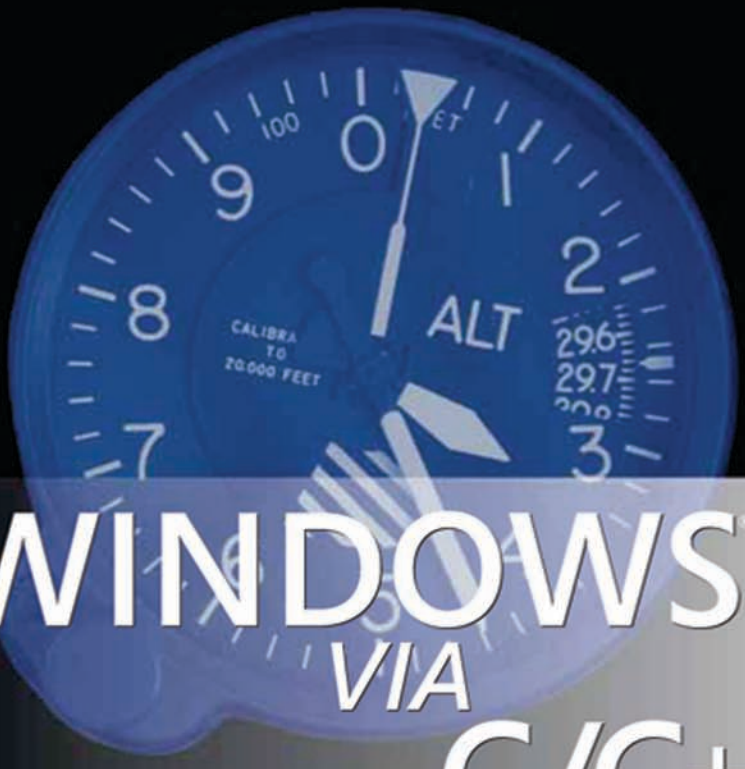
Microsoft

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

Table of Contents

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.

Part I
Required Reading

In this part:

Chapter 1

Error Handling

In this chapter:

Before we jump in and start examining the many features that Microsoft Windows has to offer, you should understand how the various Windows functions perform their error handling.

When you call a Windows function, it validates the parameters that you pass to it and then attempts to perform its duty. If you pass an invalid parameter or if for some other reason the action cannot be performed, the function's return value indicates that the function failed in some way. Table 1-1 shows the return value data types that most Windows functions use.

Table 1-1 Common Return Types for Windows Functions

Data Type	Value to Indicate Failure
VOID	This function cannot possibly fail. Very few Windows functions have a return type of **VOID**.
BOOL	If the function fails, the return value is 0; otherwise, the return value is non-zero. Avoid testing the return value to see if it is **TRUE**: it is always best to test this return value to see if it is different from **FALSE**.
HANDLE	If the function fails, the return value is usually **NULL**; otherwise, the **HANDLE** identifies an object that you can manipulate. Be careful with this one because some functions return a handle value of **INVALID_HANDLE_VALUE**, which is defined as –1. The Platform SDK documentation for the function will clearly state whether the function returns **NULL** or **INVALID_HANDLE_VALUE** to indicate failure.
PVOID	If the function fails, the return value is **NULL**; otherwise, **PVOID** identifies the memory address of a data block.
LONG/DWORD	This is a tough one. Functions that return counts usually return a **LONG** or **DWORD**. If for some reason the function can't count the thing you want counted, the function usually returns 0 or –1 (depending on the function). If you are calling a function that returns a **LONG/DWORD**, please read the Platform SDK documentation carefully to ensure that you are properly checking for potential errors.

When a Windows function returns with an error code, it's frequently useful to understand why the function failed. Microsoft has compiled a list of all possible error codes and has assigned each error code a 32-bit number.

Internally, when a Windows function detects an error, it uses a mechanism called thread-local storage to associate the appropriate error-code number with the calling thread. (Thread-local storage is

discussed in Chapter 21, "Thread-Local Storage.") This mechanism allows threads to run independently of each other without affecting each other's error codes. When the function returns to you, its return value indicates that an error has occurred. To see exactly which error this is, call the **GetLastError** function:

```
DWORD GetLastError();
```

This function simply returns the thread's 32-bit error code set by the last function call.

Now that you have the 32-bit error code number, you need to translate that number into something more useful. The WinError.h header file contains the list of Microsoft-defined error codes. I'll reproduce some of it here so that you can see what it looks like:

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
//  The operation completed successfully.
//
#define ERROR_SUCCESS                    0L

#define NO_ERROR 0L                             // dderror
#define SEC_E_OK                         ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
//  Incorrect function.
//
#define ERROR_INVALID_FUNCTION           1L    // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND             2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND             3L
```

```
//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
//  The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES        4L

//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
//  Access is denied.
//
#define ERROR_ACCESS_DENIED              5L
```

As you can see, each error has three representations: a message ID (a macro that you can use in your source code to compare against the return value of **GetLastError**), message text (an English text description of the error), and a number (which you should avoid using and instead use the message ID). Keep in mind that I selected only a very tiny portion of the WinError.h header file to show you; the complete file is more than 39,000 lines long!

When a Windows function fails, you should call **GetLastError** right away because the value is very likely to be overwritten if you call another Windows function. Notice that a Windows function that succeeds might overwrite this value with **ERROR_SUCCESS**.

Some Windows functions can succeed for several reasons. For example, attempting to create a named event kernel object can succeed either because you actually create the object or because an event kernel object with the same name already exists. Your application might need to know the reason for success. To return this information to you, Microsoft chose to use the last error-code mechanism. So when certain functions succeed, you can determine additional information by calling **GetLastError**. For functions with this behavior, the Platform SDK documentation clearly states that **GetLastError** can be used this way. See the documentation for the **CreateEvent** function for an example where **ERROR_ALREADY_EXISTS** is returned when a named event already exists.

While debugging, I find it extremely useful to monitor the thread's last error code. In Microsoft Visual Studio, Microsoft's debugger supports a useful feature—you can configure the Watch window to always show you the thread's last error code number and the text description of the error. This is done by selecting a row in the Watch window and typing **$err,hr**. Examine Figure 1-1. You'll see that I've called the **CreateFile** function. This function returned a **HANDLE** of **INVALID_HANDLE_VALUE** (–1), indicating that it failed to open the specified file. But the Watch window shows us that the last error code (the error code that would be returned by the **GetLastError** function if I called it) is 0x00000002. Thanks to the **,hr** qualifier, the Watch window further indicates that error code 2 is "The system cannot find the file specified." You'll notice that this is the same string mentioned in the WinError.h header file for error code number 2.

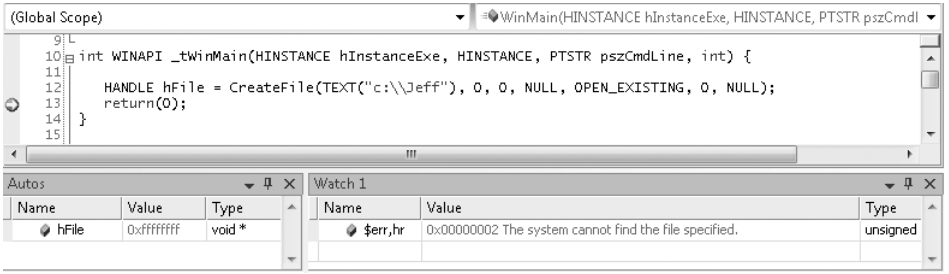
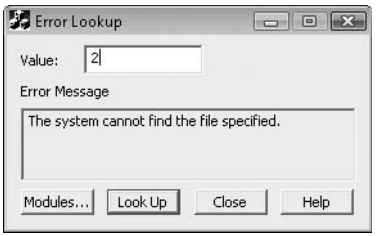


Figure 1-1 Using **`$err,hr`** in Visual Studio's Watch window to view the current thread's last error code

Visual Studio also ships with a small utility called Error Lookup. You can use Error Lookup to convert an error code number into its textual description.



If I detect an error in an application I've written, I might want to show the text description to the user. Windows offers a function that converts an error code into its text description. This function is called **`FormatMessage`**:

```
DWORD FormatMessage(
   DWORD dwFlags,
   LPCVOID pSource,
   DWORD dwMessageId,
   DWORD dwLanguageId,
   PTSTR pszBuffer,
   DWORD nSize,
   va_list *Arguments);
```

`FormatMessage` is actually quite rich in functionality and is the preferred way of constructing strings that are to be shown to the user. One reason for this function's usefulness is that it works easily with multiple languages. This function takes a language identifier as a parameter and returns the appropriate text. Of course, first you must translate the strings yourself and embed the translated message table resource inside your .exe or DLL module, but then the function will select the correct one. The ErrorShow sample application (shown later in this chapter) demonstrates how to call this function to convert a Microsoft-defined error code number into its text description.

Every now and then, someone asks me if Microsoft produces a master list indicating all the possible error codes that can be returned from every Windows function. The answer, unfortunately, is no. What's more, Microsoft will never produce this list—it's just too difficult to construct and maintain as new versions of the system are created.

The problem with assembling such a list is that you can call one Windows function, but internally that function might call another function, and so on. Any of these functions could fail, for lots of different reasons. Sometimes when a function fails, the higher-level function can recover and still perform what you want it to. To create this master list, Microsoft would have to trace the path of every function and build the list of all possible error codes. This is difficult. And as new versions of the system were created, these function-execution paths would change.

Defining Your Own Error Codes

OK, I've shown you how Windows functions indicate errors to their callers. Microsoft also makes this mechanism available to you for use in your own functions. Let's say you're writing a function that you expect others to call. Your function might fail for one reason or another and you need to indicate that failure back to your caller.

To indicate failure, simply set the thread's last error code and then have your function return **FALSE**, **INVALID_HANDLE_VALUE**, **NULL**, or whatever is appropriate. To set the thread's last error code, you simply call

