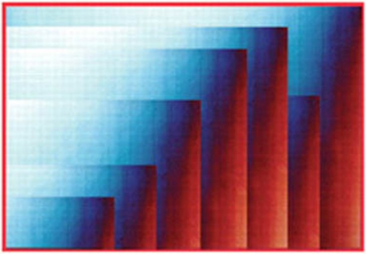
"This book fills a need for the serious developer who is creating advanced Excel applications. In it, Stephen, Rob and John share their many years of expertise, tackling complex issues with detailed information, clarity and an abundance of best practices."

—**Eric Patterson,** Program Manager, Microsoft Excel Team, Microsoft Corporation

Professional Excel Development

The Definitive Guide to Developing Applications Using Microsoft® Excel and VBA®



STEPHEN BULLEN
ROB BOVEY
JOHN GREEN

Praise for *Professional Excel Development*

"Think you know Microsoft Excel? Think again. This book covers intermediate (class modules, dictator applications, etc.) to advanced topics like XLLs, C APIs and Web Services. It offers plenty of easy to understand code listings that show exactly what the authors are trying to convey without forcing the readers to follow step-by-step."

Deepak Sharma, Sr. Systems Specialist, Tata Infotech Ltd.

"This book takes off where other Excel books stop. It covers Excel programming beyond VBA and looks at the professional issues—security, distribution, working with databases—using VB, VBA.NET and Windows API calls. The authors' depth and practical experience shows in the details. They explain complex issues clearly, describe best practices and point out traps to avoid."

Shauna Kelly, Microsoft Office MVP, `www.ShaunaKelly.com`

"The approach of following an application's development is very effective in developing the concepts as the chapters unfold. The practical, working examples used are relevant to many professional programmers."

Jan Karel Pieterse, JKP Application Development Services, `www.jkp-ads.com`

"This book stands out. While there are plenty of Excel books, I am not aware of any organized in this way. Information on .NET, and C, as well as other unique and useful chapters makes this a great offering."

Ken Bluttman, Author of Developing Microsoft Office Solutions

"This book explains difficult concepts in detail. The authors provide more than one method for complex development topics, along with the advantages and disadvantages of using the various methods described. They have my applause for the incorporation of development best practices."

Beth Melton, Microsoft Office MVP

This page intentionally left blank

Professional Excel Development

This page intentionally left blank

Professional Excel Development

The Definitive Guide to Developing Applications Using Microsoft Excel and VBA

Stephen Bullen, Rob Bovey, John Green

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Catalog Number: 2004114575

ISBN 0-321-26250-6
Text printed in the United States on recycled paper at Phoenix BookTech in Hagerstown, Maryland.
First printing, February, 2005

Contents

Acknowledgments

First and foremost, this book would never have been written without the support of our partners and families, who have graciously put up with our insatiable computer habits and many late nights over the past year. Neither would it have been done without our dogs, who kept our feet warm while we worked and forced us to get out of the house at least once each day.

We all owe a debt of gratitude to the Excel group at Microsoft, past and present, for making Excel the amazing development platform it is today. It is their dedication and commitment to us that makes Excel application development possible and enjoyable. They have repeatedly demonstrated their willingness to listen to and implement our suggestions over the years.

There are many people we want to thank at Addison-Wesley Professional, particularly Amy Fleischer for bringing us together, Stephane Thomas, Ebony Haight and Joan Murray for their support while writing the book, Kristy Hart for steering us through the production process and Curt Johnson for getting it on the shelves.

The quality of a technical book depends as much on the reviewers as the authors, so we want to thank all our technical reviewers. Most of your suggestions were implemented. At the risk of offending the others, we would particularly like to thank Dick Kusleika and John Peltier for the quality and rigor of their reviews and Beth Melton for finding numerous errors nobody else spotted.

Lastly, we want to thank you for buying this book. Please tell us what you think about it, either by e-mail or by writing a review at Amazon.com.

Thank you,

Stephen Bullen
Rob Bovey
John Green

This page intentionally left blank

About the Authors

Stephen Bullen

stephen@oaltd.co.uk

Stephen Bullen lives in Woodford Green, London, England, with his partner Clare, daughter Becky and their dog, Fluffy. A graduate of Oxford University, Stephen has an MA in Engineering, Economics and Management, providing a unique blend of both business and technical skills.

He has been providing Excel consulting and application development services since 1994, originally as an employee of Price Waterhouse Management Consultants and since 1997 as an independent consultant trading under the name of Business Modelling Solutions Limited. In September 2004, BMS changed its name to Office Automation Limited. If you would like to make use of Stephen's services, please contact him at stephen@oaltd.co.uk.

The Office Automation Web site, www.oaltd.co.uk, provides a number of helpful and interesting utilities, examples, tips and techniques to help in your use of Excel and development of Excel applications.

