Named pipe server	**CreateNamedPipe** (**pszName** is "\\.\pipe*pipename*").
Named pipe client	**CreateFile** (**pszName** is "*servername*\pipe*pipename*").
Anonymous pipe	**CreatePipe** client and server.
Socket	**socket**, **accept**, or **AcceptEx**.
Console	**CreateConsoleScreenBuffer** or **GetStdHandle**.

Each function in Table 10-2 returns a handle that identifies the device. You can pass the handle to various functions to communicate with the device. For example, you call **SetCommConfig** to set the baud rate of a serial port:

```
BOOL SetCommConfig(
   HANDLE       hCommDev,
   LPCOMMCONFIG pCC,
   DWORD        dwSize);
```

And you use **SetMailslotInfo** to set the time-out value when waiting to read data:

```
BOOL SetMailslotInfo(
   HANDLE hMailslot,
   DWORD  dwReadTimeout);
```

As you can see, these functions require a handle to a device for their first argument.

When you are finished manipulating a device, you must close it. For most devices, you do this by calling the very popular **CloseHandle** function:

```
BOOL CloseHandle(HANDLE hObject);
```

However, if the device is a socket, you must call **closesocket** instead:

```
int closesocket(SOCKET s);
```

Also, if you have a handle to a device, you can find out what type of device it is by calling **GetFileType**:

```
DWORD GetFileType(HANDLE hDevice);
```

All you do is pass to the **GetFileType** function the handle to a device, and the function returns one of the values listed in Table 10-3.

Table 10-3 Values Returned by the GetFileType Function

Value	Description
FILE_TYPE_UNKNOWN	The type of the specified file is unknown.
FILE_TYPE_DISK	The specified file is a disk file.
FILE_TYPE_CHAR	The specified file is a character file, typically an LPT device or a console.
FILE_TYPE_PIPE	The specified file is either a named pipe or an anonymous pipe.

A Detailed Look at *CreateFile*

The **CreateFile** function, of course, creates and opens disk files, but don't let the name fool you—it opens lots of other devices as well:

```
HANDLE CreateFile(
   PCTSTR pszName,
   DWORD  dwDesiredAccess,
   DWORD  dwShareMode,
   PSECURITY_ATTRIBUTES psa,
   DWORD  dwCreationDisposition,
   DWORD  dwFlagsAndAttributes,
   HANDLE hFileTemplate);
```

As you can see, **CreateFile** requires quite a few parameters, allowing for a great deal of flexibility when opening a device. At this point, I'll discuss all these parameters in detail.

When you call **CreateFile**, the **pszName** parameter identifies the device type as well as a specific instance of the device.

The **dwDesiredAccess** parameter specifies how you want to transmit data to and from the device. You can pass these four generic values, which are described in Table 10-4. Certain devices allow for additional access control flags. For example, when opening a file, you can specify access flags such as **FILE_READ_ATTRIBUTES**. See the Platform SDK documentation for more information about these flags.

Table 10-4 Generic Values That Can Be Passed for CreateFile's dwDesiredAccess Parameter

Value	Meaning
0	You do not intend to read or write data to the device. Pass **0** when you just want to change the device's configuration settings—for example, if you want to change only a file's time stamp.
GENERIC_READ	Allows read-only access from the device.
GENERIC_WRITE	Allows write-only access to the device. For example, this value can be used to send data to a printer and by backup software. Note that **GENERIC_WRITE** does not imply **GENERIC_READ**.
GENERIC_READ | GENERIC_WRITE	Allows both read and write access to the device. This value is the most common because it allows the free exchange of data.

The **dwShareMode** parameter specifies device-sharing privileges. It controls how the device can be opened by additional calls to **CreateFile** while you still have the device opened yourself (that is, you haven't closed the device yet by calling **CloseHandle**). Table 10-5 describes the possible values that can be passed for the **dwShareMode** parameter.

Table 10-5 Values Related to I/O That Can Be Passed for CreateFile's dwShareMode Parameter

Value	Meaning
0	You require exclusive access to the device. If the device is already opened, your call to **CreateFile** fails. If you successfully open the device, future calls to **CreateFile** fail.
FILE_SHARE_READ	You require that the data maintained by the device can't be changed by any other kernel object referring to this device. If the device is already opened for write or exclusive access, your call to **CreateFile** fails. If you successfully open the device, future calls to **CreateFile** fail if **GENERIC_WRITE** access is requested.
FILE_SHARE_WRITE	You require that the data maintained by the device can't be read by any other kernel object referring to this device. If the device is already opened for read or exclusive access, your call to **CreateFile** fails. If you successfully open the device, future calls to **CreateFile** fail if **GENERIC_READ** access is requested.
FILE_SHARE_READ | FILE_SHARE_WRITE	You don't care if the data maintained by the device is read or written to by any other kernel object referring to this device. If the device is already opened for exclusive access, your call to **CreateFile** fails. If you successfully open the device, future calls to **CreateFile** fail when exclusive read, exclusive write, or exclusive read/write access is requested.
FILE_SHARE_DELETE	You don't care if the file is logically deleted or moved while you are working with the file. Internally, Windows marks a file for deletion and deletes it when all open handles to the file are closed.

> **Note** If you are opening a file, you can pass a pathname that is up to **MAX_PATH** (defined as 260 in WinDef.h) characters long. However, you can transcend this limit by calling **CreateFileW** (the Unicode version of **CreateFile**) and precede the pathname with "\\?\". Calling **CreateFileW** removes the prefix and allows you to pass a path that is almost 32,000 Unicode characters long. Remember, however, that you must use fully qualified paths when using this prefix; the system does not process relative directories such as "." and "..". Also, each individual component of the path is still limited to **MAX_PATH** characters. Don't be surprised to also see the **_MAX_PATH** constant in various source code because this is what C/C++ standard libraries define in stdlib.h as 260.

The **psa** parameter points to a **SECURITY_ATTRIBUTES** structure that allows you to specify security information and whether or not you'd like **CreateFile**'s returned handle to be inheritable. The security descriptor inside this structure is used only if you are creating a file on a secure file system such as NTFS; the security descriptor is ignored in all other cases. Usually, you just pass **NULL** for the **psa** parameter, indicating that the file is created with default security and that the returned handle is noninheritable.

The **dwCreationDisposition** parameter is most meaningful when **CreateFile** is being called to open a file as opposed to another type of device. Table 10-6 lists the possible values that you can pass for this parameter.

Table 10-6 Values That Can Be Passed for CreateFile's dwCreationDisposition Parameter

Value	Meaning
CREATE_NEW	Tells **CreateFile** to create a new file and to fail if a file with the same name already exists.
CREATE_ALWAYS	Tells **CreateFile** to create a new file regardless of whether a file with the same name already exists. If a file with the same name already exists, **CreateFile** overwrites the existing file.
OPEN_EXISTING	Tells **CreateFile** to open an existing file or device and to fail if the file or device doesn't exist.
OPEN_ALWAYS	Tells **CreateFile** to open the file if it exists and to create a new file if it doesn't exist.
TRUNCATE_EXISTING	Tells **CreateFile** to open an existing file, truncate its size to 0 bytes, and fail if the file doesn't already exist.

> **Note** When you are calling **CreateFile** to open a device other than a file, you must pass **OPEN_EXISTING** for the **dwCreationDisposition** parameter.

CreateFile's **dwFlagsAndAttributes** parameter has two purposes: it allows you to set flags that fine-tune the communication with the device, and if the device is a file, you also get to set the file's attributes. Most of these communication flags are signals that tell the system how you intend to access the device. The system can then optimize its caching algorithms to help your application work more efficiently. I'll describe the communication flags first and then discuss the file attributes.

CreateFile Cache Flags

This section describes the various **CreateFile** cache flags, focusing on file system objects. For other kernel objects such as mailslots, you should refer to the MSDN documentation to get more specific details.

FILE_FLAG_NO_BUFFERING This flag indicates not to use any data buffering when accessing a file. To improve performance, the system caches data to and from disk drives. Normally, you do not specify this flag, and the cache manager keeps recently accessed portions of the file system in memory. This way, if you read a couple of bytes from a file and then read a few more bytes, the file's data is most likely loaded in memory and the disk has to be accessed only once instead of twice, greatly improving performance. However, this process does mean that portions of the file's data are in memory twice: the cache manager has a buffer, and you called some function (such as **ReadFile**) that copied some of the data from the cache manager's buffer into your own buffer.

When the cache manager is buffering data, it might also read ahead so that the next bytes you're likely to read are already in memory. Again, speed is improved by reading more bytes than necessary from the file. Memory is potentially wasted if you never attempt to read further in the file. (See the **FILE_FLAG_SEQUENTIAL_SCAN** and **FILE_FLAG_RANDOM_ACCESS** flags, discussed next, for more about reading ahead.)

By specifying the **FILE_FLAG_NO_BUFFERING** flag, you tell the cache manager that you do not want it to buffer any data—you take on this responsibility yourself! Depending on what you're doing, this flag can improve your application's speed and memory usage. Because the file system's device driver is writing the file's data directly into the buffers that you supply, you must follow certain rules:

- You must always access the file by using offsets that are exact multiples of the disk volume's sector size. (Use the **GetDiskFreeSpace** function to determine the disk volume's sector size.)
- You must always read/write a number of bytes that is an exact multiple of the sector size.
- You must make sure that the buffer in your process' address space begins on an address that is integrally divisible by the sector size.

FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_RANDOM_ACCESS These flags are useful only if you allow the system to buffer the file data for you. If you specify the **FILE_FLAG_NO_BUFFERING** flag, both of these flags are ignored.

If you specify the **FILE_FLAG_SEQUENTIAL_SCAN** flag, the system thinks you are accessing the file sequentially. When you read some data from the file, the system actually reads more of the file's data than the amount you requested. This process reduces the number of hits to the hard disk and improves the speed of your application. If you perform any direct seeks on the file, the system has spent a little extra time and memory caching data that you are not accessing. This is perfectly OK, but if you do it often, you'd be better off specifying the **FILE_FLAG_RANDOM_ACCESS** flag. This flag tells the system not to pre-read file data.

To manage a file, the cache manager must maintain some internal data structures for the file—the larger the file, the more data structures required. When working with extremely large files, the cache manager might not be able to allocate the internal data structures it requires and will fail to open the file. To access extremely large files, you must open the file using the **FILE_FLAG_NO_BUFFERING** flag.

FILE_FLAG_WRITE_THROUGH This is the last cache-related flag. It disables intermediate caching of file-write operations to reduce the potential for data loss. When you specify this flag, the system writes all file modifications directly to the disk. However, the system still maintains an internal cache of the file's data, and file-read operations use the cached data (if available) instead of reading data directly from the disk. When this flag is used to open a file on a network server, the Windows file-write functions do not return to the calling thread until the data is written to the server's disk drive.

That's it for the buffer-related communication flags. Now let's discuss the remaining communication flags.

Miscellaneous *CreateFile* Flags

This section describes the other flags that exist to customize **CreateFile** behaviors outside of caching.

FILE_FLAG_DELETE_ON_CLOSE Use this flag to have the file system delete the file after all handles to it are closed. This flag is most frequently used with the **FILE_ATTRIBUTE_TEMPORARY** attribute. When these two flags are used together, your application can create a temporary file, write to it, read from it, and close it. When the file is closed, the system automatically deletes the file—what a convenience!

FILE_FLAG_BACKUP_SEMANTICS Use this flag in backup and restore software. Before opening or creating any files, the system normally performs security checks to be sure that the process trying to open or create a file has the requisite access privileges. However, backup and restore software is special in that it can override certain file security checks. When you specify the **FILE_FLAG_BACKUP_SEMANTICS** flag, the system checks the caller's access token to see whether the Backup/Restore File and Directories privileges are enabled. If the appropriate privileges are enabled, the system allows the file to be opened. You can also use the **FILE_FLAG_BACKUP_SEMANTICS** flag to open a handle to a directory.

FILE_FLAG_POSIX_SEMANTICS In Windows, filenames are case-preserved, whereas filename searches are case-insensitive. However, the POSIX subsystem requires that filename searches be case-sensitive. The **FILE_FLAG_POSIX_SEMANTICS** flag causes **CreateFile** to use a case-sensitive filename search when creating or opening a file. Use the **FILE_FLAG_POSIX_SEMANTICS** flag with extreme caution—if you use it when you create a file, that file might not be accessible to Windows applications.

FILE_FLAG_OPEN_REPARSE_POINT In my opinion, this flag should have been called **FILE_FLAG_IGNORE_REPARSE_POINT** because it tells the system to ignore the file's reparse attribute (if it exists). Reparse attributes allow a file system filter to modify the behavior of opening, reading, writing, and closing a file. Usually, the modified behavior is desired, so using the **FILE_FLAG_OPEN_REPARSE_POINT** flag is not recommended.

FILE_FLAG_OPEN_NO_RECALL This flag tells the system not to restore a file's contents from offline storage (such as tape) back to online storage (such as a hard disk). When files are not accessed for long periods of time, the system can transfer the file's contents to offline storage, freeing up hard disk space. When the system does this, the file on the hard disk is not destroyed; only the data in the file is destroyed. When the file is opened, the system automatically restores the data from offline storage. The **FILE_FLAG_OPEN_NO_RECALL** flag instructs the system not to restore the data and causes I/O operations to be performed against the offline storage medium.

FILE_FLAG_OVERLAPPED This flag tells the system that you want to access a device asynchronously. You'll notice that the default way of opening a device is synchronous I/O (not specifying **FILE_FLAG_OVERLAPPED**). Synchronous I/O is what most developers are used to. When you read data from a file, your thread is suspended, waiting for the information to be read. Once the information has been read, the thread regains control and continues executing.