```
VOID SetLastError(DWORD dwErrCode);
```

passing into the function whatever 32-bit number you think is appropriate. I try to use codes that already exist in WinError.h—as long as the code maps well to the error I'm trying to report. If you don't think that any of the codes in WinError.h accurately reflect the error, you can create your own code. The error code is a 32-bit number that is divided into the fields shown in Table 1-2.

Table 1-2 Error Code Fields

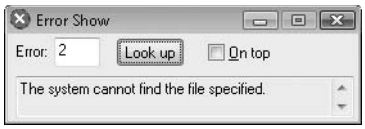
Bits:	31–30	29	28	27–16	15–0
Contents	Severity	Microsoft/ customer	Reserved	Facility code	Exception code
Meaning	0 = Success 1 = Informational 2 = Warning 3 = Error	0 = Microsoft-defined code 1 = customer-defined code	Must be 0	The first 256 values are reserved by Microsoft	Microsoft/ customer-defined code

These fields are discussed in detail in Chapter 24, "Exception Handlers and Software Exceptions." For now, the only important field you need to be aware of is in bit 29. Microsoft promises that all error codes it produces will have a 0 in this bit. If you create your own error codes, you must put a 1 in this bit. This way, you're guaranteed that your error code will never conflict with a Microsoft-defined error code that currently exists or is created in the future. Note that the Facility field is large enough to hold 4096 possible values. Of these, the first 256 values are reserved for Microsoft; the remaining values can be defined by your own application.

The ErrorShow Sample Application

The ErrorShow application, 01-ErrorShow.exe, demonstrates how to get the text description for an error code. The source code and resource files for the application are in the 01-ErrorShow directory on this book's companion content Web page, which is located at *http://wintellect.com/Books.aspx*.

Basically, this application shows how the debugger's Watch window and Error Lookup programs do their things. When you start the program, the following window appears.

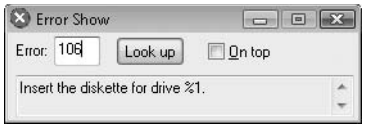


You can type any error number into the edit control. When you click the Look Up button, the error's text description is displayed in the scrollable window at the bottom. The only interesting feature of this application is how to call **FormatMessage**. Here's how I use this function:

```
// Get the error code
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL;   // Buffer that gets the error message string

// Use the default system locale since we look for Windows messages
// Note: this MAKELANGID combination has a value of 0
DWORD systemLocale = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);

// Get the error code's textual description
BOOL fOk = FormatMessage(
   FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
   FORMAT_MESSAGE_ALLOCATE_BUFFER,
   NULL, dwError, systemLocale,
   (PTSTR) &hlocal, 0, NULL);

if (!fOk) {
   // Is it a network-related error?
   HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
      DONT_RESOLVE_DLL_REFERENCES);

   if (hDll != NULL) {
      fOk = FormatMessage(
         FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_IGNORE_INSERTS |
         FORMAT_MESSAGE_ALLOCATE_BUFFER,
         hDll, dwError, systemLocale,
         (PTSTR) &hlocal, 0, NULL);
      FreeLibrary(hDll);
   }
}

if (fOk && (hlocal != NULL)) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT,
      TEXT("No text found for this error number."));
```

The first line retrieves the error code number out of the edit control. Then a handle to a memory block is instantiated and initialized to **NULL**. The **FormatMessage** function internally allocates the block of memory and returns its handle back to us.

When calling **FormatMessage**, I pass the **FORMAT_MESSAGE_FROM_SYSTEM** flag. This flag tells **FormatMessage** that we want the string for a system-defined error code. I also pass the **FORMAT_MESSAGE_ALLOCATE_BUFFER** flag to tell the function to allocate a block of memory large enough for the error's text description. The handle to this block will be returned in the **hlocal** variable. The **FORMAT_MESSAGE_IGNORE_INSERTS** flag lets you get messages with % placeholders for parameters that are used by Windows to provide more contextual information, as shown by the following screen shot:



If you don't pass this flag, you have to provide the values for these placeholders in the **Arguments** parameter; but this is not possible for Error Show because the content of the messages is not known in advance.

The third parameter indicates the error number we want looked up. The fourth parameter indicates what language we want the text description in. Because we are interested in messages provided by Windows itself, the language identifier is built based on the two specific constants whose association ends up being to the 0 value—meaning the default language of the operating system. This is a case where you can't hardcode a specific language because you don't know in advance what the operating system installation language will be.

If **FormatMessage** returns success, the text description is in the memory block and I copy it to the scrollable window at the bottom of the dialog box. If **FormatMessage** fails, I try to look up the message code in the NetMsg.dll module to see if the error is network related (look at Chapter 20, "DLL Advanced Techniques," for details about how DLLs are searched on the disk). Using the handle of the NetMsg.dll module, I again call **FormatMessage**. You see, each DLL (or .exe) can have its own set of error codes that you can add to the module by using the Message Compiler (MC.exe) and adding a resource to the module. This is what Visual Studio's Error Lookup tool allows you to do using the Modules dialog box.

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.

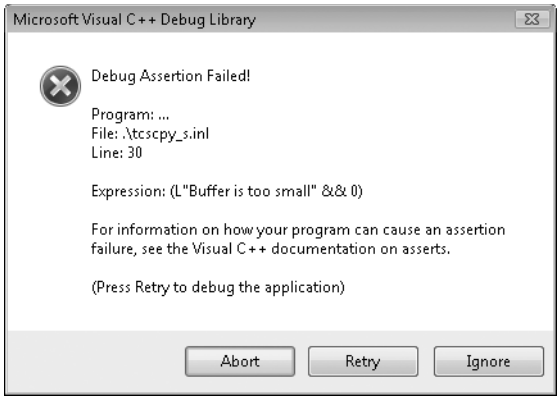


Figure 2-1 Assertion dialog box displayed when an error occurs

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.

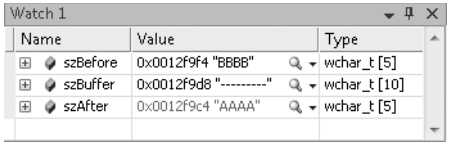


Figure 2-2 Variable state before the *_tcscpy_s* call

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.

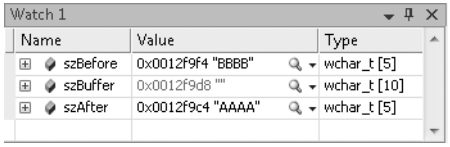


Figure 2-3 Variable state after the *_tcscpy_s* call

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.

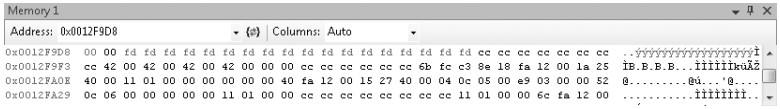


Figure 2-4 Content of *szBuffer* memory after a failed call

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 **StringCb Printf(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.

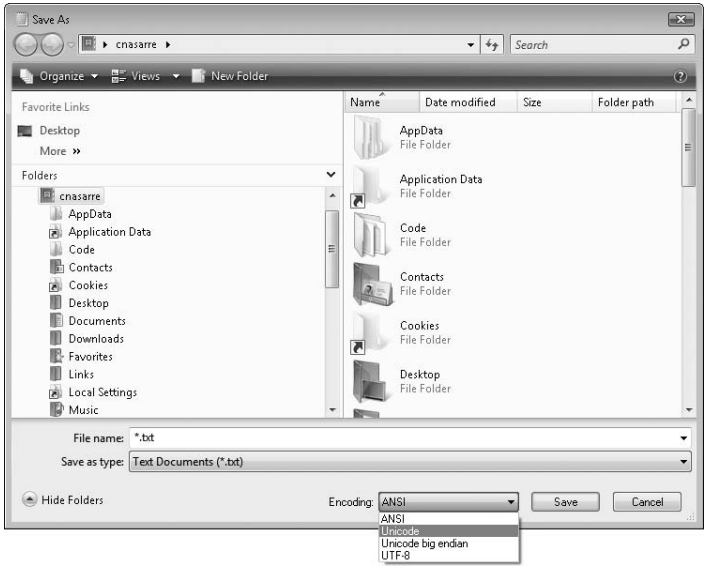


Figure 2-5 The Windows Vista Notepad File Save As dialog box

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 3

Kernel Objects

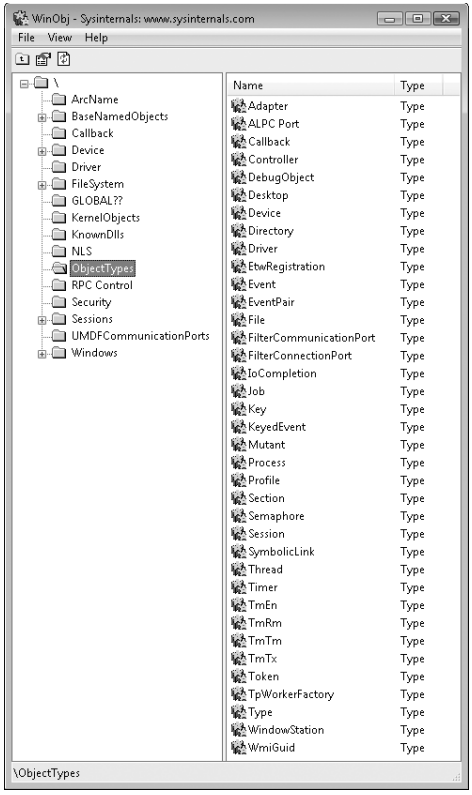
In this chapter:

We begin our understanding of the Microsoft Windows application programming interface (API) by examining kernel objects and their handles. This chapter covers relatively abstract concepts—I'm not going to discuss the particulars of any specific kernel object. Instead, I'm going to discuss features that apply to all kernel objects.

I would have preferred to start off with a more concrete topic, but a solid understanding of kernel objects is critical to becoming a proficient Windows software developer. Kernel objects are used by the system and by the applications we write to manage numerous resources such as processes, threads, and files (to name just a few). The concepts presented in this chapter will appear throughout most of the remaining chapters in this book. However, I do realize that some of the material covered in this chapter won't sink in until you start manipulating kernel objects using actual functions. So, as you read various other chapters in this book, you'll probably want to refer back to this chapter from time to time.

What Is a Kernel Object?

As a Windows software developer, you create, open, and manipulate kernel objects regularly. The system creates and manipulates several types of kernel objects, such as access token objects, event objects, file objects, file-mapping objects, I/O completion port objects, job objects, mailslot objects, mutex objects, pipe objects, process objects, semaphore objects, thread objects, waitable timer objects, and thread pool worker factory objects. The free WinObj tool from Sysinternals (located at *http://www.microsoft.com/technet/sysinternals/utilities/winobj.mspx*) allows you to see the list of all the kernel object types. Notice that you have to run it as Administrator through Windows Explorer to be able to see the list on the next page.



These objects are created by calling various functions with names that don't necessarily map to the type of the objects used at kernel level. For example, the **`CreateFileMapping`** function causes the system to create a file mapping that corresponds to a **`Section`** object, as you can see in WinObj. Each kernel object is simply a memory block allocated by the kernel and is accessible only by the kernel. This memory block is a data structure whose members maintain information about the object. Some members (security descriptor, usage count, and so on) are the same across all object types, but most are specific to a particular object type. For example, a process object has a process ID, a base priority, and an exit code, whereas a file object has a byte offset, a sharing mode, and an open mode.

Because the kernel object data structures are accessible only by the kernel, it is impossible for an application to locate these data structures in memory and directly alter their contents. Microsoft enforces this restriction deliberately to ensure that the kernel object structures maintain a consistent state. This restriction also allows Microsoft to add, remove, or change the members in these structures without breaking any applications.

If we cannot alter these structures directly, how do our applications manipulate these kernel objects? The answer is that Windows offers a set of functions that manipulate these structures in well-defined ways. These kernel objects are always accessible via these functions. When you call a function that creates a kernel object, the function returns a handle that identifies the object. Think of this handle as an opaque value that can be used by any thread in your process. A handle is a

32-bit value in a 32-bit Windows process and a 64-bit value in a 64-bit Windows process. You pass this handle to the various Windows functions so that the system knows which kernel object you want to manipulate. I'll talk a lot more about these handles later in this chapter.

To make the operating system robust, these handle values are process-relative. So if you were to pass this handle value to a thread in another process (using some form of interprocess communication), the calls that this other process would make using your process' handle value might fail or, even worse, they will create a reference to a totally different kernel object at the same index in your process handle table. In "Sharing Kernel Objects Across Process Boundaries" on page 43, we'll look at three mechanisms that allow multiple processes to successfully share a single kernel object.

Usage Counting

Kernel objects are owned by the kernel, not by a process. In other words, if your process calls a function that creates a kernel object and then your process terminates, the kernel object is not necessarily destroyed. Under most circumstances, the object will be destroyed; but if another process is using the kernel object your process created, the kernel knows not to destroy the object until the other process has stopped using it. The important thing to remember is that a kernel object can outlive the process that created it.

The kernel knows how many processes are using a particular kernel object because each object contains a usage count. The usage count is one of the data members common to all kernel object types. When an object is first created, its usage count is set to 1. When another process gains access to an existing kernel object, the usage count is incremented. When a process terminates, the kernel automatically decrements the usage count for all the kernel objects the process still has open. If the object's usage count goes to 0, the kernel destroys the object. This ensures that no kernel object will remain in the system if no processes are referencing the object.

Security

Kernel objects can be protected with a security descriptor. A security descriptor describes who owns the object (usually its creator), which group and users can gain access to or use the object, and which group and users are denied access to the object. Security descriptors are usually used when writing server applications. However, with Microsoft Windows Vista, this feature becomes more visible for client-side applications with private namespaces, as you will see later in this chapter and in "When Administrator Runs as a Standard User" on page 110.

Almost all functions that create kernel objects have a pointer to a **SECURITY_ATTRIBUTES** structure as an argument, as shown here with the **CreateFileMapping** function:

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);
```

Most applications simply pass **NULL** for this argument so that the object is created with a default security build based on the current process security token. However, you can allocate a

SECURITY_ATTRIBUTES structure, initialize it, and pass the address of the structure for this parameter. A **SECURITY_ATTRIBUTES** structure looks like this:

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Even though this structure is called **SECURITY_ATTRIBUTES**, it really includes only one member that has anything to do with security: **lpSecurityDescriptor**. If you want to restrict access to a kernel object you create, you must create a security descriptor and then initialize the **SECURITY_ATTRIBUTES** structure as follows:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);         // Used for versioning
sa.lpSecurityDescriptor = pSD;   // Address of an initialized SD
sa.bInheritHandle = FALSE;       // Discussed later
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
   PAGE_READWRITE, 0, 1024, TEXT("MyFileMapping"));
```

Because this member has nothing to do with security, I'll postpone discussing the **bInheritHandle** member until "Using Object Handle Inheritance" on page 43.

When you want to gain access to an existing kernel object (rather than create a new one), you must specify the operations you intend to perform on the object. For example, if I want to gain access to an existing file-mapping kernel object so that I could read data from it, I call **OpenFileMapping** as follows:

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
   TEXT("MyFileMapping"));
```

By passing **FILE_MAP_READ** as the first parameter to **OpenFileMapping**, I am indicating that I intend to read from this file mapping after I gain access to it. The **OpenFileMapping** function performs a security check first, before it returns a valid handle value. If I (the logged-on user) am allowed access to the existing file-mapping kernel object, **OpenFileMapping** returns a valid handle. However, if I am denied this access, **OpenFileMapping** returns **NULL** and a call to **GetLastError** will return a value of 5 (**ERROR_ACCESS_DENIED**). Don't forget that if the returned handle is used to call an API that requires a right different from **FILE_MAP_READ**, the same "access denied" error occurs. Again, most applications do not use security, so I won't go into this issue any further.

Although many applications do not need to be concerned about security, many Windows functions require that you pass desired security access information. Several applications designed for previous versions of Windows do not work properly on Windows Vista because security was not given enough consideration when the application was implemented.

For example, imagine an application that, when started, reads some data from a registry subkey. To do this properly, your code should call **RegOpenKeyEx**, passing **KEY_QUERY_VALUE** for the desired access.

However, many applications were originally developed for pre–Windows 2000 operating systems without any consideration for security. Some software developers could have called

RegOpenKeyEx, passing **KEY_ALL_ACCESS** as the desired access. Developers used this approach because it was a simpler solution and didn't require the developer to really think about what access was required. The problem is that the registry subkey, such as HKLM, might be readable, but not writable, to a user who is not an administrator. So, when such an application runs on Windows Vista, the call to **RegOpenKeyEx** with **KEY_ALL_ACCESS** fails, and without proper error checking, the application could run with totally unpredictable results.

If the developer had thought about security just a little and changed **KEY_ALL_ACCESS** to **KEY_QUERY_VALUE** (which is all that is necessary in this example), the product would work on all operating system platforms.

Neglecting proper security access flags is one of the biggest mistakes that developers make. Using the correct flags will certainly make it much easier to port an application between Windows versions. However, you also need to realize that each new version of Windows brings a new set of constraints that did not exist in the previous versions. For example, in Windows Vista, you need to take care of the User Account Control (UAC) feature. By default, UAC forces applications to run in a restricted context for security safety even though the current user is part of the Administrators group. We'll look at UAC more in Chapter 4, "Processes."