Because device I/O is slow when compared with most other operations, you might want to consider communicating with some devices asynchronously. Here's how it works: Basically, you call a function to tell the operating system to read or write data, but instead of waiting for the I/O to complete, your call returns immediately, and the operating system completes the I/O on your behalf using its own threads. When the operating system has finished performing your requested I/O, you can be notified. Asynchronous I/O is the key to creating high-performance, scalable, responsive, and robust applications. Windows offers several methods of asynchronous I/O, all of which are discussed in this chapter.

File Attribute Flags

Now it's time to examine the attribute flags for **CreateFile**'s **dwFlagsAndAttributes** parameter, described in Table 10-7. These flags are completely ignored by the system unless you are creating a brand new file and you pass **NULL** for **CreateFile**'s **hFileTemplate** parameter. Most of the attributes should already be familiar to you.

Table 10-7 File Attribute Flags That Can Be Passed for CreateFile's dwFlagsAndAttributes Parameter

Flag	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this flag to mark files for backup or removal. When **CreateFile** creates a new file, this flag is automatically set.
FILE_ATTRIBUTE_ENCRYPTED	The file is encrypted.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It won't be included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only when it's used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The file exists, but its data has been moved to offline storage. This flag is useful for hierarchical storage systems.
FILE_ATTRIBUTE_READONLY	The file is read-only. Applications can read the file but can't write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of the operating system or is used exclusively by the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file's data will be used only for a short time. The file system tries to keep the file's data in RAM rather than on disk to keep the access time to a minimum.

Use **FILE_ATTRIBUTE_TEMPORARY** if you are creating a temporary file. When **CreateFile** creates a file with the temporary attribute, **CreateFile** tries to keep the file's data in memory instead of on the disk. This makes accessing the file's contents much faster. If you keep writing to the file and the system can no longer keep the data in RAM, the operating system will be forced to start writing the data to the hard disk. You can improve the system's performance by combining the **FILE_ATTRIBUTE_TEMPORARY** flag with the **FILE_FLAG_DELETE_ON_CLOSE** flag (discussed earlier). Normally, the system flushes a file's cached data when the file is closed. However, if the system sees that the file is to be deleted when it is closed, the system doesn't need to flush the file's cached data.

In addition to all these communication and attribute flags, a number of flags allow you to control the security quality of service when opening a named-pipe device. Because these flags are specific to named pipes only, I will not discuss them here. To learn about them, please read about the **CreateFile** function in the Platform SDK documentation.

CreateFile's last parameter, **hFileTemplate**, identifies the handle of an open file or is **NULL**. If **hFileTemplate** identifies a file handle, **CreateFile** ignores the attribute flags in the **dwFlagsAndAttributes** parameter completely and uses the attributes associated with the file identified by **hFileTemplate**. The file identified by **hFileTemplate** must have been opened with the **GENERIC_READ** flag for this to work. If **CreateFile** is opening an existing file (as opposed to creating a new file), the **hFileTemplate** parameter is ignored.

If **CreateFile** succeeds in creating or opening a file or device, the handle of the file or device is returned. If **CreateFile** fails, **INVALID_HANDLE_VALUE** is returned.

Note Most Windows functions that return a handle return **NULL** when the function fails. However, **CreateFile** returns **INVALID_HANDLE_VALUE** (defined as –1) instead. I have often seen code like this, which is incorrect:

```
HANDLE hFile = CreateFile(...);
if (hFile == NULL) {
   // We'll never get in here
} else {
   // File might or might not be created OK
}
```

Here's the correct way to check for an invalid file handle:

```
HANDLE hFile = CreateFile(...);
if (hFile == INVALID_HANDLE_VALUE) {
   // File not created
} else {
   // File created OK
}
```

Working with File Devices

Because working with files is so common, I want to spend some time addressing issues that apply specifically to file devices. This section shows how to position a file's pointer and change a file's size.

The first issue you must be aware of is that Windows was designed to work with extremely large files. Instead of representing a file's size using 32-bit values, the original Microsoft designers chose to use 64-bit values. This means that theoretically a file can reach a size of 16 EB (exabytes).

Dealing with 64-bit values in a 32-bit operating system makes working with files a little unpleasant because a lot of Windows functions require you to pass a 64-bit value as two separate 32-bit values. But as you'll see, working with the values is not too difficult and, in normal day-to-day operations, you probably won't need to work with a file greater than 4 GB. This means that the high 32 bits of the file's 64-bit size will frequently be 0 anyway.

Getting a File's Size

When working with files, quite often you will need to acquire the file's size. The easiest way to do this is by calling **GetFileSizeEx**:

```
BOOL GetFileSizeEx(
   HANDLE         hFile,
   PLARGE_INTEGER pliFileSize);
```

The first parameter, **hFile**, is the handle of an opened file, and the **pliFileSize** parameter is the address of a **LARGE_INTEGER** union. This union allows a 64-bit signed value to be referenced as two 32-bit values or as a single 64-bit value, and it can be quite convenient when working with file sizes and offsets. Here is (basically) what the union looks like:

```
typedef union _LARGE_INTEGER {
   struct {
      DWORD LowPart;    // Low  32-bit unsigned value
      LONG HighPart;    // High 32-bit signed value
   };
   LONGLONG QuadPart;   // Full 64-bit signed value
} LARGE_INTEGER, *PLARGE_INTEGER;
```

In addition to **LARGE_INTEGER**, there is a **ULARGE_INTEGER** structure representing an unsigned 64-bit value:

```
typedef union _ULARGE_INTEGER {
   struct {
      DWORD LowPart;     // Low  32-bit unsigned value
      DWORD HighPart;    // High 32-bit unsigned value
   };
    ULONGLONG QuadPart;   // Full 64-bit unsigned value
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Another very useful function for getting a file's size is **GetCompressedFileSize**:

```
DWORD GetCompressedFileSize(
   PCTSTR pszFileName,
   PDWORD pdwFileSizeHigh);
```

This function returns the file's physical size, whereas **GetFileSizeEx** returns the file's logical size. For example, consider a 100-KB file that has been compressed to occupy 85 KB. Calling **GetFileSizeEx** returns the logical size of the file—100 KB—whereas **GetCompressedFileSize** returns the actual number of bytes on disk occupied by the file—85 KB.

Unlike **GetFileSizeEx**, **GetCompressedFileSize** takes a filename passed as a string instead of taking a handle for the first parameter. The **GetCompressedFileSize** function returns the 64-bit size of the file in an unusual way: the low 32 bits of the file's size are the function's return value. The high 32 bits of the file's size are placed in the **DWORD** pointed to by the **pdwFileSizeHigh** parameter. Here the use of the **ULARGE_INTEGER** structure comes in handy:

```
ULARGE_INTEGER ulFileSize;
ulFileSize.LowPart = GetCompressedFileSize(TEXT("SomeFile.dat"),
   &ulFileSize.HighPart);

// 64-bit file size is now in ulFileSize.QuadPart
```

Positioning a File Pointer

Calling **CreateFile** causes the system to create a file kernel object that manages operations on the file. Inside this kernel object is a file pointer. This file pointer indicates the 64-bit offset within the file where the next synchronous read or write should be performed. Initially, this file pointer is set to 0, so if you call **ReadFile** immediately after a call to **CreateFile**, you will start reading the file from offset 0. If you read 10 bytes of the file into memory, the system updates the pointer associated with the file handle so that the next call to **ReadFile** starts reading at the eleventh byte in the file at offset 10. For example, look at this code, in which the first 10 bytes from the file are read into the buffer, and then the next 10 bytes are read into the buffer:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
ReadFile(hFile, pb, 10, &dwNumBytes, NULL);   // Reads bytes  0 - 9
ReadFile(hFile, pb, 10, &dwNumBytes, NULL);   // Reads bytes 10 - 19
```

Because each file kernel object has its own file pointer, opening the same file twice gives slightly different results:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
HANDLE hFile2 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL);   // Reads bytes 0 - 9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);   // Reads bytes 0 - 9
```

In this example, two different kernel objects manage the same file. Because each kernel object has its own file pointer, manipulating the file with one file object has no effect on the file pointer maintained by the other object, and the first 10 bytes of the file are read twice.

I think one more example will help make all this clear:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
HANDLE hFile2;
DuplicateHandle(
   GetCurrentProcess(), hFile1,
   GetCurrentProcess(), &hFile2,
   0, FALSE, DUPLICATE_SAME_ACCESS);
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL);   // Reads bytes  0 - 9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);   // Reads bytes 10 - 19
```

In this example, one file kernel object is referenced by two file handles. Regardless of which handle is used to manipulate the file, the one file pointer is updated. As in the first example, different bytes are read each time.

If you need to access a file randomly, you will need to alter the file pointer associated with the file's kernel object. You do this by calling **SetFilePointerEx**:

```
BOOL SetFilePointerEx(
   HANDLE         hFile,
   LARGE_INTEGER  liDistanceToMove,
   PLARGE_INTEGER pliNewFilePointer,
   DWORD          dwMoveMethod);
```

The **hFile** parameter identifies the file kernel object whose file pointer you want to change. The **liDistanceToMove** parameter tells the system by how many bytes you want to move the pointer. The number you specify is added to the current value of the file's pointer, so a negative number has the effect of stepping backward in the file. The last parameter of **SetFilePointerEx**, **dwMoveMethod**, tells **SetFilePointerEx** how to interpret the **liDistanceToMove** parameter. Table 10-8 describes the three possible values you can pass via **dwMoveMethod** to specify the starting point for the move.

Table 10-8 Values That Can Be Passed for SetFilePointerEx's dwMoveMethod Parameter

Value	Meaning
FILE_BEGIN	The file object's file pointer is set to the value specified by the **liDistanceToMove** parameter. Note that **liDistanceToMove** is interpreted as an unsigned 64-bit value.
FILE_CURRENT	The file object's file pointer has the value of **liDistanceToMove** added to it. Note that **liDistanceToMove** is interpreted as a signed 64-bit value, allowing you to seek backward in the file.
FILE_END	The file object's file pointer is set to the logical file size plus the **liDistanceToMove** parameter. Note that **liDistanceToMove** is interpreted as a signed 64-bit value, allowing you to seek backward in the file.

After **SetFilePointerEx** has updated the file object's file pointer, the new value of the file pointer is returned in the **LARGE_INTEGER** pointed to by the **pliNewFilePointer** parameter. You can pass **NULL** for **pliNewFilePointer** if you're not interested in the new pointer value.

Here are a few facts to note about **SetFilePointerEx**:

- Setting a file's pointer beyond the end of the file's current size is legal. Doing so does not actually increase the size of the file on disk unless you write to the file at this position or call **SetEndOfFile**.
- When using **SetFilePointerEx** with a file opened with **FILE_FLAG_NO_BUFFERING**, the file pointer can be positioned only on sector-aligned boundaries. The FileCopy sample application later in this chapter demonstrates how to do this properly.
- Windows does not offer a **GetFilePointerEx** function, but you can use **SetFilePointerEx** to move the pointer by 0 bytes to get the desired effect, as shown in the following code snippet:

```
LARGE_INTEGER liCurrentPosition = { 0 };
SetFilePointerEx(hFile, liCurrentPosition, &liCurrentPosition, FILE_CURRENT);
```

Setting the End of a File

Usually, the system takes care of setting the end of a file when the file is closed. However, you might sometimes want to force a file to be smaller or larger. On those occasions, call

```
BOOL SetEndOfFile(HANDLE hFile);
```

This **SetEndOfFile** function truncates or extends a file's size to the size indicated by the file object's file pointer. For example, if you wanted to force a file to be 1024 bytes long, you'd use **SetEndOfFile** this way:

```
HANDLE hFile = CreateFile(...);
LARGE_INTEGER liDistanceToMove;
liDistanceToMove.QuadPart = 1024;
SetFilePointerEx(hFile, liDistanceToMove, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);
```

Using Windows Explorer to examine the properties of this file reveals that the file is exactly 1024 bytes long.

Performing Synchronous Device I/O

This section discusses the Windows functions that allow you to perform synchronous device I/O. Keep in mind that a device can be a file, mailslot, pipe, socket, and so on. No matter which device is used, the I/O is performed using the same functions.

Without a doubt, the easiest and most commonly used functions for reading from and writing to devices are **ReadFile** and **WriteFile**:

```
BOOL ReadFile(
   HANDLE      hFile,
   PVOID       pvBuffer,
   DWORD       nNumBytesToRead,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);

BOOL WriteFile(
   HANDLE      hFile,
   CONST VOID  *pvBuffer,
   DWORD       nNumBytesToWrite,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);
```

The **hFile** parameter identifies the handle of the device you want to access. When the device is opened, you must not specify the **FILE_FLAG_OVERLAPPED** flag, or the system will think that you want to perform asynchronous I/O with the device. The **pvBuffer** parameter points to the buffer to which the device's data should be read or to the buffer containing the data that should be written to the device. The **nNumBytesToRead** and **nNumBytesToWrite** parameters tell **ReadFile** and **WriteFile** how many bytes to read from the device and how many bytes to write to the device, respectively.

The **pdwNumBytes** parameters indicate the address of a **DWORD** that the functions fill with the number of bytes successfully transmitted to and from the device. The last parameter, **pOverlapped**, should be **NULL** when performing synchronous I/O. You'll examine this parameter in more detail shortly when asynchronous I/O is discussed.