In addition to using kernel objects, your application might use other types of objects, such as menus, windows, mouse cursors, brushes, and fonts. These objects are User objects or Graphical Device Interface (GDI) objects, not kernel objects. When you first start programming for Windows, you might be confused when you try to differentiate a User object or a GDI object from a kernel object. For example, is an icon a User object or a kernel object? The easiest way to determine whether an object is a kernel object is to examine the function that creates the object. Almost all functions that create kernel objects have a parameter that allows you to specify security attribute information, as did the **CreateFileMapping** function shown earlier.

None of the functions that create User or GDI objects have a **PSECURITY_ATTRIBUTES** parameter. For example, take a look at the **CreateIcon** function:

```
HICON CreateIcon(
   HINSTANCE hinst,
   int nWidth,
   int nHeight,
   BYTE cPlanes,
   BYTE cBitsPixel,
   CONST BYTE *pbANDbits,
   CONST BYTE *pbXORbits);
```

The MSDN article *http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks* provides plenty of details about GDI and User objects and how to track them.

A Process' Kernel Object Handle Table

When a process is initialized, the system allocates a handle table for it. This handle table is used only for kernel objects, not for User objects or GDI objects. The details of how the handle table is structured and managed are undocumented. Normally, I would refrain from discussing undocumented parts of the operating system. In this case, however, I'm making an exception because I believe that a competent Windows programmer must understand how a process' handle table is

managed. Because this information is undocumented, I will not get all the details completely correct, and the internal implementation is certainly different among various versions of Windows. So read the following discussion to improve your understanding, not to learn how the system really does it.

Table 3-1 shows what a process' handle table looks like. As you can see, it is simply an array of data structures. Each structure contains a pointer to a kernel object, an access mask, and some flags.

Table 3-1 The Structure of a Process' Handle Table

Index	Pointer to Kernel Object Memory Block	Access Mask (DWORD of Flag Bits)	Flags
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...	...	...	...

Creating a Kernel Object

When a process first initializes, its handle table is empty. When a thread in the process calls a function that creates a kernel object, such as **`CreateFileMapping`**, the kernel allocates a block of memory for the object and initializes it. The kernel then scans the process' handle table for an empty entry. Because the handle table in Table 3-1 is empty, the kernel finds the structure at index 1 and initializes it. The pointer member will be set to the internal memory address of the kernel object's data structure, the access mask will be set to full access, and the flags will be set. (I'll discuss the flags in "Using Object Handle Inheritance" on page 43.

Here are some of the functions that create kernel objects (but this is in no way a complete list):

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   size_t dwStackSize,
   LPTHREAD_START_ROUTINE pfnStartAddress,
   PVOID pvParam,
   DWORD dwCreationFlags,
   PDWORD pdwThreadId);

HANDLE CreateFile(
   PCTSTR pszFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwCreationDisposition,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile);

HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);
```

```
HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);
```

All functions that create kernel objects return process-relative handles that can be used successfully by any and all threads that are running in the same process. This handle value should actually be divided by 4 (or shifted right two bits to ignore the last two bits that are used internally by Windows) to obtain the real index into the process' handle table that identifies where the kernel object's information is stored. So when you debug an application and examine the actual value of a kernel object handle, you'll see small values such as 4, 8, and so on. Remember that the meaning of the handle is undocumented and is subject to change.

Whenever you call a function that accepts a kernel object handle as an argument, you pass the value returned by one of the **`Create*`** functions. Internally, the function looks in your process' handle table to get the address of the kernel object you want to manipulate and then manipulates the object's data structure in a well-defined fashion.

If you pass an invalid handle, the function returns failure and **`GetLastError`** returns 6 (**`ERROR_INVALID_HANDLE`**). Because handle values are actually used as indexes into the process' handle table, these handles are process-relative and cannot be used successfully from other processes. And if you ever tried to do so, you would simply reference the kernel object stored at the same index into the other process' handle table, without any idea of what this object would be.

If you call a function to create a kernel object and the call fails, the handle value returned is usually 0 (**`NULL`**), and this is why the first valid handle value is 4. The system would have to be very low on memory or encountering a security problem for this to happen. Unfortunately, a few functions return a handle value of −1 (**`INVALID_HANDLE_VALUE`** defined in WinBase.h) when they fail. For example, if **`CreateFile`** fails to open the specified file, it returns **`INVALID_HANDLE_VALUE`** instead of **`NULL`**. You must be very careful when checking the return value of a function that creates a kernel object. Specifically, you can compare the value with **`INVALID_HANDLE_VALUE`** only when you call **`CreateFile`**. The following code is incorrect:

```
HANDLE hMutex = CreateMutex(É);
if (hMutex == INVALID_HANDLE_VALUE) {
   // We will never execute this code because
   // CreateMutex returns NULL if it fails.
}
```

Likewise, the following code is also incorrect:

```
HANDLE hFile = CreateFile(É);
if (hFile == NULL) {
   // We will never execute this code because CreateFile
   // returns INVALID_HANDLE_VALUE (-1) if it fails.
}
```

Closing a Kernel Object

Regardless of how you create a kernel object, you indicate to the system that you are done manipulating the object by calling **`CloseHandle`**:

```
BOOL CloseHandle(HANDLE hobject);
```

Internally, this function first checks the calling process' handle table to ensure that the handle value passed to it identifies an object that the process does in fact have access to. If the handle is valid, the system gets the address of the kernel object's data structure and decrements the usage count member in the structure. If the count is zero, the kernel object is destroyed and removed from memory.

If an invalid handle is passed to **CloseHandle**, one of two things might happen. If your process is running normally, **CloseHandle** returns **FALSE** and **GetLastError** returns **ERROR_INVALID_HANDLE**. Or, if your process is being debugged, the system throws the exception 0xC0000008 ("An invalid handle was specified") so that you can debug the error.

Right before **CloseHandle** returns, it clears out the entry in the process' handle table—this handle is now invalid for your process, and you should not attempt to use it. The clearing happens whether or not the kernel object has been destroyed! After you call **CloseHandle**, you no longer have access to the kernel object; however, if the object's count did not decrement to zero, the object has not been destroyed. This is OK; it just means that one or more other processes are still using the object. When the other processes stop using the object (by calling **CloseHandle**), the object will be destroyed.

Note Usually, when you create a kernel object, you store the corresponding handle in a variable. After you call **CloseHandle** with this variable as a parameter, you should also reset the variable to **NULL**. If, by mistake, you reuse this variable to call a Win32 function, two unexpected situations might occur. Because the handle table slot referenced by the variable has been cleared, Windows receives an invalid parameter and you get an error. But another situation that is harder to debug is also possible. When you create a new kernel object, Windows looks for a free slot in the handle table. So, if new kernel objects have been constructed in your application workflows, the handle table slot referenced by the variable will certainly contain one of these new kernel objects. Thus, the call might target a kernel object of the wrong type or, even worse, a kernel object of the same type as the closed one. Your application state then becomes corrupted without any chance to recover.

Let's say that you forget to call **CloseHandle**—will there be an object leak? Well, yes and no. It is possible for a process to leak resources (such as kernel objects) while the process runs. However, when the process terminates, the operating system ensures that all resources used by the process are freed—this is guaranteed. For kernel objects, the system performs the following actions: When your process terminates, the system automatically scans the process' handle table. If the table has any valid entries (objects that you didn't close before terminating), the system closes these object handles for you. If the usage count of any of these objects goes to zero, the kernel destroys the object.

So your application can leak kernel objects while it runs, but when your process terminates, the system guarantees that everything is cleaned up properly. By the way, this is true for *all* objects, resources such as GDI objects, and memory blocks—when a process terminates, the system ensures that your process leaves nothing behind. An easy way to detect whether kernel objects leak while your application is running is simply to use Windows Task Manager. First, as shown in Figure 3-1, you need to force the corresponding Handles column to appear on the Processes tab from the Select Process Page Columns dialog box that is accessible through the View/Select Columns menu item.

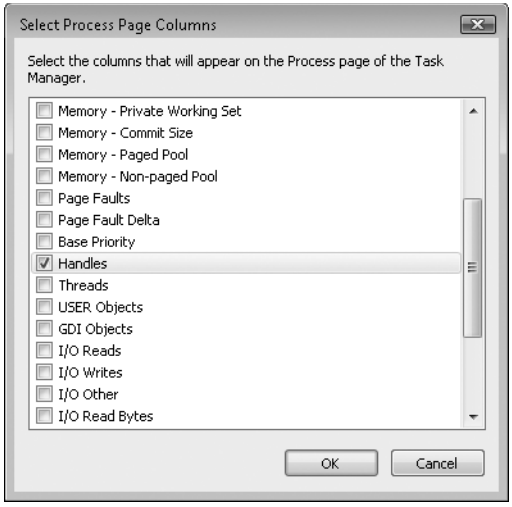


Figure 3-1 Selecting the Handles column in the Select Process Page Columns dialog box

Then you can monitor the number of kernel objects used by any application, as shown in Figure 3-2.

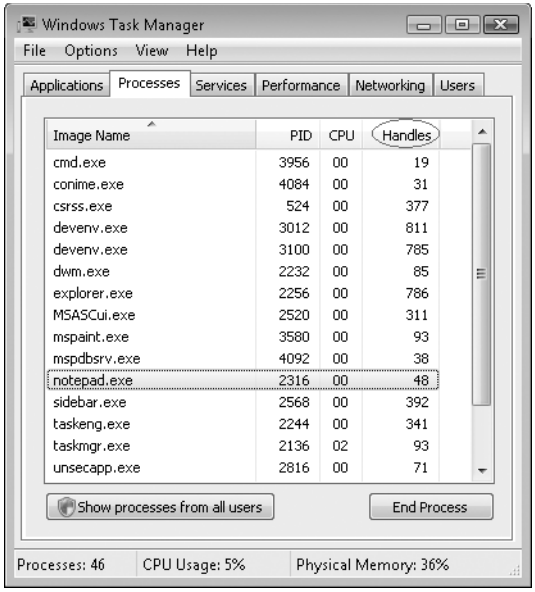


Figure 3-2 Counting handles in Windows Task Manager

If the number in the Handles column grows continuously, the next step to identifying which kernel objects are not closed is to take advantage of the free Process Explorer tool from Sysinternals (available at *http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx*). First, right-click in the header of the lower Handles pane. Then, in the Select Columns dialog box shown in Figure 3-3, select all the columns.

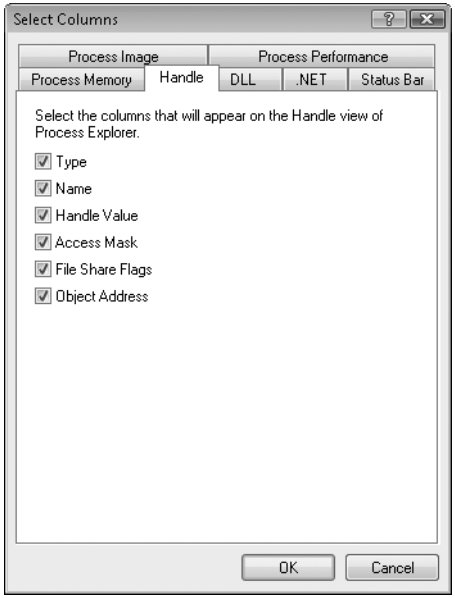


Figure 3-3 Selecting details for the Handle view in Process Explorer

Once this is done, change the Update Speed to Paused on the View menu. Select your process in the upper pane, and press F5 to get the up-to-date list of kernel objects. Execute the workflow you need to validate with your application, and once it is finished, press F5 again in Process Explorer. Each new kernel object is displayed in green, which you see represented as a darker gray in Figure 3-4.

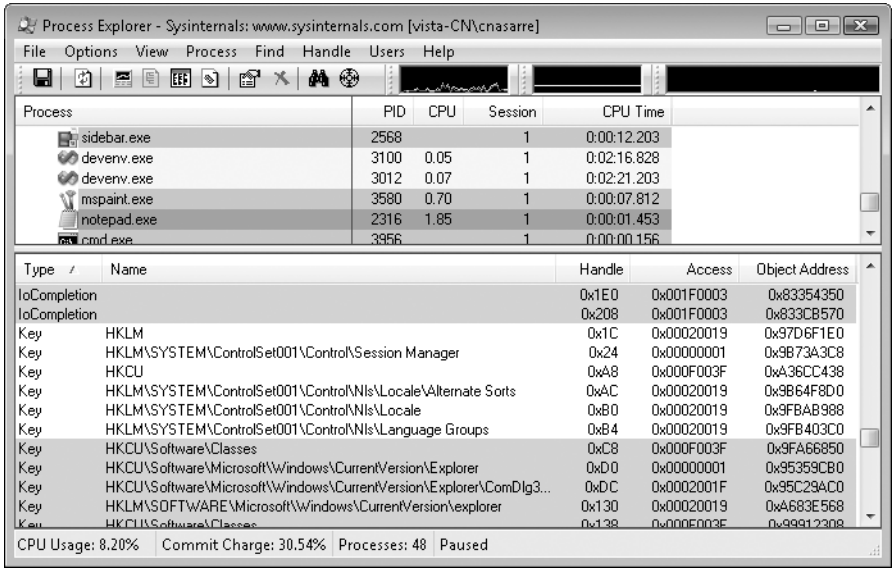


Figure 3-4 Detecting new kernel objects in Process Explorer

Note that the first column gives you the type of the kernel object that is not closed. To enhance your chances of finding where the leaks are, the second column provides the name of the kernel object. As you'll see in the next section, the string you give as a name to a kernel object allows you to share this object among processes. Obviously, it also helps you to figure out much more easily which object is not closed based on its type (first column) and its name (second column). If you are leaking lots of objects, they are probably not named because you can create only one instance of a named object—other attempts just open it.

Sharing Kernel Objects Across Process Boundaries

Frequently, threads running in different processes need to share kernel objects. Here are some of the reasons why:

- File-mapping objects allow you to share blocks of data between two processes running on a single machine.
- Mailslots and named pipes allow applications to send blocks of data between processes running on different machines connected to the network.
- Mutexes, semaphores, and events allow threads in different processes to synchronize their continued execution, as in the case of an application that needs to notify another application when it has completed some task.

Because kernel object handles are process-relative, performing these tasks is difficult. However, Microsoft had several good reasons for designing the handles to be process-relative. The most important reason was robustness. If kernel object handles were systemwide values, one process could easily obtain the handle to an object that another process was using and wreak havoc on that process. Another reason for process-relative handles is security. Kernel objects are protected with security, and a process must request permission to manipulate an object before attempting to manipulate it. The creator of the object can prevent an unauthorized user from touching the object simply by denying access to it.

In the following section, we'll look at the three different mechanisms that allow processes to share kernel objects: using object handle inheritance, naming objects, and duplicating object handles.

Using Object Handle Inheritance

Object handle inheritance can be used only when processes have a parent-child relationship. In this scenario, one or more kernel object handles are available to the parent process, and the parent decides to spawn a child process, giving the child access to the parent's kernel objects. For this type of inheritance to work, the parent process must perform several steps.

First, when the parent process creates a kernel object, the parent must indicate to the system that it wants the object's handle to be inheritable. Sometimes I hear people use the term *object inheritance*. However, there is no such thing as object inheritance; Windows supports *object handle inheritance*. In other words, it is the handles that are inheritable, not the objects themselves.

To create an inheritable handle, the parent process must allocate and initialize a **SECURITY_ATTRIBUTES** structure and pass the structure's address to the specific **Create** function. The following code creates a mutex object and returns an inheritable handle to it:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;   // Make the returned handle inheritable.

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);
```

This code initializes a **SECURITY_ATTRIBUTES** structure indicating that the object should be created using default security and that the returned handle should be inheritable.

Now we come to the flags that are stored in a process' handle table entry. Each handle table entry has a flag bit indicating whether the handle is inheritable. If you pass **NULL** as the **PSECURITY_ATTRIBUTES** parameter when you create a kernel object, the handle returned is not inheritable and this bit is zero. Setting the **bInheritHandle** member to **TRUE** causes this flag bit to be set to 1.

Imagine a process' handle table that looks like the one shown in Table 3-2.

Table 3-2 Process' Handle Table Containing Two Valid Entries

Index	Pointer to Kernel Object Memory Block	Access Mask (DWORD of Flag Bits)	Flags
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(N/A)	(N/A)
3	0xF0000010	0x????????	0x00000001

Table 3-2 indicates that this process has access to two kernel objects (handles 1 and 3). Handle 1 is not inheritable, and handle 3 is inheritable.

The next step to perform when using object handle inheritance is for the parent process to spawn the child process. This is done using the **CreateProcess** function:

```
BOOL CreateProcess(
   PCTSTR pszApplicationName,
   PTSTR pszCommandLine,
   PSECURITY_ATTRIBUTES psaProcess,
   PSECURITY_ATTRIBUTES psaThread,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   PVOID pvEnvironment,
   PCTSTR pszCurrentDirectory,
   LPSTARTUPINFO pStartupInfo,
   PPROCESS_INFORMATION pProcessInformation);
```

We'll examine this function in detail in the next chapter, but for now I want to draw your attention to the **bInheritHandles** parameter. Usually, when you spawn a process, you pass **FALSE** for this parameter. This value tells the system that you do not want the child process to inherit the inheritable handles that are in the parent process' handle table.

If you pass **TRUE** for this parameter, however, the child inherits the parent's inheritable handle values. When you pass **TRUE**, the operating system creates the new child process but does not allow the child process to begin executing its code right away. Of course, the system creates a new, empty process handle table for the child process—just as it would for any new process. But because you passed **TRUE** to **CreateProcess**' **bInheritHandles** parameter, the system does one more thing: it walks the parent process' handle table, and for each entry it finds that contains a valid inheritable handle, the system copies the entry exactly into the child process' handle table. The entry is copied to the exact same position in the child process' handle table as in the parent's handle table. This fact is important because it means that the handle value that identifies a kernel object is identical in both the parent and child processes.

In addition to copying the handle table entry, the system increments the usage count of the kernel object because two processes are now using the object. For the kernel object to be destroyed, both the parent process and the child process must either call **CloseHandle** on the object or terminate. The child does not have to terminate first—but neither does the parent. In fact, the parent process can close its handle to the object immediately after the **CreateProcess** function returns without affecting the child's ability to manipulate the object.

Table 3-3 shows the child process' handle table immediately before the process is allowed to begin execution. You can see that entries 1 and 2 are not initialized and are therefore invalid handles for the child process to use. However, index 3 does identify a kernel object. In fact, it identifies the kernel object at address 0xF0000010—the same object as in the parent process' handle table.

Table 3-3 A Child Process' Handle Table After Inheriting the Parent Process' Inheritable Handle

Index	Pointer to Kernel Object Memory Block	Access Mask (DWORD of Flag Bits)	Flags
1	0x00000000	(N/A)	(N/A)
2	0x00000000	(N/A)	(N/A)
3	0xF0000010	0x????????	0x00000001

As you will see in Chapter 13, "Windows Memory Architecture," the content of kernel objects is stored in the kernel address space that is shared by all processes running on the system. For 32-bit systems, this is in memory between the following memory addresses: 0x80000000 and 0xFFFFFFFF. For 64-bit systems, this is in memory between the following memory addresses: 0x00000400'00000000 and 0xFFFFFFF'FFFFFFFF. The access mask is identical to the mask in the parent, and the flags are also identical. This means that if the child process were to spawn its own child process (a grandchild process of the parent) with the same **bInheritHandles** parameter of **CreateProcess** set to **TRUE**, this grandchild process would also inherit this kernel object handle with the same handle value, same access, and same flags, and the usage count on the object would again be incremented.

Be aware that object handle inheritance applies only at the time the child process is spawned. If the parent process were to create any new kernel objects with inheritable handles, an already-running child process would not inherit these new handles.

Object handle inheritance has one very strange characteristic: when you use it, the child has no idea that it has inherited any handles. Kernel object handle inheritance is useful only when the

child process documents the fact that it expects to be given access to a kernel object when spawned from another process. Usually, the parent and child applications are written by the same company; however, a different company can write the child application if that company documents what the child application expects.

By far, the most common way for a child process to determine the handle value of the kernel object that it's expecting is to have the handle value passed as a command-line argument to the child process. The child process' initialization code parses the command line (usually by calling **_stscanf_s**) and extracts the handle value. Once the child has the handle value, it has the same access to the object as its parent. Note that the only reason handle inheritance works is because the handle value of the shared kernel object is identical in both the parent process and the child process. This is why the parent process is able to pass the handle value as a command-line argument.

Of course, you can use other forms of interprocess communication to transfer an inherited kernel object handle value from the parent process into the child process. One technique is for the parent to wait for the child to complete initialization (using the **WaitForInputIdle** function discussed in Chapter 9, "Thread Synchronization with Kernel Objects"); then the parent can send or post a message to a window created by a thread in the child process.

Another technique is for the parent process to add an environment variable to its environment block. The variable's name would be something that the child process knows to look for, and the variable's value would be the handle value of the kernel object to be inherited. Then when the parent spawns the child process, the child process inherits the parent's environment variables and can easily call **GetEnvironmentVariable** to obtain the inherited object's handle value. This approach is excellent if the child process is going to spawn another child process, because the environment variables can be inherited again. The special case of a child process inheriting its parent console is detailed in the Microsoft Knowledge Base at *http://support.microsoft.com/kb/190351*.

Changing a Handle's Flags

Occasionally, you might encounter a situation in which a parent process creates a kernel object retrieving an inheritable handle and then spawns two child processes. The parent process wants only one child to inherit the kernel object handle. In other words, you might at times want to control which child processes inherit kernel object handles. To alter the inheritance flag of a kernel object handle, you can call the **SetHandleInformation** function:

```
BOOL SetHandleInformation(
   HANDLE hObject,
   DWORD dwMask,
   DWORD dwFlags);
```

As you can see, this function takes three parameters. The first, **hObject**, identifies a valid handle. The second parameter, **dwMask**, tells the function which flag or flags you want to change. Currently, two flags are associated with each handle:

```
#define HANDLE_FLAG_INHERIT             0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE  0x00000002
```

You can perform a bitwise OR on both of these flags together if you want to change each object's flags simultaneously. **SetHandleInformation**'s third parameter, **dwFlags**, indicates what you want to set the flags to. For example, to turn on the inheritance flag for a kernel object handle, do the following:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

To turn off this flag, do this:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, 0);
```

The **HANDLE_FLAG_PROTECT_FROM_CLOSE** flag tells the system that this handle should not be allowed to be closed:

```
SetHandleInformation(hObj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
   HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hObj);   // Exception is raised
```

When running under a debugger, if a thread attempts to close a protected handle, **CloseHandle** raises an exception. Outside the control of a debugger, **CloseHandle** simply returns **FALSE**. You rarely want to protect a handle from being closed. However, this flag might be useful if you had a process that spawned a child that in turn spawned a grandchild process. The parent process might be expecting the grandchild to inherit the object handle given to the immediate child. It is possible, however, that the immediate child might close the handle before spawning the grandchild. If this were to happen, the parent might not be able to communicate with the grandchild because the grandchild did not inherit the kernel object. By marking the handle as "protected from close," the grandchild has a better chance to inherit a handle to a valid and live object.

This approach has one flaw, however. The immediate child process might call the following code to turn off the **HANDLE_FLAG_PROTECT_FROM_CLOSE** flag and then close the handle:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hObj);
```

The parent process is gambling that the child process will not execute this code. Of course, the parent is also gambling that the child process will spawn the grandchild, so this bet is not that risky.

For the sake of completeness, I'll also mention the **GetHandleInformation** function:

```
BOOL GetHandleInformation(
   HANDLE hObject,
   PDWORD pdwFlags);
```

This function returns the current flag settings for the specified handle in the DWORD pointed to by **pdwFlags**. To see if a handle is inheritable, do the following:

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

Naming Objects

The second method available for sharing kernel objects across process boundaries is to name the objects. Many—though not all—kernel objects can be named. For example, all of the following functions create named kernel objects:

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL bInitialOwner,
   PCTSTR pszName);

HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   BOOL bInitialState,
   PCTSTR pszName);

HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);

HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   PCTSTR pszName);

HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);

HANDLE CreateJobObject(
   PSECURITY_ATTRIBUTES psa,
   PCTSTR pszName);
```

All these functions have a common last parameter, **pszName**. When you pass **NULL** for this parameter, you are indicating to the system that you want to create an unnamed (anonymous) kernel object. When you create an unnamed object, you can share the object across processes by using either inheritance (as discussed in the previous section) or **DuplicateHandle** (discussed in the next section). To share an object by name, you must give the object a name.

If you don't pass **NULL** for the **pszName** parameter, you should pass the address of a zero-terminated string name. This name can be up to **MAX_PATH** characters long (defined as 260). Unfortunately, Microsoft offers no guidance for assigning names to kernel objects. For example, if you attempt to create an object called "JeffObj," there's no guarantee that an object called "JeffObj" doesn't already exist. To make matters worse, all these objects share a single namespace even though they don't share the same type. Because of this, the following call to **CreateSemaphore** always returns **NULL**—because a mutex already exists with the same name:

```
HANDLE hMutex = CreateMutex(NULL, FALSE, TEXT("JeffObj"));
HANDLE hSem = CreateSemaphore(NULL, 1, 1, TEXT("JeffObj"));
DWORD dwErrorCode = GetLastError();
```

If you examine the value of **dwErrorCode** after executing the preceding code, you'll see a return code of 6 (**ERROR_INVALID_HANDLE**). This error code is not very descriptive, but what can you do?

Now that you know how to name an object, let's see how to share objects this way. Let's say that Process A starts up and calls the following function:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

This function call creates a new mutex kernel object and assigns it the name "JeffMutex". Notice that in Process A's handle, **hMutexProcessA** is not an inheritable handle—and it doesn't have to be when you're only naming objects.

Some time later, some process spawns Process B. Process B does not have to be a child of Process A; it might be spawned from Windows Explorer or any other application. The fact that Process B need not be a child of Process A is an advantage of using named objects instead of inheritance. When Process B starts executing, it executes the following code:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

When Process B's call to **CreateMutex** is made, the system first checks to find out whether a kernel object with the name "JeffMutex" already exists. Because an object with this name does exist, the kernel then checks the object type. Because we are attempting to create a mutex and the object with the name "JeffMutex" is also a mutex, the system then makes a security check to see whether the caller has full access to the object. If it does, the system locates an empty entry in Process B's handle table and initializes the entry to point to the existing kernel object. If the object types don't match or if the caller is denied access, **CreateMutex** fails (returns **NULL**).

Note Kernel object creation functions (such as **CreateSemaphore**) always return handles with a full access right. If you want to restrict the available access rights for a handle, you can take advantage of the extended versions of kernel object creation functions (with an **Ex** postfix) that accept an additional DWORD **dwDesiredAccess** parameter. For example, you can allow or disallow **ReleaseSemaphore** to be called on a semaphore handle by using or not **SEMAPHORE_MODIFY_STATE** in the call to **CreateSemaphoreEx**. Read the Windows SDK documentation for the details of the specific rights corresponding to each kind of kernel object at *http://msdn2.microsoft.com/en-us/library/ms686670.aspx*.

When Process B's call to **CreateMutex** is successful, a mutex is not actually created. Instead, Process B is simply assigned a process-relative handle value that identifies the existing mutex object in the kernel. Of course, because a new entry in Process B's handle table references this object, the mutex object's usage count is incremented. The object will not be destroyed until both Process A and Process B have closed their handles to the object. Notice that the handle values in the two processes are most likely going to be different values. This is OK. Process A will use its handle value, and Process B will use its own handle value to manipulate the one mutex kernel object.

Note When you have kernel objects sharing names, be aware of one extremely important detail. When Process B calls **CreateMutex**, it passes security attribute information and a second parameter to the function. These parameters are ignored if an object with the specified name already exists! An application can determine whether it did, in fact, create a new kernel object rather than simply open an existing object by calling **GetLastError** immediately after the call to the **Create*** function:

```
HANDLE hMutex = CreateMutex(&sa, FALSE, TEXT("JeffObj"));
if (GetLastError() == ERROR_ALREADY_EXISTS) {
   // Opened a handle to an existing object.
   // sa.lpSecurityDescriptor and the second parameter
   // (FALSE) are ignored.
} else {
   // Created a brand new object.
   // sa.lpSecurityDescriptor and the second parameter
   // (FALSE) are used to construct the object.
}
```

An alternative method exists for sharing objects by name. Instead of calling a **Create*** function, a process can call one of the **Open*** functions shown here:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenEvent(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenJobObject(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

Notice that all these functions have the same prototype. The last parameter, **pszName**, indicates the name of a kernel object. You cannot pass **NULL** for this parameter; you must pass the address of a

zero-terminated string. These functions search the single namespace of kernel objects attempting to find a match. If no kernel object with the specified name exists, the functions return **NULL** and **GetLastError** returns 2 (**ERROR_FILE_NOT_FOUND**). However, if a kernel object with the specified name does exist, but it has a different type, the functions return **NULL** and **GetLastError** returns 6 (**ERROR_INVALID_HANDLE**). And if it is the same type of object, the system then checks to see whether the requested access (via the **dwDesiredAccess** parameter) is allowed. If it is, the calling process' handle table is updated and the object's usage count is incremented. The returned handle will be inheritable if you pass **TRUE** for the **bInheritHandle** parameter.

The main difference between calling a **Create*** function versus calling an **Open*** function is that if the object doesn't already exist, the **Create*** function will create it, whereas the **Open*** function will simply fail.

As I mentioned earlier, Microsoft offers no real guidelines on how to create unique object names. In other words, it would be a problem if a user attempted to run two programs from different companies and each program attempted to create an object called "MyObject." For uniqueness, I recommend that you create a GUID and use the string representation of the GUID for your object names. You will see another way to ensure name uniqueness in "Private Namespaces" on page 53.

Named objects are commonly used to prevent multiple instances of an application from running. To do this, simply call a **Create*** function in your **_tmain** or **_tWinMain** function to create a named object. (It doesn't matter what type of object you create.) When the **Create*** function returns, call **GetLastError**. If **GetLastError** returns **ERROR_ALREADY_EXISTS**, another instance of your application is running and the new instance can exit. Here's some code that illustrates this:

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine,
   int nCmdShow) {
   HANDLE h = CreateMutex(NULL, FALSE,
      TEXT("{FA531CC1-0497-11d3-A180-00105A276C3E}"));
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // There is already an instance of this application running.
      // Close the object and immediately return.
      CloseHandle(h);
      return(0);
   }

   // This is the first instance of this application running.
   ...
   // Before exiting, close the object.
   CloseHandle(h);
   return(0);
}
```

Terminal Services Namespaces

Note that Terminal Services changes the preceding scenario a little bit. A machine running Terminal Services has multiple namespaces for kernel objects. There is one global namespace, which is used by kernel objects that are meant to be accessible by all client sessions. This namespace is mostly used by services. In addition, each client session has its own namespace. This arrangement keeps two or more sessions that are running the same application from trampling over each other—one session cannot access another session's objects even though the objects share the same name.

These scenarios are not just related to server machines because Remote Desktop and Fast User Switching features are also implemented by taking advantage of Terminal Services sessions.

Note Before any user logs in, the services are starting in the first session, which is noninterative. In Windows Vista, unlike previous version of Windows, as soon as a user logs in, the applications are started in a new session—different from Session 0, which is dedicated to services. That way, these core components of the system, which are usually running with high privileges, are more isolated from any malware started by an unfortunate user.

For service developers, necessarily running in a session different from their client application affects the naming convention for shared kernel objects. It is now mandatory to create objects to be shared with user applications in the global namespace. This is the same kind of issue that you face when you need to write a service that is supposed to communicate with applications that might run when different users are logged in to different sessions through Fast User Switching—the service can't assume that it is running in the same session as the user application. For more details on Session 0 isolation and the impact it has on service developers, you should read "Impact of Session 0 Isolation on Services and Drivers in Windows Vista," which is located at *http://www.microsoft.com/whdc/system/vista/services.mspx*.

If you have to know in which Terminal Services session your process is running, the **ProcessIdToSessionId** function (exported by kernel32.dll and declared in WinBase.h) is what you need, as shown in the following example:

```
DWORD processID = GetCurrentProcessId();
DWORD sessionID;
if (ProcessIdToSessionId(processID, &sessionID)) {
   tprintf(
      TEXT("Process '%u' runs in Terminal Services session '%u'"),
      processID, sessionID);
} else {
   // ProcessIdToSessionId might fail if you don't have enough rights
   // to access the process for which you pass the ID as parameter.
   // Notice that it is not the case here because we're using our own process ID.
   tprintf(
      TEXT("Unable to get Terminal Services session ID for process '%u'"),
      processID);
}
```

A service's named kernel objects always go in the global namespace. By default, in Terminal Services, an application's named kernel object goes in the session's namespace. However, it is possible to force the named object to go into the global namespace by prefixing the name with "Global\", as in the following example:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Global\\MyName"));
```

You can also explicitly state that you want a kernel object to go in the current session's namespace by prefixing the name with "Local\", as in the following example:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Local\\MyName"));
```

Microsoft considers *Global* and *Local* to be reserved keywords that you should not use in object names except to force a particular namespace. Microsoft also considers *Session* to be a reserved keyword. So, for example, you could use Session\<*current session ID*>\. However, it is not possible to create an object with a name in another session with the *Session* prefix—the function call fails, and **`GetLastError`** returns **`ERROR_ACCESS_DENIED`**.

Note All these reserved keywords are case sensitive.

Private Namespaces

When you create a kernel object, you can protect the access to it by passing a pointer to a **`SECURITY_ATTRIBUTES`** structure. However, prior to the release of Windows Vista, it was not possible to protect the name of a shared object against hijacking. Any process, even with the lowest privileges, is able to create an object with a given name. If you take the previous example where an application is using a named mutex to detect whether or not it is already started, you could very easily write another application that creates a kernel object with the same name. If it gets started before the singleton application, this application becomes a "none-gleton" because it will start and then always immediately exit, thinking that another instance of itself is already running. This is the base mechanism behind a couple of attacks known as Denial of Service (DoS) attacks. Notice that unnamed kernel objects are not subject to DoS attacks, and it is quite common for an application to use unnamed objects, even though they can't be shared between processes.

If you want to ensure that the kernel object names created by your own applications never conflict with any other application's names or are the subject of hijack attacks, you can define a custom prefix and use it as a private namespace as you do with *Global* and *Local*. The server process responsible for creating the kernel object defines a *boundary descriptor* that protects the namespace name itself.

The Singleton application 03-Singleton.exe (with the corresponding Singleton.cpp source code listed a bit later in the chapter) shows how to use private namespaces to implement the same singleton pattern presented earlier but in a more secure way. When you start the program, the window shown in Figure 3-5 appears.

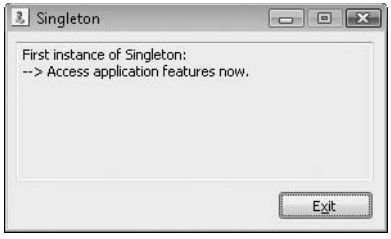


Figure 3-5 First instance of Singleton running

If you start the same program with the first one still running, the window shown in Figure 3-6 explains that the previous instance has been detected.

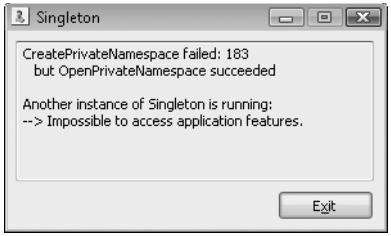


Figure 3-6 Second instance of Singleton when started while the first one is still running

The **CheckInstances** function in the following source code shows how to create a boundary, associate a *security identifier* (or SID) corresponding to the Local Administrators group with it, and create or open the private namespace whose name is used as a prefix by the mutex kernel object. The boundary descriptor gets a name, but more importantly, it gets a SID of a privileged user group that is associated with it. That way, Windows ensures that only the applications running under the context of a user that is a part of this privileged group is able to create the same namespace in the same boundary and thereby access the kernel objects created within this boundary that are prefixed with the private namespace name.