Both **ReadFile** and **WriteFile** return **TRUE** if successful. By the way, **ReadFile** can be called only for devices that were opened with the **GENERIC_READ** flag. Likewise, **WriteFile** can be called only when the device is opened with the **GENERIC_WRITE** flag.

Flushing Data to the Device

Remember from our look at the **CreateFile** function that you can pass quite a few flags to alter the way in which the system caches file data. Some other devices, such as serial ports, mailslots, and pipes, also cache data. If you want to force the system to write cached data to the device, you can call **FlushFileBuffers**:

```
BOOL FlushFileBuffers(HANDLE hFile);
```

The **FlushFileBuffers** function forces all the buffered data associated with a device that is identified by the **hFile** parameter to be written. For this to work, the device has to be opened with the **GENERIC_WRITE** flag. If the function is successful, **TRUE** is returned.

Synchronous I/O Cancellation

Functions that do synchronous I/O are easy to use, but they block any other operations from occurring on the thread that issued the I/O until the request is completed. A great example of this is a **CreateFile** operation. When a user performs mouse and keyboard input, window messages are inserted into a queue that is associated with the thread that created the window that the input is destined for. If that thread is stuck inside a call to **CreateFile**, waiting for **CreateFile** to return, the window messages are not getting processed and all the windows created by the thread are frozen. The most common reason why applications hang is because their threads are stuck waiting for synchronous I/O operations to complete!

With Windows Vista, Microsoft has added some big features in an effort to alleviate this problem. For example, if a console (CUI) application hangs because of synchronous I/O, the user is now able to hit Ctrl+C to gain control back and continue using the console; the user no longer has to kill the console process. Also, the new Vista file open/save dialog box allows the user to press the Cancel button when opening a file is taking an excessively long time (typically, as a result of attempting to access a file on a network server).

To build a responsive application, you should try to perform asynchronous I/O operations as much as possible. This typically also allows you to use very few threads in your application, thereby saving resources (such as thread kernel objects and stacks). Also, it is usually easy to offer your users the ability to cancel an operation when you initiate it asynchronously. For example, Internet Explorer allows the user to cancel (via a red X button or the Esc key) a Web request if it is taking too long and the user is impatient.

Unfortunately, certain Windows APIs, such as **CreateFile**, offer no way to call the methods asynchronously. Although some of these methods do ultimately time out if they wait too long (such as when attempting to access a network server), it would be best if there was an application programming interface (API) that you could call to force the thread to abort waiting and to just cancel the synchronous I/O operation. In Windows Vista, the following function allows you to cancel a pending synchronous I/O request for a given thread:

```
BOOL CancelSynchronousIo(HANDLE hThread);
```

The **hThread** parameter is a handle of the thread that is suspended waiting for the synchronous I/O request to complete. This handle must have been created with the **THREAD_TERMINATE** access. If this is not the case, **CancelSynchronousIo** fails and **GetLastError** returns **ERROR_ACCESS_DENIED**. When you create the thread yourself by using **CreateThread** or **_beginthreadex**, the returned handle has **THREAD_ALL_ACCESS**, which includes **THREAD_TERMINATE** access. However, if you are taking advantage of the thread pool or your cancellation code is called by a timer callback, you usually have to call **OpenThread** to get a thread handle corresponding to the current thread ID; don't forget to pass **THREAD_TERMINATE** as the first parameter.

If the specified thread was suspended waiting for a synchronous I/O operation to complete, **CancelSynchronousIo** wakes the suspended thread and the operation it was trying to perform returns failure; calling **GetLastError** returns **ERROR_OPERATION_ABORTED**. Also, **CancelSynchronousIo** returns **TRUE** to its caller.

Note that the thread calling **CancelSynchronousIo** doesn't really know where the thread that called the synchronous operation is. The thread could have been pre-empted and it has yet to actually communicate with the device; it could be suspended, waiting for the device to respond; or the device could have just responded, and the thread is in the process of returning from its call. If **CancelSynchronousIo** is called when the specified thread is not actually suspended waiting for the device to respond, **CancelSynchronousIo** returns **FALSE** and **GetLastError** returns **ERROR_NOT_FOUND**.

For this reason, you might want to use some additional thread synchronization (as discussed in Chapter 8, "Thread Synchronization in User Mode," and Chapter 9, "Thread Synchronization with

Kernel Objects") to know for sure whether you are cancelling a synchronous operation or not. However, in practice, this is usually not necessary, as it is typical for a user to initiate the cancellation and this usually happens because the user sees the application is suspended. Also, a user could try to initiate cancellation twice (or more) if the first attempt to cancel doesn't seem to work. By the way, Windows calls **CancelSynchronousIo** internally to allow the user to regain control of a command console and the file open/save dialog box.

> **Caution** Cancellation of I/O requests depends on the driver implementing the corresponding system layer. It might happen that such a driver does not support cancellation. In that case, **CancelSynchronousIo** would have returned **TRUE** anyway because this function has found a request to be marked as being cancelled. The real cancellation of the request is left as the responsibility of the driver. An example of a driver that was updated to support synchronous cancellation for Windows Vista is the network redirector.

Basics of Asynchronous Device I/O

Compared to most other operations carried out by a computer, device I/O is one of the slowest and most unpredictable. The CPU performs arithmetic operations and even paints the screen much faster than it reads data from or writes data to a file or across a network. However, using asynchronous device I/O enables you to better use resources and thus create more efficient applications.

Consider a thread that issues an asynchronous I/O request to a device. This I/O request is passed to a device driver, which assumes the responsibility of actually performing the I/O. While the device driver waits for the device to respond, the application's thread is not suspended as it waits for the I/O request to complete. Instead, this thread continues executing and performs other useful tasks.

At some point, the device driver finishes processing the queued I/O request and must notify the application that data has been sent, data has been received, or an error has occurred. You'll learn how the device driver notifies you of I/O completions in "Receiving Completed I/O Request Notifications" on page 310. For now, let's concentrate on how to queue asynchronous I/O requests. Queuing asynchronous I/O requests is the essence of designing a high-performance, scalable application, and it is what the remainder of this chapter is all about.

To access a device asynchronously, you must first open the device by calling **CreateFile**, specifying the **FILE_FLAG_OVERLAPPED** flag in the **dwFlagsAndAttributes** parameter. This flag notifies the system that you intend to access the device asynchronously.

To queue an I/O request for a device driver, you use the **ReadFile** and **WriteFile** functions that you already learned about in "Performing Synchronous Device I/O" on page 302. For convenience, I'll list the function prototypes again:

```
BOOL ReadFile(
   HANDLE      hFile,
   PVOID       pvBuffer,
   DWORD       nNumBytesToRead,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);
```

```
BOOL WriteFile(
   HANDLE      hFile,
   CONST VOID  *pvBuffer,
   DWORD       nNumBytesToWrite,
   PDWORD      pdwNumBytes,
   OVERLAPPED* pOverlapped);
```

When either of these functions is called, the function checks to see if the device, identified by the **hFile** parameter, was opened with the **FILE_FLAG_OVERLAPPED** flag. If this flag is specified, the function performs asynchronous device I/O. By the way, when calling either function for asynchronous I/O, you can (and usually do) pass **NULL** for the **pdwNumBytes** parameter. After all, you expect these functions to return before the I/O request has completed, so examining the number of bytes transferred is meaningless at this time.

The *OVERLAPPED* Structure

When performing asynchronous device I/O, you must pass the address to an initialized **OVERLAPPED** structure via the **pOverlapped** parameter. The word "overlapped" in this context means that the time spent performing the I/O request overlaps the time your thread spends performing other tasks. Here's what an **OVERLAPPED** structure looks like:

```
typedef struct _OVERLAPPED {
   DWORD  Internal;     // [out] Error code
   DWORD  InternalHigh; // [out] Number of bytes transferred
   DWORD  Offset;       // [in]  Low  32-bit file offset
   DWORD  OffsetHigh;   // [in]  High 32-bit file offset
   HANDLE hEvent;       // [in]  Event handle or data
} OVERLAPPED, *LPOVERLAPPED;
```

This structure contains five members. Three of these members—**Offset**, **OffsetHigh**, and **hEvent**—must be initialized prior to calling **ReadFile** or **WriteFile**. The other two members, **Internal** and **InternalHigh**, are set by the device driver and can be examined when the I/O operation completes. Here is a more detailed explanation of these member variables:

Offset and OffsetHigh When a file is being accessed, these members indicate the 64-bit offset in the file where you want the I/O operation to begin. Recall that each file kernel object has a file pointer associated with it. When issuing a synchronous I/O request, the system knows to start accessing the file at the location identified by the file pointer. After the operation is complete, the system updates the file pointer automatically so that the next operation can pick up where the last operation left off.

When performing asynchronous I/O, this file pointer is ignored by the system. Imagine what would happen if your code placed two asynchronous calls to **ReadFile** (for the same file kernel object). In this scenario, the system wouldn't know where to start reading for the second call to **ReadFile**. You probably wouldn't want to start reading the file at the same location used by the first call to **ReadFile**. You might want to start the second read at the byte in the file that followed the last byte that was read by the first call to **ReadFile**. To avoid the confusion of multiple asynchronous calls to the same object, all asynchronous I/O requests must specify the starting file offset in the **OVERLAPPED** structure.

Note that the **Offset** and **OffsetHigh** members are not ignored for nonfile devices—you must initialize both members to **0** or the I/O request will fail and **GetLastError** will return **ERROR_INVALID_PARAMETER**.

hEvent This member is used by one of the four methods available for receiving I/O completion notifications. When using the alertable I/O notification method, this member can be used for your own purposes. I know many developers who store the address of a C++ object in **hEvent**. (This member will be discussed more in "Signaling an Event Kernel Object" on page 312.)

Internal This member holds the processed I/O's error code. As soon as you issue an asynchronous I/O request, the device driver sets **Internal** to **STATUS_PENDING**, indicating that no error has occurred because the operation has not started. In fact, the macro **HasOverlappedIoCompleted**, which is defined in WinBase.h, allows you to check whether an asynchronous I/O operation has completed. If the request is still pending, **FALSE** is returned; if the I/O request is completed, **TRUE** is returned. Here is the macro's definition:

```
#define HasOverlappedIoCompleted(pOverlapped) \
   ((pOverlapped)->Internal != STATUS_PENDING)
```

InternalHigh When an asynchronous I/O request completes, this member holds the number of bytes transferred.

When first designing the **OVERLAPPED** structure, Microsoft decided not to document the **Internal** and **InternalHigh** members (which explains their names). As time went on, Microsoft realized that the information contained in these members would be useful to developers, so it documented them. However, Microsoft didn't change the names of the members because the operating system source code referenced them frequently, and Microsoft didn't want to modify the code.

> **Note** When an asynchronous I/O request completes, you receive the address of the **OVERLAPPED** structure that was used when the request was initiated. Having more contextual information passed around with an **OVERLAPPED** structure is frequently useful—for example, if you wanted to store the handle of the device used to initiate the I/O request inside the **OVERLAPPED** structure. The **OVERLAPPED** structure doesn't offer a device handle member or other potentially useful members for storing context, but you can solve this problem quite easily.
>
> I frequently create a C++ class that is derived from an **OVERLAPPED** structure. This C++ class can have any additional information in it that I want. When my application receives the address of an **OVERLAPPED** structure, I simply cast the address to a pointer of my C++ class. Now I have access to the **OVERLAPPED** members and any additional context information my application needs. The FileCopy sample application at the end of this chapter demonstrates this technique. See my **CIOReq** C++ class in the FileCopy sample application for the details.

Asynchronous Device I/O Caveats

You should be aware of a couple of issues when performing asynchronous I/O. First, the device driver doesn't have to process queued I/O requests in a first-in first-out (FIFO) fashion. For example, if a thread executes the following code, the device driver will quite possibly write to the file and then read from the file:

```
OVERLAPPED o1 = { 0 };
OVERLAPPED o2 = { 0 };
BYTE bBuffer[100];
ReadFile (hFile, bBuffer, 100, NULL, &o1);
WriteFile(hFile, bBuffer, 100, NULL, &o2);
```

A device driver typically executes I/O requests out of order if doing so helps performance. For example, to reduce head movement and seek times, a file system driver might scan the queued I/O request list looking for requests that are near the same physical location on the hard drive.

The second issue you should be aware of is the proper way to perform error checking. Most Windows functions return **FALSE** to indicate failure or nonzero to indicate success. However, the **ReadFile** and **WriteFile** functions behave a little differently. An example might help to explain.

When attempting to queue an asynchronous I/O request, the device driver might choose to process the request synchronously. This can occur if you're reading from a file and the system checks whether the data you want is already in the system's cache. If the data is available, your I/O request is not queued to the device driver; instead, the system copies the data from the cache to your buffer, and the I/O operation is complete. The driver always performs certain operations synchronously, such as NTFS file compression, extending the length of a file or appending information to a file. For more information about operations that are always performed synchronously, please see *http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932*.

ReadFile and **WriteFile** return a nonzero value if the requested I/O was performed synchronously. If the requested I/O is executing asynchronously, or if an error occurred while calling **ReadFile** or **WriteFile**, **FALSE** is returned. When **FALSE** is returned, you must call **GetLastError** to determine specifically what happened. If **GetLastError** returns **ERROR_IO_PENDING**, the I/O request was successfully queued and will complete later.

If **GetLastError** returns a value other than **ERROR_IO_PENDING**, the I/O request could not be queued to the device driver. Here are the most common error codes returned from **GetLastError** when an I/O request can't be queued to the device driver:

ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY Each device driver maintains a fixed-size list (in a nonpaged pool) of outstanding I/O requests. If this list is full, the system can't queue your request, **ReadFile** and **WriteFile** return **FALSE**, and **GetLastError** reports one of these two error codes (depending on the driver).

ERROR_NOT_ENOUGH_QUOTA Certain devices require that your data buffer's storage be page locked so that the data cannot be swapped out of RAM while the I/O is pending. This page-locked storage requirement is certainly true of file I/O when using the **FILE_FLAG_NO_BUFFERING** flag. However, the system restricts the amount of storage that a single process can page lock. If **ReadFile** and **WriteFile** cannot page lock your buffer's storage, the functions return **FALSE** and **GetLastError** reports **ERROR_NOT_ENOUGH_QUOTA**. You can increase a process' quota by calling **SetProcessWorkingSetSize**.

How should you handle these errors? Basically, these errors occur because a number of outstanding I/O requests have not yet completed, so you need to allow some pending I/O requests to complete and then reissue the calls to **ReadFile** and **WriteFile**.

The third issue you should be aware of is that the data buffer and **OVERLAPPED** structure used to issue the asynchronous I/O request must not be moved or destroyed until the I/O request has completed. When queuing an I/O request to a device driver, the driver is passed the *address* of the data buffer and the *address* of the **OVERLAPPED** structure. Notice that just the address is passed, not the actual block. The reason for this should be quite obvious: memory copies are very expensive and waste a lot of CPU time.

When the device driver is ready to process your queued request, it transfers the data referenced by the **pvBuffer** address, and it accesses the file's offset member and other members contained within the **OVERLAPPED** structure pointed to by the **pOverlapped** parameter. Specifically, the device driver updates the **Internal** member with the I/O's error code and the **InternalHigh** member with the number of bytes transferred.

> **Note** It is absolutely essential that these buffers not be moved or destroyed until the I/O request has completed; otherwise, memory will be corrupted. Also, you must allocate and initialize a unique **OVERLAPPED** structure for each I/O request.

The preceding note is very important and is one of the most common bugs developers introduce when implementing an asynchronous device I/O architecture. Here's an example of what *not* to do:

```
VOID ReadData(HANDLE hFile) {
   OVERLAPPED o = { 0 };
   BYTE b[100];
   ReadFile(hFile, b, 100, NULL, &o);
}
```

This code looks fairly harmless, and the call to **ReadFile** is perfect. The only problem is that the function returns after queuing the asynchronous I/O request. Returning from the function essentially frees the buffer and the **OVERLAPPED** structure from the thread's stack, but the device driver is not aware that **ReadData** returned. The device driver still has two memory addresses that point to the thread's stack. When the I/O completes, the device driver is going to modify memory on the thread's stack, corrupting whatever happens to be occupying that spot in memory at the time. This bug is particularly difficult to find because the memory modification occurs asynchronously. Sometimes the device driver might perform I/O synchronously, in which case you won't see the bug. Sometimes the I/O might complete right after the function returns, or it might complete over an hour later, and who knows what the stack is being used for then.

Canceling Queued Device I/O Requests

Sometimes you might want to cancel a queued device I/O request before the device driver has processed it. Windows offers a few ways to do this:

- You can call **CancelIo** to cancel all I/O requests queued by the calling thread for the specified handle (unless the handle has been associated with an I/O completion port):

  ```
  BOOL CancelIo(HANDLE hFile);
  ```

- You can cancel all queued I/O requests, regardless of which thread queued the request, by closing the handle to a device itself.
- When a thread dies, the system automatically cancels all I/O requests issued by the thread, except for requests made to handles that have been associated with an I/O completion port.
- If you need to cancel a single, specific I/O request submitted on a given file handle, you can call **CancelIoEx**:

  ```
  BOOL CancelIoEx(HANDLE hFile, LPOVERLAPPED pOverlapped);.
  ```

  With **CancelIoEx**, you are able to cancel pending I/O requests emitted by a thread different from the calling thread. This function marks as canceled all I/O requests that are pending on **hFile** and associated with the given **pOverlapped** parameter. Because each outstanding I/O request should have its own **OVERLAPPED** structure, each call to **CancelIoEx** should cancel just one outstanding request. However, if the **pOverlapped** parameter is **NULL**, **CancelIoEx** cancels all outstanding I/O requests for the specified **hFile**.

> **Note** Canceled I/O requests complete with an error code of **ERROR_OPERATION_ABORTED**.

Receiving Completed I/O Request Notifications

At this point, you know how to queue an asynchronous device I/O request, but I haven't discussed how the device driver notifies you after the I/O request has completed.

Windows offers four different methods (briefly described in Table 10-9) for receiving I/O completion notifications, and this chapter covers all of them. The methods are shown in order of complexity, from the easiest to understand and implement (signaling a device kernel object) to the hardest to understand and implement (I/O completion ports).

Table 10-9 Methods for Receiving I/O Completion Notifications

Technique	Summary
Signaling a device kernel object	Not useful for performing multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Signaling an event kernel object	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Using alertable I/O	Allows multiple simultaneous I/O requests against a single device. The thread that issued an I/O request must also process it.
Using I/O completion ports	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it. This technique is highly scalable and has the most flexibility.

As stated at the beginning of this chapter, the I/O completion port is the hands-down best method of the four for receiving I/O completion notifications. By studying all four, you'll learn why Microsoft added the I/O completion port to Windows and how the I/O completion port solves all the problems that exist for the other methods.

Signaling a Device Kernel Object

Once a thread issues an asynchronous I/O request, the thread continues executing, doing useful work. Eventually, the thread needs to synchronize with the completion of the I/O operation. In other words, you'll hit a point in your thread's code at which the thread can't continue to execute unless the data from the device is fully loaded into the buffer.

In Windows, a device kernel object can be used for thread synchronization, so the object can either be in a signaled or nonsignaled state. The **ReadFile** and **WriteFile** functions set the device kernel object to the nonsignaled state just before queuing the I/O request. When the device driver completes the request, the driver sets the device kernel object to the signaled state.

A thread can determine whether an asynchronous I/O request has completed by calling either **WaitForSingleObject** or **WaitForMultipleObjects**. Here is a simple example:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
BYTE bBuffer[100];
OVERLAPPED o = { 0 };
o.Offset = 345;

BOOL bReadDone = ReadFile(hFile, bBuffer, 100, NULL, &o);
DWORD dwError = GetLastError();

if (!bReadDone && (dwError == ERROR_IO_PENDING)) {
   // The I/O is being performed asynchronously; wait for it to complete
   WaitForSingleObject(hFile, INFINITE);
   bReadDone = TRUE;
}

if (bReadDone) {
   // o.Internal contains the I/O error
   // o.InternalHigh contains the number of bytes transferred
   // bBuffer contains the read data
} else {
   // An error occurred; see dwError
}
```

This code issues an asynchronous I/O request and then immediately waits for the request to finish, defeating the purpose of asynchronous I/O! Obviously, you would never actually write code similar to this, but the code does demonstrate important concepts, which I'll summarize here:

- The device must be opened for asynchronous I/O by using the **FILE_FLAG_OVERLAPPED** flag.
- The **OVERLAPPED** structure must have its **Offset**, **OffsetHigh**, and **hEvent** members initialized. In the code example, I set them all to **0** except for **Offset**, which I set to **345** so that **ReadFile** reads data from the file starting at byte 346.
- **ReadFile**'s return value is saved in **bReadDone**, which indicates whether the I/O request was performed synchronously.

- If the I/O request was not performed synchronously, I check to see whether an error occurred or whether the I/O is being performed asynchronously. Comparing the result of **GetLastError** with **ERROR_IO_PENDING** gives me this information.
- To wait for the data, I call **WaitForSingleObject**, passing the handle of the device kernel object. As you saw in Chapter 9, calling this function suspends the thread until the kernel object becomes signaled. The device driver signals the object when it completes the I/O. After **WaitForSingleObject** returns, the I/O is complete and I set **bReadDone** to **TRUE**.
- After the read completes, you can examine the data in **bBuffer**, the error code in the **OVERLAPPED** structure's **Internal** member, and the number of bytes transferred in the **OVERLAPPED** structure's **InternalHigh** member.
- If a true error occurred, **dwError** contains the error code giving more information.

Signaling an Event Kernel Object

The method for receiving I/O completion notifications just described is very simple and straightforward, but it turns out not to be all that useful because it does not handle multiple I/O requests well. For example, suppose you were trying to carry out multiple asynchronous operations on a single file at the same time. Say that you wanted to read 10 bytes from the file and write 10 bytes to the file simultaneously. The code might look like this:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
...
WaitForSingleObject(hFile, INFINITE);

// We don't know what completed: Read? Write? Both?
```

You can't synchronize your thread by waiting for the device to become signaled because the object becomes signaled as soon as either of the operations completes. If you call **WaitForSingleObject**, passing it the device handle, you will be unsure whether the function returned because the read operation completed, the write operation completed, or both operations completed. Clearly, there needs to be a better way to perform multiple, simultaneous asynchronous I/O requests so that you don't run into this predicament—fortunately, there is.

The last member of the **OVERLAPPED** structure, **hEvent**, identifies an event kernel object. You must create this event object by calling **CreateEvent**. When an asynchronous I/O request completes, the device driver checks to see whether the **hEvent** member of the **OVERLAPPED** structure is **NULL**. If **hEvent** is not **NULL**, the driver signals the event by calling **SetEvent**. The driver also sets the device object to the signaled state just as it did before. However, if you are using events to determine when a device operation has completed, you shouldn't wait for the device object to become signaled—wait for the event instead.

> **Note** To improve performance slightly, you can tell Windows not to signal the file object when the operation completes. You do so by calling the **SetFileCompletionNotificationModes** function:
>
> `BOOL SetFileCompletionNotificationModes(HANDLE hFile, UCHAR uFlags);`
>
> The **hFile** parameter identifies a file handle, and the **uFlags** parameter indicates how Windows should modify its normal behavior with respect to completing an I/O operation. If you pass the **FILE_SKIP_SET_EVENT_ON_HANDLE** flag, Windows will not signal the file handle when operations on the file complete. Note that the **FILE_SKIP_SET_EVENT_ON_HANDLE** flag is very poorly named; a better name would have been something like **FILE_SKIP_SIGNAL**.

If you want to perform multiple asynchronous device I/O requests simultaneously, you must create a separate event object for each request, initialize the **hEvent** member in each request's **OVERLAPPED** structure, and then call **ReadFile** or **WriteFile**. When you reach the point in your code at which you need to synchronize with the completion of the I/O request, simply call **WaitForMultipleObjects**, passing in the event handles associated with each outstanding I/O request's **OVERLAPPED** structures. With this scheme, you can easily and reliably perform multiple asynchronous device I/O operations simultaneously and use the same device object. The following code demonstrates this approach:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
oRead.hEvent = CreateEvent(...);
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
oWrite.hEvent = CreateEvent(...);
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
...

HANDLE h[2];
h[0] = oRead.hEvent;
h[1] = oWrite.hEvent;
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw Ð WAIT_OBJECT_0) {
   case 0:   // Read completed
      break;

   case 1:   // Write completed
      break;
}
```

This code is somewhat contrived and is not *exactly* what you'd do in a real-life application, but it does illustrate my point. Typically, a real-life application has a loop that waits for I/O requests to complete. As each request completes, the thread performs the desired task, queues another asynchronous I/O request, and loops back around, waiting for more I/O requests to complete.

GetOverlappedResult

Recall that originally Microsoft was not going to document the **OVERLAPPED** structure's **Internal** and **InternalHigh** members, which meant it needed to provide another way for you to know how many bytes were transferred during the I/O processing and get the I/O's error code. To make this information available to you, Microsoft created the **GetOverlappedResult** function:

```
BOOL GetOverlappedResult(
   HANDLE      hFile,
   OVERLAPPED* pOverlapped,
   PDWORD      pdwNumBytes,
   BOOL        bWait);
```

Microsoft now documents the **Internal** and **InternalHigh** members, so the **GetOverlapped Result** function is not very useful. However, when I was first learning asynchronous I/O, I decided to reverse engineer the function to help solidify concepts in my head. The following code shows how **GetOverlappedResult** is implemented internally:

```
BOOL GetOverlappedResult(
   HANDLE hFile,
   OVERLAPPED* po,
   PDWORD pdwNumBytes,
   BOOL bWait) {


   if (po->Internal == STATUS_PENDING) {
      DWORD dwWaitRet = WAIT_TIMEOUT;
      if (bWait) {
         // Wait for the I/O to complete
         dwWaitRet = WaitForSingleObject(
            (po->hEvent != NULL) ? po->hEvent : hFile, INFINITE);
      }

      if (dwWaitRet == WAIT_TIMEOUT) {
         // I/O not complete and we're not supposed to wait
         SetLastError(ERROR_IO_INCOMPLETE);
         return(FALSE);
      }

      if (dwWaitRet != WAIT_OBJECT_0) {
         // Error calling WaitForSingleObject
         return(FALSE);
      }
   }