If a low-privileged malware application creates the same boundary descriptor because the name and the SID have been stolen, for example, when it tries to create or open the private namespace protected with a high-privileged account, the corresponding calls fail, with **GetLastError** returning **ERROR_ACCESS_DENIED**. If the malware application has enough privileges to create or open the namespace, worrying about this is not important because the malware application has enough control that it can cause much more damage than simply hijacking a kernel object name.

Singleton.cpp

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

#include "stdafx.h"
#include "resource.h"

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <Sddl.h>          // for SID management
#include <tchar.h>
#include <strsafe.h>



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


// Main dialog
HWND     g_hDlg;
```

```
// Mutex, boundary and namespace used to detect previous running instance
HANDLE   g_hSingleton = NULL;
HANDLE   g_hBoundary = NULL;
HANDLE   g_hNamespace = NULL;

// Keep track whether or not the namespace was created or open for clean-up
BOOL     g_bNamespaceOpened = FALSE;

// Names of boundary and private namespace
PCTSTR   g_szBoundary = TEXT("3-Boundary");
PCTSTR   g_szNamespace = TEXT("3-Namespace");


#define DETAILS_CTRL GetDlgItem(g_hDlg, IDC_EDIT_DETAILS)


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


// Adds a string to the "Details" edit control
void AddText(PCTSTR pszFormat, ...) {

   va_list argList;
   va_start(argList, pszFormat);

   TCHAR sz[20 * 1024];

   Edit_GetText(DETAILS_CTRL, sz, _countof(sz));
   _vstprintf_s(
      _tcschr(sz, TEXT('\0')), _countof(sz) - _tcslen(sz),
      pszFormat, argList);
   Edit_SetText(DETAILS_CTRL, sz);
   va_end(argList);
}


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


void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDOK:
      case IDCANCEL:
         // User has clicked on the Exit button
         // or dismissed the dialog with ESCAPE
         EndDialog(hwnd, id);
         break;
   }
}


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

```
void CheckInstances() {

   // Create the boundary descriptor
   g_hBoundary = CreateBoundaryDescriptor(g_szBoundary, 0);

   // Create a SID corresponding to the Local Administrator group
   BYTE localAdminSID[SECURITY_MAX_SID_SIZE];
   PSID pLocalAdminSID = &localAdminSID;
   DWORD cbSID = sizeof(localAdminSID);
   if (!CreateWellKnownSid(
      WinBuiltinAdministratorsSid, NULL, pLocalAdminSID, &cbSID)) {
      AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
         GetLastError());
      return;
   }

   // Associate the Local Admin SID to the boundary descriptor
   // --> only applications running under an administrator user
   //     will be able to access the kernel objects in the same namespace
   if (!AddSIDToBoundaryDescriptor(&g_hBoundary, pLocalAdminSID)) {
      AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
         GetLastError());
      return;
   }

   // Create the namespace for Local Administrators only
   SECURITY_ATTRIBUTES sa;
   sa.nLength = sizeof(sa);
   sa.bInheritHandle = FALSE;
   if (!ConvertStringSecurityDescriptorToSecurityDescriptor(
      TEXT("D:(A;;GA;;;BA)"),
      SDDL_REVISION_1, &sa.lpSecurityDescriptor, NULL)) {
      AddText(TEXT("Security Descriptor creation failed: %u\r\n"), GetLastError());
      return;
   }

   g_hNamespace =
      CreatePrivateNamespace(&sa, g_hBoundary, g_szNamespace);

   // Don't forget to release memory for the security descriptor
   LocalFree(sa.lpSecurityDescriptor);


   // Check the private namespace creation result
   DWORD dwLastError = GetLastError();
   if (g_hNamespace == NULL) {
      // Nothing to do if access is denied
      // --> this code must run under a Local Administrator account
      if (dwLastError == ERROR_ACCESS_DENIED) {
         AddText(TEXT("Access denied when creating the namespace.\r\n"));
         AddText(TEXT("   You must be running as Administrator.\r\n\r\n"));
         return;
```

```
      } else {
         if (dwLastError == ERROR_ALREADY_EXISTS) {
         // If another instance has already created the namespace,
         // we need to open it instead.
            AddText(TEXT("CreatePrivateNamespace failed: %u\r\n"), dwLastError);
            g_hNamespace = OpenPrivateNamespace(g_hBoundary, g_szNamespace);
            if (g_hNamespace == NULL) {
               AddText(TEXT("   and OpenPrivateNamespace failed: %u\r\n"),
                  dwLastError);
               return;
            } else {
               g_bNamespaceOpened = TRUE;
               AddText(TEXT("   but OpenPrivateNamespace succeeded\r\n\r\n"));
            }
         } else {
            AddText(TEXT("Unexpected error occurred: %u\r\n\r\n"),
               dwLastError);
            return;
         }
      }
   }

   // Try to create the mutex object with a name
   // based on the private namespace
   TCHAR szMutexName[64];
   StringCchPrintf(szMutexName, _countof(szMutexName), TEXT("%s\\%s"),
      g_szNamespace, TEXT("Singleton"));

   g_hSingleton = CreateMutex(NULL, FALSE, szMutexName);
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // There is already an instance of this Singleton object
      AddText(TEXT("Another instance of Singleton is running:\r\n"));
      AddText(TEXT("--> Impossible to access application features.\r\n"));
   } else  {
      // First time the Singleton object is created
      AddText(TEXT("First instance of Singleton:\r\n"));
      AddText(TEXT("--> Access application features now.\r\n"));
   }
}


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


BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SINGLETON);

   // Keep track of the main dialog window handle
   g_hDlg = hwnd;

   // Check whether another instance is already running
   CheckInstances();

   return(TRUE);
}
```

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


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

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
   }

   return(FALSE);
}


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


int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
   UNREFERENCED_PARAMETER(hPrevInstance);
   UNREFERENCED_PARAMETER(lpCmdLine);

   // Show main window
   DialogBox(hInstance, MAKEINTRESOURCE(IDD_SINGLETON), NULL, Dlg_Proc);

   // Don't forget to clean up and release kernel resources
   if (g_hSingleton != NULL) {
      CloseHandle(g_hSingleton);
   }

   if (g_hNamespace != NULL) {
      if (g_bNamespaceOpened) {  // Open namespace
         ClosePrivateNamespace(g_hNamespace, 0);
      } else { // Created namespace
         ClosePrivateNamespace(g_hNamespace, PRIVATE_NAMESPACE_FLAG_DESTROY);
      }
   }

   if (g_hBoundary != NULL) {
      DeleteBoundaryDescriptor(g_hBoundary);
   }

   return(0);
}

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

Let's examine the different steps of the **CheckInstances** function. First, the creation of a boundary descriptor requires a string identifier to name the scope where the private namespace will be defined. This name is passed as the first parameter of the following function:

```
HANDLE CreateBoundaryDescriptor(
   PCTSTR pszName,
   DWORD dwFlags);
```

Current versions of Windows do not use the second parameter, and therefore you should pass 0 for it. Note that the function signature implies that the return value is a kernel object handle; however, it is not. The return value is a pointer to a user-mode structure containing the definition of the boundary. For this reason, you should never pass the returned handle value to **CloseHandle**; you should pass it to **DeleteBoundaryDescriptor** instead.

The next step is to associate the SID of a privileged group of users that the client applications are supposed to run under with the boundary descriptor by calling the following function:

```
BOOL AddSIDToBoundaryDescriptor(
   HANDLE* phBoundaryDescriptor,
   PSID pRequiredSid);
```

In the Singleton example, the SID of the Local Administrator group is created by calling **AllocateAndInitializeSid** with **SECURITY_BUILTIN_DOMAIN_RID** and **DOMAIN_ALIAS_RID_ADMINS** as parameters that describe the group. The list of all well-known groups is defined in the WinNT.h header file.

This boundary descriptor handle is passed as the second parameter when you call the following function to create the private namespace:

```
HANDLE CreatePrivateNamespace(
   PSECURITY_ATTRIBUTES psa,
   PVOID pvBoundaryDescriptor,
   PCTSTR pszAliasPrefix);
```

The **SECURITY_ATTRIBUTES** that you pass as the first parameter to this function is used by Windows to allow or not allow an application calling **OpenPrivateNamespace** to access the namespace and open or create objects within that namespace. You have exactly the same options as within a file system directory. This is the level of filter that you provide for opening the namespace. The SID you added to the boundary descriptor is used to define who is able to enter the boundary and create the namespace. In the Singleton example, the **SECURITY_ATTRIBUTE** is constructed by calling the **ConvertStringSecurityDescriptorToSecurityDescriptor** function that takes a string with a complicated syntax as the first parameter. The security descriptor string syntax is documented at *http://msdn2.microsoft.com/en-us/library/aa374928.aspx* and *http://msdn2.microsoft.com/en-us/library/aa379602.aspx*.

The type of **pvBoundaryDescriptor** is **PVOID**, even though **CreateBoundaryDescriptor** returns a **HANDLE**—even at Microsoft it is seen as a pseudohandle. The string prefix you want to use to create your kernel objects is given as the third parameter. If you try to create a private namespace that already exists, **CreatePrivateNamespace** returns **NULL** and **GetLastError**