   // I/O is complete; return number of bytes transferred
   *pdwNumBytes = po->InternalHigh;

   if (SUCCEEDED(po->Internal)) {
      return(TRUE);   // No I/O error
   }

   // Set last error to I/O error
   SetLastError(po->Internal);
   return(FALSE);
}
```

Alertable I/O

The third method available to you for receiving I/O completion notifications is called *alertable I/O*. At first, Microsoft touted alertable I/O as the absolute best mechanism for developers who wanted to create high-performance, scalable applications. But as developers started using alertable I/O, they soon realized that it was not going to live up to the promise.

I have worked with alertable I/O quite a bit, and I'll be the first to tell you that alertable I/O is horrible and should be avoided. However, to make alertable I/O work, Microsoft added some infrastructure into the operating system that I have found to be extremely useful and valuable. As you read this section, concentrate on the infrastructure that is in place and don't get bogged down in the I/O aspects.

Whenever a thread is created, the system also creates a queue that is associated with the thread. This queue is called the asynchronous procedure call (APC) queue. When issuing an I/O request, you can tell the device driver to append an entry to the calling thread's APC queue. To have completed I/O notifications queued to your thread's APC queue, you call the **ReadFileEx** and **WriteFileEx** functions:

```
BOOL ReadFileEx(
   HANDLE      hFile,
   PVOID       pvBuffer,
   DWORD       nNumBytesToRead,
   OVERLAPPED* pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);

BOOL WriteFileEx(
   HANDLE      hFile,
   CONST VOID  *pvBuffer,
   DWORD       nNumBytesToWrite,
   OVERLAPPED* pOverlapped,
   LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
```

Like **ReadFile** and **WriteFile**, **ReadFileEx** and **WriteFileEx** issue I/O requests to a device driver, and the functions return immediately. The **ReadFileEx** and **WriteFileEx** functions have the same parameters as the **ReadFile** and **WriteFile** functions, with two exceptions. First, the ***Ex** functions are not passed a pointer to a **DWORD** that gets filled with the number of bytes transferred; this information can be retrieved only by the callback function. Second, the ***Ex** functions require that you pass the address of a callback function, called a *completion routine*. This routine must have the following prototype:

```
VOID WINAPI CompletionRoutine(
   DWORD       dwError,
   DWORD       dwNumBytes,
   OVERLAPPED* po);
```

When you issue an asynchronous I/O request with **ReadFileEx** and **WriteFileEx**, the functions pass the address of this function to the device driver. When the device driver has completed the I/O request, it appends an entry in the issuing thread's APC queue. This entry contains the address of the completion routine function and the address of the **OVERLAPPED** structure used to initiate the I/O request.

> **Note** By the way, when an alertable I/O completes, the device driver does not attempt to signal an event object. In fact, the device does not reference the **OVERLAPPED** structure's **hEvent** member at all. Therefore, you can use the **hEvent** member for your own purposes if you like.

When the thread is in an alertable state (discussed shortly), the system examines its APC queue and, for every entry in the queue, the system calls the completion function, passing it the I/O error code, the number of bytes transferred, and the address of the **OVERLAPPED** structure. Note that the error code and number of bytes transferred can also be found in the **OVERLAPPED** structure's **Internal** and **InternalHigh** members. (As I mentioned earlier, Microsoft originally didn't want to document these, so it passed them as parameters to the function.)

I'll get back to this completion routine function shortly. First let's look at how the system handles the asynchronous I/O requests. The following code queues three different asynchronous operations:

```
hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

ReadFileEx(hFile, ...);    // Perform first ReadFileEx
WriteFileEx(hFile, ...);   // Perform first WriteFileEx
ReadFileEx(hFile, ...);    // Perform second ReadFileEx

SomeFunc();
```

If the call to **SomeFunc** takes some time to execute, the system completes the three operations before **SomeFunc** returns. While the thread is executing the **SomeFunc** function, the device driver is appending completed I/O entries to the thread's APC queue. The APC queue might look something like this:

```
first WriteFileEx completed
second ReadFileEx completed
first ReadFileEx completed
```

The APC queue is maintained internally by the system. You'll also notice from the list that the system can execute your queued I/O requests in any order, and that the I/O requests that you issue last might be completed first and vice versa. Each entry in your thread's APC queue contains the address of a callback function and a value that is passed to the function.

As I/O requests complete, they are simply queued to your thread's APC queue—the callback routine is not immediately called because your thread might be busy doing something else and cannot be interrupted. To process entries in your thread's APC queue, the thread must put itself in an alertable state. This simply means that your thread has reached a position in its execution where it can handle being interrupted. Windows offers six functions that can place a thread in an alertable state:

```
DWORD SleepEx(
   DWORD dwMilliseconds,
   BOOL  bAlertable);

DWORD WaitForSingleObjectEx(
   HANDLE hObject,
   DWORD  dwMilliseconds,
   BOOL   bAlertable);

DWORD WaitForMultipleObjectsEx(
   DWORD   cObjects,
   CONST HANDLE* phObjects,
   BOOL    bWaitAll,
   DWORD   dwMilliseconds,
   BOOL    bAlertable);

BOOL SignalObjectAndWait(
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD  dwMilliseconds,
   BOOL   bAlertable);

BOOL GetQueuedCompletionStatusEx(
   HANDLE hCompPort,
   LPOVERLAPPED_ENTRY pCompPortEntries,
   ULONG ulCount,
   PULONG pulNumEntriesRemoved,
   DWORD dwMilliseconds,
   BOOL bAlertable);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD   nCount,
   CONST HANDLE* pHandles,
   DWORD   dwMilliseconds,
   DWORD   dwWakeMask,
   DWORD   dwFlags);
```

The last argument to the first five functions is a Boolean value indicating whether the calling thread should place itself in an alertable state. For **MsgWaitForMultipleObjectsEx**, you must use the **MWMO_ALERTABLE** flag to have the thread enter an alertable state. If you're familiar with the **Sleep**, **WaitForSingleObject**, and **WaitForMultipleObjects** functions, you shouldn't be surprised to learn that, internally, these non-**Ex** functions call their **Ex** counterparts, always passing **FALSE** for the **bAlertable** parameter.

When you call one of the six functions just mentioned and place your thread in an alertable state, the system first checks your thread's APC queue. If at least one entry is in the queue, the system does not put your thread to sleep. Instead, the system pulls the entry from the APC queue and your thread calls the callback routine, passing the routine the completed I/O request's error code, number of bytes transferred, and address of the **OVERLAPPED** structure. When the callback routine returns to the system, the system checks for more entries in the APC queue. If more entries exist, they are processed. However, if no more entries exist, your call to the alertable function returns. Something to keep in mind is that if any entries are in your thread's APC queue when you call any of these functions, your thread never sleeps!

The only time these functions suspend your thread is when no entries are in your thread's APC queue at the time you call the function. While your thread is suspended, the thread will wake up if the kernel object (or objects) that you're waiting on becomes signaled or if an APC entry appears in your thread's queue. Because your thread is in an alertable state, as soon as an APC entry appears, the system wakes your thread and empties the queue (by calling the callback routines). Then the functions immediately return to the caller—your thread does not go back to sleep waiting for kernel objects to become signaled.

The return value from these six functions indicates why they have returned. If they return **WAIT_IO_COMPLETION** (or if **GetLastError** returns **WAIT_IO_COMPLETION**), you know that the thread is continuing to execute because at least one entry was processed from the thread's APC queue. If the methods return for any other reason, the thread woke up because the sleep period expired, the specified kernel object or objects became signaled, or a mutex was abandoned.

The Bad and the Good of Alertable I/O

At this point, we've discussed the mechanics of performing alertable I/O. Now you need to know about the two issues that make alertable I/O a horrible method for doing device I/O:

Callback functions Alertable I/O requires that you create callback functions, which makes implementing your code much more difficult. These callback functions typically don't have enough contextual information about a particular problem to guide you, so you end up placing a lot of information in global variables. Fortunately, these global variables don't need to be synchronized because the thread calling one of the six alterable functions is the same thread executing the callback functions. A single thread can't be in two places at one time, so the variables are safe.

Threading issues The real big problem with alertable I/O is this: The thread issuing the I/O request must also handle the completion notification. If a thread issues several requests, that thread must respond to each request's completion notification, even if other threads are sitting completely idle. Because there is no load balancing, the application doesn't scale well.

Both of these problems are pretty severe, so I strongly discourage the use of alertable I/O for device I/O. I'm sure you guessed by now that the I/O completion port mechanism, discussed in the next section, solves both of the problems I just described. I promised to tell you some good stuff about the alertable I/O infrastructure, so before I move on to the I/O completion port, I'll do that.

Windows offers a function that allows you to manually queue an entry to a thread's APC queue:

```
DWORD QueueUserAPC(
   PAPCFUNC  pfnAPC,
   HANDLE    hThread,
   ULONG_PTR dwData);
```

The first parameter is a pointer to an APC function that must have the following prototype:

```
VOID WINAPI APCFunc(ULONG_PTR dwParam);
```

The second parameter is the handle of the thread for which you want to queue the entry. Note that this thread can be any thread in the system. If **hThread** identifies a thread in a different process' address space, **pfnAPC** must specify the memory address of a function that is in the address space of the target thread's process. The last parameter to **QueueUserAPC**, **dwData**, is a value that simply gets passed to the callback function.

Even though **QueueUserAPC** is prototyped as returning a **DWORD**, the function actually returns a **BOOL** indicating success or failure. You can use **QueueUserAPC** to perform extremely efficient interthread communication, even across process boundaries. Unfortunately, however, you can pass only a single value.

QueueUserAPC can also be used to force a thread out of a wait state. Suppose you have a thread calling **WaitForSingleObject**, waiting for a kernel object to become signaled. While the thread is waiting, the user wants to terminate the application. You know that threads should cleanly destroy themselves, but how do you force the thread waiting on the kernel object to wake up and kill itself? **QueueUserAPC** is the answer.

The following code demonstrates how to force a thread out of a wait state so that the thread can exit cleanly. The main function spawns a new thread, passing it the handle of some kernel object. While the secondary thread is running, the primary thread is also running. The secondary thread (executing the **ThreadFunc** function) calls **WaitForSingleObjectEx**, which suspends the thread, placing it in an alertable state. Now, say that the user tells the primary thread to terminate the application. Sure, the primary thread could just exit, and the system would kill the whole process. However, this approach is not very clean, and in many scenarios, you'll just want to kill an operation without terminating the whole process.

So the primary thread calls **QueueUserAPC**, which places an APC entry in the secondary thread's APC queue. Because the secondary thread is in an alertable state, it now wakes and empties its APC queue by calling the **APCFunc** function. This function does absolutely nothing and just returns. Because the APC queue is now empty, the thread returns from its call to **WaitForSingleObjectEx** with a return value of **WAIT_IO_COMPLETION**. The **ThreadFunc** function checks specifically for this return value, knowing that it received an APC entry indicating that the thread should exit.

```
// The APC callback function has nothing to do
VOID WINAPI APCFunc(ULONG_PTR dwParam) {
   // Nothing to do in here
}

UINT WINAPI ThreadFunc(PVOID pvParam) {
   HANDLE hEvent = (HANDLE) pvParam;   // Handle is passed to this thread

   // Wait in an alertable state so that we can be forced to exit cleanly
   DWORD dw = WaitForSingleObjectEx(hEvent, INFINITE, TRUE);
   if (dw == WAIT_OBJECT_0) {
      // Object became signaled
   }
   if (dw == WAIT_IO_COMPLETION) {
      // QueueUserAPC forced us out of a wait state
      return(0);   // Thread dies cleanly
   }
   ...
   return(0);
}
```

```
void main() {
   HANDLE hEvent = CreateEvent(...);
   HANDLE hThread = (HANDLE) _beginthreadex(NULL, 0,
      ThreadFunc, (PVOID) hEvent, 0, NULL);
   ...

   // Force the secondary thread to exit cleanly
   QueueUserAPC(APCFunc, hThread, NULL);
   WaitForSingleObject(hThread, INFINITE);
   CloseHandle(hThread);
   CloseHandle(hEvent);
}
```

I know that some of you are thinking that this problem could have been solved by replacing the call to **WaitForSingleObjectEx** with a call to **WaitForMultipleObjects** and by creating another event kernel object to signal the secondary thread to terminate. For my simple example, your solution would work. However, if my secondary thread called **WaitForMultipleObjects** to wait until all objects became signaled, **QueueUserAPC** would be the only way to force the thread out of a wait state.

I/O Completion Ports

Windows is designed to be a secure, robust operating system running applications that service literally thousands of users. Historically, you've been able to architect a service application by following one of two models:

Serial model A single thread waits for a client to make a request (usually over the network). When the request comes in, the thread wakes and handles the client's request.

Concurrent model A single thread waits for a client request and then creates a new thread to handle the request. While the new thread is handling the client's request, the original thread loops back around and waits for another client request. When the thread that is handling the client's request is completely processed, the thread dies.

The problem with the serial model is that it does not handle multiple, simultaneous requests well. If two clients make requests at the same time, only one can be processed at a time; the second request must wait for the first request to finish processing. A service that is designed using the serial approach cannot take advantage of multiprocessor machines. Obviously, the serial model is good only for the simplest of server applications, in which few client requests are made and requests can be handled very quickly. A Ping server is a good example of a serial server.

Because of the limitations in the serial model, the concurrent model is extremely popular. In the concurrent model, a thread is created to handle each client request. The advantage is that the thread waiting for incoming requests has very little work to do. Most of the time, this thread is sleeping. When a client request comes in, the thread wakes, creates a new thread to handle the request, and then waits for another client request. This means that incoming client requests are handled expediently. Also, because each client request gets its own thread, the server application scales well and can easily take advantage of multiprocessor machines. So if you are using the concurrent model and upgrade the hardware (add another CPU), the performance of the server application improves.

Service applications using the concurrent model were implemented using Windows. The Windows team noticed that application performance was not as high as desired. In particular, the team noticed that handling many simultaneous client requests meant that many threads were running in the system concurrently. Because all these threads were *runnable* (not suspended and waiting for something to happen), Microsoft realized that the Windows kernel spent too much time context switching between the running threads, and the threads were not getting as much CPU time to do their work. To make Windows an awesome server environment, Microsoft needed to address this problem. The result is the I/O completion port kernel object.

Creating an I/O Completion Port

The theory behind the I/O completion port is that the number of threads running concurrently must have an upper bound—that is, 500 simultaneous client requests cannot allow 500 runnable threads to exist. What, then, is the proper number of concurrent, runnable threads? Well, if you think about this question for a moment, you'll come to the realization that if a machine has two CPUs, having more than two runnable threads—one for each processor—really doesn't make sense. As soon as you have more runnable threads than CPUs available, the system has to spend time performing thread context switches, which wastes precious CPU cycles—a potential deficiency of the concurrent model.

Another deficiency of the concurrent model is that a new thread is created for each client request. Creating a thread is cheap when compared to creating a new process with its own virtual address space, but creating threads is far from free. The service application's performance can be improved if a pool of threads is created when the application initializes, and these threads hang around for the duration of the application. I/O completion ports were designed to work with a pool of threads.

An I/O completion port is probably the most complex kernel object. To create an I/O completion port, you call **CreateIoCompletionPort**:

```
HANDLE CreateIoCompletionPort(
   HANDLE    hFile,
   HANDLE    hExistingCompletionPort,
   ULONG_PTR CompletionKey,
   DWORD     dwNumberOfConcurrentThreads);
```

This function performs two different tasks: it creates an I/O completion port, and it associates a device with an I/O completion port. This function is overly complex, and in my opinion, Microsoft should have split it into two separate functions. When I work with I/O completion ports, I separate these two capabilities by creating two tiny functions that abstract the call to **CreateIoCompletionPort**. The first function I write is called **CreateNewCompletionPort**, and I implement it as follows:

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads) {

   return(CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0,
      dwNumberOfConcurrentThreads));
}
```

This function takes a single argument, **dwNumberOfConcurrentThreads**, and then calls the Windows **CreateIoCompletionPort** function, passing in hard-coded values for the first three parameters and **dwNumberOfConcurrentThreads** for the last parameter. You see, the first three parameters to **CreateIoCompletionPort** are used only when you are associating a device with a completion port. (I'll talk about this shortly.) To create just a completion port, I pass **INVALID_HANDLE_VALUE**, **NULL**, and **0**, respectively, to **CreateIoCompletionPort**'s first three parameters.

The **dwNumberOfConcurrentThreads** parameter tells the I/O completion port the maximum number of threads that should be runnable at the same time. If you pass **0** for the **dwNumberOfConcurrentThreads** parameter, the completion port defaults to allowing as many concurrent threads as there are CPUs on the host machine. This is usually exactly what you want so that extra context switching is avoided. You might want to increase this value if the processing of a client request requires a lengthy computation that rarely blocks, but increasing this value is strongly discouraged. You might experiment with the **dwNumberOfConcurrentThreads** parameter by trying different values and comparing your application's performance on your target hardware.

You'll notice that **CreateIoCompletionPort** is about the only Windows function that creates a kernel object but does not have a parameter that allows you to pass the address of a **SECURITY_ATTRIBUTES** structure. This is because completion ports are intended for use within a single process only. The reason will be clear to you when I explain how to use completion ports.

Associating a Device with an I/O Completion Port

When you create an I/O completion port, the kernel actually creates five different data structures, as shown in Figure 10-1. You should refer to this figure as you continue reading.

The first data structure is a device list indicating the device or devices associated with the port. You associate a device with the port by calling **CreateIoCompletionPort**. Again, I created my own function, **AssociateDeviceWithCompletionPort**, which abstracts the call to **CreateIoCompletionPort**:

```
BOOL AssociateDeviceWithCompletionPort(
   HANDLE hCompletionPort, HANDLE hDevice, DWORD dwCompletionKey) {

   HANDLE h = CreateIoCompletionPort(hDevice, hCompletionPort, dwCompletionKey, 0);
   return(h == hCompletionPort);
}
```

AssociateDeviceWithCompletionPort appends an entry to an existing completion port's device list. You pass to the function the handle of an existing completion port (returned by a previous call to **CreateNewCompletionPort**), the handle of the device (this can be a file, a socket, a mailslot, a pipe, and so on), and a completion key (a value that has meaning to you; the operating system doesn't care what you pass here). Each time you associate a device with the port, the system appends this information to the completion port's device list.

> **Note** The **CreateIoCompletionPort** function is complex, and I recommend that you mentally separate the two reasons for calling it. There is one advantage to having the function be so complex: you can create an I/O completion port and associate a device with it at the same time. For example, the following code opens a file and creates a new completion port, associating the file with it. All I/O requests to the file complete with a completion key of **CK_FILE**, and the port allows as many as two threads to execute concurrently.

```
#define CK_FILE   1
HANDLE hFile = CreateFile(...);
HANDLE hCompletionPort = CreateIoCompletionPort(hFile,NULL,CK_FILE,2);
```

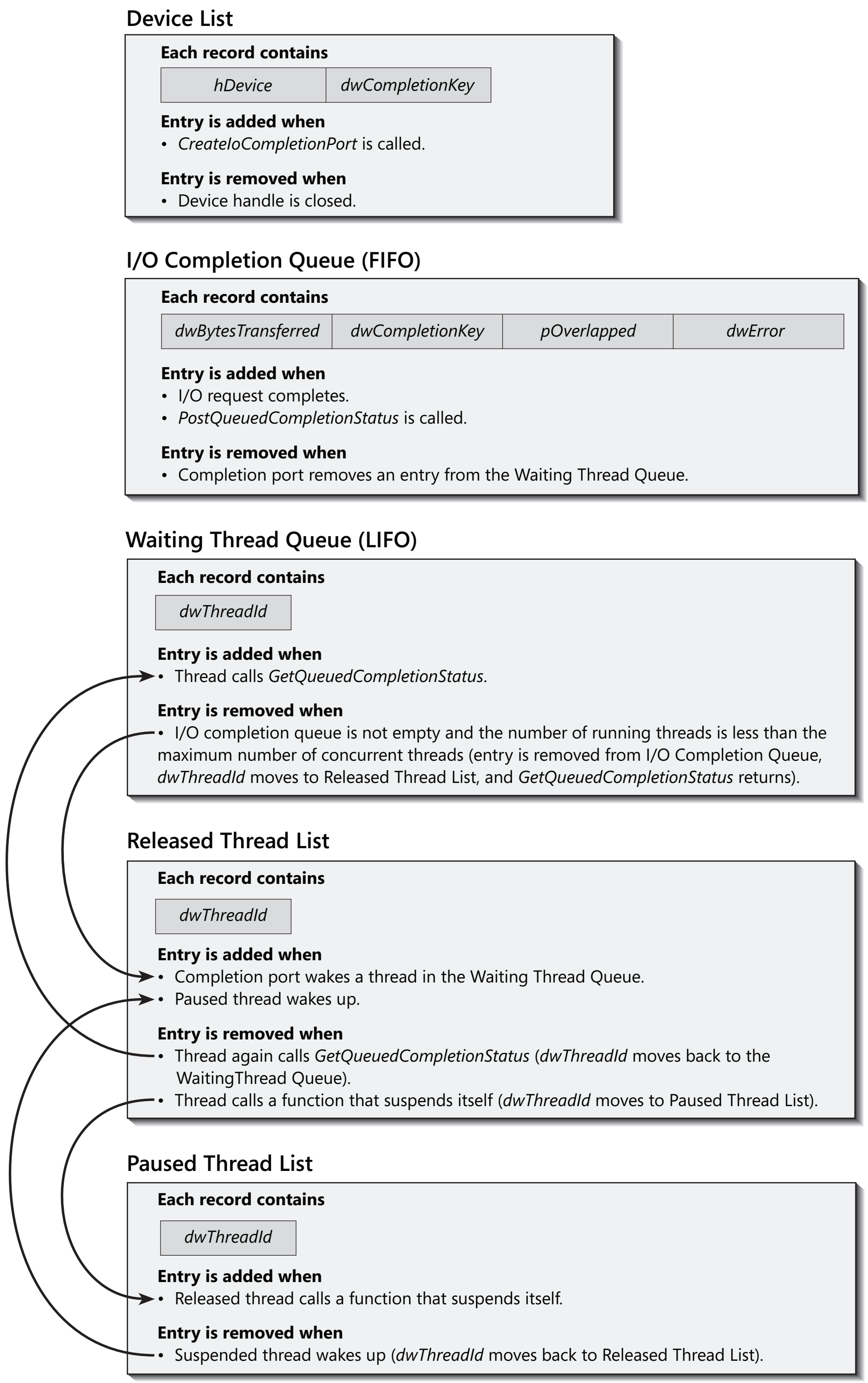


Figure 10-1 The internal workings of an I/O completion port

The second data structure is an I/O completion queue. When an asynchronous I/O request for a device completes, the system checks to see whether the device is associated with a completion port and, if it is, the system appends the completed I/O request entry to the end of the completion port's I/O completion queue. Each entry in this queue indicates the number of bytes transferred, the completion key value that was set when the device was associated with the port, the pointer to the I/O request's **OVERLAPPED** structure, and an error code. I'll discuss how entries are removed from this queue shortly.

Note Issuing an I/O request to a device and not having an I/O completion entry queued to the I/O completion port is possible. This is not usually necessary, but it can come in handy occasionally—for example, when you send data over a socket and you don't care whether the data actually makes it or not.

To issue an I/O request without having a completion entry queued, you must load the **OVERLAPPED** structure's **hEvent** member with a valid event handle and bitwise-**OR** this value with 1, like this:

```
Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWORD_PTR) Overlapped.hEvent | 1);
ReadFile(..., &Overlapped);
```

Now you can issue your I/O request, passing the address of this **OVERLAPPED** structure to the desired function (such as **ReadFile** above).

It would be nice if you didn't have to create an event just to stop the queuing of the I/O completion. I would like to be able to do the following, but it doesn't work:

```
Overlapped.hEvent = 1;
ReadFile(..., &Overlapped);
```

Also, don't forget to reset the low-order bit before closing this event handle:

```
CloseHandle((HANDLE) ((DWORD_PTR) Overlapped.hEvent & ~1));
```

Architecting Around an I/O Completion Port

When your service application initializes, it should create the I/O completion port by calling a function such as **CreateNewCompletionPort**. The application should then create a pool of threads to handle client requests. The question you ask now is, "How many threads should be in the pool?" This is a tough question to answer, and I will address it in more detail later in "How Many Threads in the Pool?" on page 328. For now, a standard rule of thumb is to take the number of CPUs on the host machine and multiply it by 2. So on a dual-processor machine, you should create a pool of four threads.

All the threads in the pool should execute the same function. Typically, this thread function performs some sort of initialization and then enters a loop that should terminate when the service process is instructed to stop. Inside the loop, the thread puts itself to sleep waiting for device I/O requests to complete to the completion port. Calling **GetQueuedCompletionStatus** does this:

```
BOOL GetQueuedCompletionStatus(
   HANDLE       hCompletionPort,
   PDWORD       pdwNumberOfBytesTransferred,
   PULONG_PTR   pCompletionKey,
   OVERLAPPED** ppOverlapped,
   DWORD        dwMilliseconds);
```

The first parameter, **hCompletionPort**, indicates which completion port the thread is interested in monitoring. Many service applications use a single I/O completion port and have all I/O request notifications complete to this one port. Basically, the job of **GetQueuedCompletionStatus** is to put the calling thread to sleep until an entry appears in the specified completion port's I/O completion queue or until the specified time-out occurs (as specified in the **dwMilliseconds** parameter).

The third data structure associated with an I/O completion port is the waiting thread queue. As each thread in the thread pool calls **GetQueuedCompletionStatus**, the ID of the calling thread is placed in this waiting thread queue, enabling the I/O completion port kernel object to always know which threads are currently waiting to handle completed I/O requests. When an entry appears in the port's I/O completion queue, the completion port wakes one of the threads in the waiting thread queue. This thread gets the pieces of information that make up a completed I/O entry: the number of bytes transferred, the completion key, and the address of the **OVERLAPPED** structure. This information is returned to the thread via the **pdwNumberOfBytesTransferred**, **pCompletionKey**, and **ppOverlapped** parameters passed to **GetQueuedCompletionStatus**.

Determining the reason that **GetQueuedCompletionStatus** returned is somewhat difficult. The following code demonstrates the proper way to do it:

```
DWORD dwNumBytes;
ULONG_PTR CompletionKey;
OVERLAPPED* pOverlapped;

// hIOCP is initialized somewhere else in the program
BOOL bOk = GetQueuedCompletionStatus(hIOCP,
   &dwNumBytes, &CompletionKey, &pOverlapped, 1000);
DWORD dwError = GetLastError();

if (bOk) {
   // Process a successfully completed I/O request
} else {
   if (pOverlapped != NULL) {
      // Process a failed completed I/O request
      // dwError contains the reason for failure
    } else {
      if (dwError == WAIT_TIMEOUT) {
         // Time-out while waiting for completed I/O entry
      } else {
         // Bad call to GetQueuedCompletionStatus
         // dwError contains the reason for the bad call
      }
   }
}
```

As you would expect, entries are removed from the I/O completion queue in a first-in first-out fashion. However, as you might not expect, threads that call **GetQueuedCompletionStatus** are awakened in a last-in first-out (LIFO) fashion. The reason for this is again to improve performance. For example, say that four threads are waiting in the waiting thread queue. If a single completed I/O entry appears, the last thread to call **GetQueuedCompletionStatus** wakes up to process the entry. When this last thread is finished processing the entry, the thread again calls **GetQueued CompletionStatus** to enter the waiting thread queue. Now if another I/O completion entry appears, the same thread that processed the first entry is awakened to process the new entry.

As long as I/O requests complete so slowly that a single thread can handle them, the system just keeps waking the one thread, and the other three threads continue to sleep. By using this LIFO algorithm, threads that don't get scheduled can have their memory resources (such as stack space) swapped out to the disk and flushed from a processor's cache. This means having many threads waiting on a completion port isn't bad. If you do have several threads waiting but few I/O requests completing, the extra threads have most of their resources swapped out of the system anyway.

In Windows Vista, if you expect a large number of I/O requests to be constantly submitted, instead of multiplying the number of threads to wait on the completion port and incurring the increasing cost of the corresponding context switches, you can retrieve the result of several I/O requests at the same time by calling the following function:

```
BOOL GetQueuedCompletionStatusEx(
  HANDLE hCompletionPort,
  LPOVERLAPPED_ENTRY pCompletionPortEntries,
  ULONG ulCount,
  PULONG pulNumEntriesRemoved,
  DWORD dwMilliseconds,
  BOOL bAlertable);
```

The first parameter, **hCompletionPort**, indicates which completion port the thread is interested in monitoring. The entries present in the specified completion port's I/O completion queue when this function is called are retrieved, and their description is copied into the **pCompletionPortEntries** array parameter. The **ulCount** parameter indicates how many entries can be copied in this array, and the long value pointed to by **pulNumEntriesRemoved** receives the exact number of I/O requests that were extracted from the completion queue.

Each element of the **pCompletionPortEntries** array is an **OVERLAPPED_ENTRY** that stores the pieces of information that make up a completed I/O entry: the completion key, the address of the **OVERLAPPED** structure, the result code (error) of the I/O request, and the number of bytes transferred.

```
typedef struct _OVERLAPPED_ENTRY {
   ULONG_PTR lpCompletionKey;
   LPOVERLAPPED lpOverlapped;
   ULONG_PTR Internal;
   DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

The **Internal** field is opaque and should not be used.

If the last **bAlertable** parameter is set to **FALSE**, the function waits for a completed I/O request to be queued on the completion port until the specified time-out occurs (as specified in the **dwMilliseconds** parameter). If the **bAlertable** parameter is set to **TRUE** and there is no completed I/O request in the queue, the thread enters an alertable state as explained earlier in this chapter.

Note When you issue an asynchronous I/O request to a device that is associated with a completion port, Windows queues the result to the completion port. Windows does this even if the asynchronous request is performed synchronously in order to give the programmer a consistent programming model. However, maintaining this consistent programming model hurts performance slightly because the completed request information must be placed in the port and a thread must extract it from the port.

To improve performance slightly, you can tell Windows not to queue a synchronously performed asynchronous request to the completion port associated with the device by calling the **SetFileCompletionNotificationModes** function (described in "Signaling an Event Kernel Object" on page 312) passing it the **FILE_SKIP_COMPLETION_PORT_ON_SUCCESS** flag.

The extremely performance-conscious programmer might also want to consider use of the **SetFileIoOverlappedRange** function. (See the Platform SDK documentation for more information.)

How the I/O Completion Port Manages the Thread Pool

Now it's time to discuss why I/O completion ports are so useful. First, when you create the I/O completion port, you specify the number of threads that can run concurrently. As I said, you usually set this value to the number of CPUs on the host machine. As completed I/O entries are queued, the I/O completion port wants to wake up waiting threads. However, the completion port wakes up only as many threads as you have specified. So if four I/O requests complete and four threads are waiting in a call to **GetQueuedCompletionStatus**, the I/O completion port will allow only two threads to wake up; the other two threads continue to sleep. As each thread processes a completed I/O entry, the thread again calls **GetQueuedCompletionStatus**. The system sees that more entries are queued and wakes the same threads to process the remaining entries.

If you're thinking about this carefully, you should notice that something just doesn't make a lot of sense: if the completion port only ever allows the specified number of threads to wake up concurrently, why have more threads waiting in the thread pool? For example, suppose I'm running on a machine with two CPUs and I create the I/O completion port, telling it to allow no more than two threads to process entries concurrently. But I create four threads (twice the number of CPUs) in the thread pool. It seems as though I am creating two additional threads that will never be awakened to process anything.

But I/O completion ports are very smart. When a completion port wakes a thread, the completion port places the thread's ID in the fourth data structure associated with the completion port, a released thread list. (See Figure 10-1.) This allows the completion port to remember which threads it awakened and to monitor the execution of these threads. If a released thread calls any function that places the thread in a wait state, the completion port detects this and updates its internal data structures by moving the thread's ID from the released thread list to the paused thread list (the fifth and final data structure that is part of an I/O completion port).

The goal of the completion port is to keep as many entries in the released thread list as are specified by the concurrent number of threads value used when creating the completion port. If a released thread enters a wait state for any reason, the released thread list shrinks and the completion port releases another waiting thread. If a paused thread wakes, it leaves the paused thread list and reenters the released thread list. This means that the released thread list can now have more entries in it than are allowed by the maximum concurrency value.

> **Note** Once a thread calls **GetQueuedCompletionStatus**, the thread is "assigned" to the specified completion port. The system assumes that all assigned threads are doing work on behalf of the completion port. The completion port wakes threads from the pool only if the number of running assigned threads is less than the completion port's maximum concurrency value.
>
> You can break the thread/completion port assignment in one of three ways:
>
> - Have the thread exit.
> - Have the thread call **GetQueuedCompletionStatus**, passing the handle of a different I/O completion port.
> - Destroy the I/O completion port that the thread is currently assigned to.

Let's tie all of this together now. Say that we are again running on a machine with two CPUs. We create a completion port that allows no more than two threads to wake concurrently, and we create four threads that are waiting for completed I/O requests. If three completed I/O requests get queued to the port, only two threads are awakened to process the requests, reducing the number of runnable threads and saving context-switching time. Now if one of the running threads calls **Sleep**, **WaitForSingleObject**, **WaitForMultipleObjects**, **SignalObjectAndWait**, a synchronous I/O call, or any function that would cause the thread not to be runnable, the I/O completion port would detect this and wake a third thread immediately. The goal of the completion port is to keep the CPUs saturated with work.

Eventually, the first thread will become runnable again. When this happens, the number of runnable threads will be higher than the number of CPUs in the system. However, the completion port again is aware of this and will not allow any additional threads to wake up until the number of threads drops below the number of CPUs. The I/O completion port architecture presumes that the number of runnable threads will stay above the maximum for only a short time and will die down quickly as the threads loop around and again call **GetQueuedCompletionStatus**. This explains why the thread pool should contain more threads than the concurrent thread count set in the completion port.

How Many Threads in the Pool?

Now is a good time to discuss how many threads should be in the thread pool. Consider two issues. First, when the service application initializes, you want to create a minimum set of threads so that you don't have to create and destroy threads on a regular basis. Remember that creating and destroying threads wastes CPU time, so you're better off minimizing this process as much as possible. Second, you want to set a maximum number of threads because creating too many threads wastes system resources. Even if most of these resources can be swapped out of RAM, minimizing the use of system resources and not wasting even paging file space is to your advantage, if you can manage it.

You will probably want to experiment with different numbers of threads. Most services (including Microsoft Internet Information Services) use heuristic algorithms to manage their thread pools. I recommend that you do the same. For example, you can create the following variables to manage the thread pool:

```
LONG g_nThreadsMin;    // Minimum number of threads in pool
LONG g_nThreadsMax;    // Maximum number of threads in pool
LONG g_nThreadsCrnt;   // Current number of threads in pool
LONG g_nThreadsBusy;   // Number of busy threads in pool
```

When your application initializes, you can create the **g_nThreadsMin** number of threads, all executing the same thread pool function. The following pseudocode shows how this thread function might look:

```
DWORD WINAPI ThreadPoolFunc(PVOID pv) {

   // Thread is entering pool
   InterlockedIncrement(&g_nThreadsCrnt);
   InterlockedIncrement(&g_nThreadsBusy);

   for (BOOL bStayInPool = TRUE; bStayInPool;) {

      // Thread stops executing and waits for something to do
      InterlockedDecrement(&m_nThreadsBusy);
      BOOL bOk = GetQueuedCompletionStatus(...);
      DWORD dwIOError = GetLastError();

      // Thread has something to do, so it's busy
      int nThreadsBusy = InterlockedIncrement(&m_nThreadsBusy);

      // Should we add another thread to the pool?
      if (nThreadsBusy == m_nThreadsCrnt) {    // All threads are busy
         if (nThreadsBusy < m_nThreadsMax) {   // The pool isn't full
            if (GetCPUUsage() < 75) {   // CPU usage is below 75%

               // Add thread to pool
               CloseHandle(chBEGINTHREADEX(...));
            }
         }
      }

      if (!bOk && (dwIOError == WAIT_TIMEOUT)) {   // Thread timed out
         // There isn't much for the server to do, and this thread
         // can die even if it still has outstanding I/O requests
         bStayInPool = FALSE;
      }

      if (bOk || (po != NULL)) {
         // Thread woke to process something; process it
         ...

         if (GetCPUUsage() > 90) {       // CPU usage is above 90%
             if (g_nThreadsCrnt > g_nThreadsMin)) { // Pool above min
                bStayInPool = FALSE;   // Remove thread from pool
            }
         }
      }
   }
```

```
   // Thread is leaving pool
   InterlockedDecrement(&g_nThreadsBusy);
   InterlockedDecrement(&g_nThreadsCurrent);
   return(0);
}
```

This pseudocode shows how creative you can get when using an I/O completion port. The **GetCPUUsage** function is not part of the Windows API. If you want its behavior, you'll have to implement the function yourself. In addition, you must make sure that the thread pool always contains at least one thread in it, or clients will never get tended to. Use my pseudocode as a guide, but your particular service might perform better if structured differently.

> **Note** Earlier in this chapter, in "Canceling Queued Device I/O Requests" on page 309, I said that the system automatically cancels all pending I/O requests issued by a thread when that thread terminates. Before Windows Vista, when a thread issued an I/O request against a device associated with a completion port, it was mandatory that the thread remain alive until the request completed; otherwise, Windows canceled any outstanding requests made by the thread. With Windows Vista, this is no longer necessary: threads can now issue requests and terminate; the request will still be processed and the result will be queued to the completion port.

Many services offer a management tool that allows an administrator to have some control over the thread pool's behavior—for example, to set the minimum and maximum number of threads, the CPU time usage thresholds, and also the maximum concurrency value used when creating the I/O completion port.

Simulating Completed I/O Requests

I/O completion ports do not have to be used with device I/O at all. This chapter is also about interthread communication techniques, and the I/O completion port kernel object is an awesome mechanism to use to help with this. In "Alertable I/O" on page 315, I presented the **QueueUserAPC** function, which allows a thread to post an APC entry to another thread. I/O completion ports have an analogous function, **PostQueuedCompletionStatus**:

```
BOOL PostQueuedCompletionStatus(
   HANDLE      hCompletionPort,
   DWORD       dwNumBytes,
   ULONG_PTR   CompletionKey,
   OVERLAPPED* pOverlapped);
```

This function appends a completed I/O notification to an I/O completion port's queue. The first parameter, **hCompletionPort**, identifies the completion port that you want to queue the entry for. The remaining three parameters—**dwNumBytes**, **CompletionKey**, and **pOverlapped**—indicate the values that should be returned by a thread's call to **GetQueuedCompletionStatus**. When a thread pulls a simulated entry from the I/O completion queue, **GetQueuedCompletionStatus** returns **TRUE**, indicating a successfully executed I/O request.

The **PostQueuedCompletionStatus** function is incredibly useful—it gives you a way to communicate with all the threads in your pool. For example, when the user terminates a service application, you want all the threads to exit cleanly. But if the threads are waiting on the completion port and no I/O requests are coming in, the threads can't wake up. By calling **PostQueuedCompletionStatus** once for each thread in the pool, each thread can wake up, examine the values returned from **GetQueuedCompletionStatus**, see that the application is terminating, and clean up and exit appropriately.

You must be careful when using a thread termination technique like the one I just described. My example works because the threads in the pool are dying and not calling **GetQueuedCompletionStatus** again. However, if you want to notify each of the pool's threads of something and have them loop back around to call **GetQueuedCompletionStatus** again, you will have a problem because the threads wake up in a LIFO order. So you will have to employ some additional thread synchronization in your application to ensure that each pool thread gets the opportunity to see its simulated I/O entry. Without this additional thread synchronization, one thread might see the same notification several times.

> **Note** In Windows Vista, when you call **CloseHandle** passing the handle of a completion port, all threads waiting in a call to **GetQueuedCompletionStatus** wake up and **FALSE** is returned to them. A call to **GetLastError** will return **ERROR_INVALID_HANDLE**; the threads can use this to know that it is time to die gracefully.

The FileCopy Sample Application

The FileCopy sample application (10-FileCopy.exe), shown at the end of this chapter, demonstrates the use of I/O completion ports. The source code and resource files for the application are in the 10-FileCopy directory on the companion content Web page. The program simply copies a file specified by the user to a new file called FileCopy.cpy. When the user executes FileCopy, the dialog box shown in Figure 10-2 appears.

Figure 10-2 The dialog box for the FileCopy sample application

The user clicks the Pathname button to select the file to be copied, and the Pathname and File Size fields are updated. When the user clicks the Copy button, the program calls the **FileCopy** function, which does all the hard work. Let's concentrate our discussion on the **FileCopy** function.

When preparing to copy, **FileCopy** opens the source file and retrieves its size, in bytes. I want the file copy to execute as blindingly fast as possible, so the file is opened using the **FILE_FLAG_NO_BUFFERING** flag. Opening the file with the **FILE_FLAG_NO_BUFFERING** flag allows me to access the file directly, bypassing the additional memory copy overhead incurred when allowing the system's cache to "help" access the file. Of course, accessing the file directly means slightly more work for me: I must always access the file using offsets that are multiples of the disk volume's sector size, and I must read and write data that is a multiple of the sector's size as well. I chose to transfer the file's data in **BUFFSIZE** (64 KB) chunks, which is guaranteed to be a multiple of the sector size. This is why I round up the source file's size to a multiple of **BUFFSIZE**. You'll also notice that the source file is opened with the **FILE_FLAG_OVERLAPPED** flag so that I/O requests against the file are performed asynchronously.

The destination file is opened similarly: both the **FILE_FLAG_NO_BUFFERING** and **FILE_FLAG_OVERLAPPED** flags are specified. I also pass the handle of the source file as **CreateFile**'s **hFileTemplate** parameter when creating the destination file, causing the destination file to have the same attributes as the source.

> **Note** Once both files are open, the destination file size is immediately set to its maximum size by calling **SetFilePointerEx** and **SetEndOfFile**. Adjusting the destination file's size now is extremely important because NTFS maintains a high-water marker that indicates the highest point at which the file was written. If you read past this marker, the system knows to return zeros. If you write past the marker, the file's data from the old high-water marker to the write offset is filled with zeros, your data is written to the file, and the file's high-water marker is updated. This behavior satisfies C2 security requirements pertaining to not presenting prior data. When you write to the end of a file on an NTFS partition, causing the high-water marker to move, NTFS must perform the I/O request synchronously even if asynchronous I/O was desired. If the **FileCopy** function didn't set the size of the destination file, none of the overlapped I/O requests would be performed asynchronously.

Now that the files are opened and ready to be processed, **FileCopy** creates an I/O completion port. To make working with I/O completion ports easier, I created a small C++ class, CIOCP, that is a very simple wrapper around the I/O completion port functions. This class can be found in the IOCP.h file discussed in Appendix A, "The Build Environment." **FileCopy** creates an I/O completion port by creating an instance (named **iocp**) of my CIOCP class.

The source file and destination file are associated with the completion port by calling the CIOCP's **AssociateDevice** member function. When associated with the completion port, each device is assigned a completion key. When an I/O request completes against the source file, the completion key is **CK_READ**, indicating that a read operation must have completed. Likewise, when an I/O request completes against the destination file, the completion key is **CK_WRITE**, indicating that a write operation must have completed.

Now we're ready to initialize a set of I/O requests (**OVERLAPPED** structures) and their memory buffers. The **FileCopy** function keeps four (**MAX_PENDING_IO_REQS**) I/O requests outstanding at any one time. For applications of your own, you might prefer to allow the number of I/O requests to dynamically grow or shrink as necessary. In the FileCopy program, the **CIOReq** class encapsulates a single I/O request. As you can see, this C++ class is derived from an **OVERLAPPED** structure but contains some additional context information. **FileCopy** allocates an array of **CIOReq** objects and calls the **AllocBuffer** method to associate a **BUFFSIZE**-sized data buffer with each I/O request object. The data buffer is allocated using the **VirtualAlloc** function. Using **VirtualAlloc** ensures that the block begins on an even allocation-granularity boundary, which satisfies the requirement of the **FILE_FLAG_NO_BUFFERING** flag: the buffer must begin on an address that is evenly divisible by the volume's sector size.

To issue the initial read requests against the source file, I perform a little trick: I post four **CK_WRITE** I/O completion notifications to the I/O completion port. When the main loop runs, the thread waits on the port and wakes immediately, thinking that a write operation has completed. This causes the thread to issue a read request against the source file, which really starts the file copy.

The main loop terminates when there are no outstanding I/O requests. As long as I/O requests are outstanding, the interior of the loop waits on the completion port by calling CIOCP's **GetStatus** method (which calls **GetQueuedCompletionStatus** internally). This call puts the thread to sleep until an I/O request completes to the completion port. When **GetQueuedCompletionStatus** returns, the returned completion key, **CompletionKey**, is checked. If **CompletionKey** is **CK_READ**, an I/O request against the source file is completed. I then call the **CIOReq**'s **Write** method to issue a write I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file is completed. If I haven't read beyond the end of the source file, I call **CIOReq**'s **Read** method to continue reading the source file.

When there are no more outstanding I/O requests, the loop terminates and cleans up by closing the source and destination file handles. Before **FileCopy** returns, it must do one more task: it must fix the size of the destination file so that it is the same size as the source file. To do this, I reopen the destination file without specifying the **FILE_FLAG_NO_BUFFERING** flag. Because I am not using this flag, file operations do not have to be performed on sector boundaries. This allows me to shrink the size of the destination file to the same size as the source file.

```
/******************************************************************************
Module:  FileCopy.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "stdafx.h"
#include "Resource.h"


///////////////////////////////////////////////////////////////////////////////


// Each I/O request needs an OVERLAPPED structure and a data buffer
class CIOReq : public OVERLAPPED {
public:
   CIOReq() {
      Internal = InternalHigh = 0;
      Offset = OffsetHigh = 0;
      hEvent = NULL;
      m_nBuffSize = 0;
      m_pvData = NULL;
   }

   ~CIOReq() {
      if (m_pvData != NULL)
         VirtualFree(m_pvData, 0, MEM_RELEASE);
   }

   BOOL AllocBuffer(SIZE_T nBuffSize) {
      m_nBuffSize = nBuffSize;
      m_pvData = VirtualAlloc(NULL, m_nBuffSize, MEM_COMMIT, PAGE_READWRITE);
      return(m_pvData != NULL);
   }
```

```
   BOOL Read(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
      return(::ReadFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

   BOOL Write(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
      return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

private:
   SIZE_T m_nBuffSize;
   PVOID  m_pvData;
};


///////////////////////////////////////////////////////////////////////////////


#define BUFFSIZE              (64 * 1024) // The size of an I/O buffer
#define MAX_PENDING_IO_REQS   4           // The maximum # of I/Os


// The completion key values indicate the type of completed I/O.
#define CK_READ  1
#define CK_WRITE 2


///////////////////////////////////////////////////////////////////////////////


BOOL FileCopy(PCTSTR pszFileSrc, PCTSTR pszFileDst) {

   BOOL fOk = FALSE;    // Assume file copy fails
   LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst;

   try {
      {
      // Open the source file without buffering & get its size
      CEnsureCloseFile hFileSrc = CreateFile(pszFileSrc, GENERIC_READ,
         FILE_SHARE_READ, NULL, OPEN_EXISTING,
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);
      if (hFileSrc.IsInvalid()) goto leave;

      // Get the file's size
      GetFileSizeEx(hFileSrc, &liFileSizeSrc);
```

```
      // Nonbuffered I/O requires sector-sized transfers.
      // I'll use buffer-size transfers since it's easier to calculate.
      liFileSizeDst.QuadPart = chROUNDUP(liFileSizeSrc.QuadPart, BUFFSIZE);

      // Open the destination file without buffering & set its size
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, CREATE_ALWAYS,
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hFileSrc);
      if (hFileDst.IsInvalid()) goto leave;

      // File systems extend files synchronously. Extend the destination file
      // now so that I/Os execute asynchronously improving performance.
      SetFilePointerEx(hFileDst, liFileSizeDst, NULL, FILE_BEGIN);
      SetEndOfFile(hFileDst);

      // Create an I/O completion port and associate the files with it.
      CIOCP iocp(0);
      iocp.AssociateDevice(hFileSrc, CK_READ);  // Read from source file
      iocp.AssociateDevice(hFileDst, CK_WRITE); // Write to destination file

      // Initialize record-keeping variables
      CIOReq ior[MAX_PENDING_IO_REQS];
      LARGE_INTEGER liNextReadOffset = { 0 };
      int nReadsInProgress  = 0;
      int nWritesInProgress = 0;

      // Prime the file copy engine by simulating that writes have completed.
      // This causes read operations to be issued.
      for (int nIOReq = 0; nIOReq < _countof(ior); nIOReq++) {

         // Each I/O request requires a data buffer for transfers
         chVERIFY(ior[nIOReq].AllocBuffer(BUFFSIZE));
         nWritesInProgress++;
         iocp.PostStatus(CK_WRITE, 0, &ior[nIOReq]);
      }

      // Loop while outstanding I/O requests still exist
      while ((nReadsInProgress > 0) || (nWritesInProgress > 0)) {

         // Suspend the thread until an I/O completes
         ULONG_PTR CompletionKey;
         DWORD dwNumBytes;
         CIOReq* pior;
         iocp.GetStatus(&CompletionKey, &dwNumBytes, (OVERLAPPED**) &pior, INFINITE);

         switch (CompletionKey) {
         case CK_READ:  // Read completed, write to destination
            nReadsInProgress--;
            pior->Write(hFileDst);  // Write to same offset read from source
            nWritesInProgress++;
            break;
```

```
         case CK_WRITE: // Write completed, read from source
            nWritesInProgress--;
            if (liNextReadOffset.QuadPart < liFileSizeDst.QuadPart) {
               // Not EOF, read the next block of data from the source file.
               pior->Read(hFileSrc, &liNextReadOffset);
               nReadsInProgress++;
               liNextReadOffset.QuadPart += BUFFSIZE; // Advance source offset
            }
            break;
         }
      }
      fOk = TRUE;
      }
   leave:;
   }
   catch (...) {
   }

   if (fOk) {
      // The destination file size is a multiple of the page size. Open the
      // file WITH buffering to shrink its size to the source file's size.
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, OPEN_EXISTING, 0, NULL);
      if (hFileDst.IsValid()) {

         SetFilePointerEx(hFileDst, liFileSizeSrc, NULL, FILE_BEGIN);
         SetEndOfFile(hFileDst);
      }
   }

   return(fOk);
}


///////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_FILECOPY);

   // Disable Copy button since no file is selected yet.
   EnableWindow(GetDlgItem(hWnd, IDOK), FALSE);
   return(TRUE);
}


///////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   TCHAR szPathname[_MAX_PATH];
```

```
   switch (id) {
   case IDCANCEL:
      EndDialog(hWnd, id);
      break;

   case IDOK:
      // Copy the source file to the destination file.
      Static_GetText(GetDlgItem(hWnd, IDC_SRCFILE),
         szPathname, sizeof(szPathname));
      SetCursor(LoadCursor(NULL, IDC_WAIT));
      chMB(FileCopy(szPathname, TEXT("FileCopy.cpy"))
         ? "File Copy Successful" : "File Copy Failed");
      break;

   case IDC_PATHNAME:
      OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
      ofn.hwndOwner = hWnd;
      ofn.lpstrFilter = TEXT("*.*\0");
      lstrcpy(szPathname, TEXT("*.*"));
      ofn.lpstrFile = szPathname;
      ofn.nMaxFile = _countof(szPathname);
      ofn.lpstrTitle = TEXT("Select file to copy");
      ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
      BOOL fOk = GetOpenFileName(&ofn);
      if (fOk) {
         // Show user the source file's size
         Static_SetText(GetDlgItem(hWnd, IDC_SRCFILE), szPathname);
         CEnsureCloseFile hFile = CreateFile(szPathname, 0, 0, NULL,
            OPEN_EXISTING, 0, NULL);
         if (hFile.IsValid()) {
            LARGE_INTEGER liFileSize;
            GetFileSizeEx(hFile, &liFileSize);
            // NOTE: Only shows bottom 32 bits of size
            SetDlgItemInt(hWnd, IDC_SRCFILESIZE, liFileSize.LowPart, FALSE);
         }
      }
      EnableWindow(GetDlgItem(hWnd, IDOK), fOk);
      break;
   }
}


///////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
   chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
   chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}
```

```
//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_FILECOPY), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

Index

B

C

J

K

L

N

O

S

U

V

W

Jeffrey Richter

Jeffrey Richter is a co-founder of Wintellect (*http://www.Wintellect.com/*), a training, debugging, and consulting company dedicated to helping companies produce better software faster. Jeff has written many books, including *CLR via C#* (Microsoft Press, 2005). Jeff is also a contributing editor for *MSDN Magazine*, where he has written several feature articles and is a columnist. Jeff also speaks at various trade conferences worldwide, including VSLive!, and Microsoft's TechEd and PDC. Jeff has consulted for many companies, including AT&T, DreamWorks, General Electric, Hewlett-Packard, IBM, and Intel. For Microsoft, Jeff's code has shipped in many products including TerraServer, Visual Studio, the .NET Framework, Office, and various versions of Windows. On the personal front, Jeff holds both airplane and helicopter pilot licenses, though he never gets to fly as often as he'd like. He is also a member of the International Brotherhood of Magicians and enjoys showing friends sleight-of-hand card tricks from time to time. Jeff's other hobbies include music, drumming, model railroading, boating, traveling, and the theater. He and his family live in Kirkland, Washington.

Christophe Nasarre

Christophe Nasarre works as a software architect and development lead for Business Objects, a multinational software company that is focused on helping organizations gain better insight into their business, improving decision-making and enterprise performance through business intelligence solutions. He has worked as a technical editor on numerous Addison-Wesley,