Stephen contributed chapters to John Green's *Excel 2000 VBA Programmer's Reference* and co-authored the sequel, *Excel 2002 VBA Programmer's Reference* (both published by Wrox Press).

In addition to his consulting and writing assignments, Stephen actively supports the Excel user community in Microsoft's peer-to-peer support newsgroups. In recognition of his knowledge, skills and contributions, Microsoft has awarded him the title of Most Valuable Professional each year since 1996.

Rob Bovey

robbovey@appspro.com

Rob Bovey is president of Application Professionals, a software development company specializing in Microsoft Office, Visual Basic, and SQL Server applications. He brings many years' experience creating financial, accounting and executive information systems for corporate users to Application Professionals. You can visit the Application Professionals Web site at `www.appspro.com`.

Rob developed several add-ins shipped by Microsoft for Microsoft Excel, co-authored the *Microsoft Excel 97 Developers Kit* and contributed to the *Excel 2002 VBA Programmer's Reference*. He earned his Bachelor of Science degree from The Rochester Institute of Technology and his MBA from the University of North Carolina at Chapel Hill. He is a Microsoft Certified Systems Engineer (MCSE) and a Microsoft Certified Solution Developer (MCSD). Microsoft has awarded him the title of Most Valuable Professional each year since 1995. He currently resides in Edmonds, Washington, with his wife Michelle, and their two black labs, Jasper and Jade.

John Green

greenj@bigpond.net.au

John Green lives and works in Sydney, Australia, as an independent computer consultant, specializing in integrating Excel, Access, Word and Outlook using VBA. He has more than 30 years of computing experience, a Chemical Engineering degree and an MBA.

He wrote his first programs in FORTRAN, took a part in the evolution of specialized planning languages on mainframes and, in the early 1980s, became interested in spreadsheet systems, including 1-2-3 and Excel.

John established his company, Execuplan Consulting, in 1980, developing computer-based planning applications and training users and developers.

John has had regular columns in a number of Australian magazines and has contributed chapters to a number of books, including *Excel Expert Solutions* and *Using Visual Basic for Applications 5*, published by Que. He is the principal author of *Excel 2000 VBA Programmer's Reference* and its subsequent editions, published by Wrox Press.

Since 1995 he has been accorded the status of Most Valuable Professional by Microsoft for his contributions to the CompuServe Excel forum and MS Internet newsgroups.

CHAPTER 9

Understanding and Using Windows API Calls

In the *Programming with the Windows API* chapter of our *Excel 2002 VBA Programmers Reference*, we approached the subject of using Windows API calls by explaining how to locate the definitions for various functions on the MSDN Web site and translate those functions for use in VBA. The idea was to enable readers to browse through the API documentation and use anything of interest they found.

In reality, extremely few people use Windows API calls in that manner; indeed, trying to include previously unexplored API calls in our Excel applications is very likely to result in a maintenance problem, because it's doubtful that another developer will understand what we were trying to do. Instead, most of us go to Google and search the Web or the newsgroups for the answer to a problem and find that the solution requires the use of API calls. (Searching Google for "Excel Windows API" results in more than 200,000 Web pages and 19,000 newsgroup posts.) We copy the solution into our application and hope it works, usually without really understanding what it does. This chapter shines a light on many of those solutions, explaining how they work, what they use the API calls for, and how they can be modified to better fit our applications. Along the way, we fill in some of the conceptual framework of common Windows API techniques and terminology.

By the end of the chapter, you will be comfortable about including API calls in your applications, understand how they work, accept their use in the example applications we develop in this book and be able to modify them to suit your needs.

Overview

When developing Excel-based applications, we can get most things done by using the Excel object model. Occasionally, though, we need some information or feature that Excel doesn't provide. In those cases, we can usually go directly to the files that comprise the Windows operating system to find what we're looking for. The first step in doing that is to tell VBA the function exists, where to find it, what arguments it takes and what data type it returns. This is done using the Declare statement, such as that for GetSystemMetrics:

```
Declare Function GetSystemMetrics Lib "user32" _
                   (ByVal nIndex As Long) As Long
```

This statement tells the VBA interpreter that there is a function called GetSystemMetrics located in the file user32.exe (or user32.dll, it'll check both) that takes one argument of a Long value and returns a Long value. Once defined, we can call GetSystemMetrics in exactly the same way as if it is the VBA function:

```
Function GetSystemMetrics(ByVal nIndex As Long) As Long
End Function
```

The Declare statements can be used in any type of code module, can be Public or Private (just like standard procedures), but must always be placed in the Declarations section at the top of the module.

Finding Documentation

All of the functions in the Windows API are fully documented in the *Windows Development/Platform SDK* section of the MSDN library on the Microsoft Web site, at `http://msdn.microsoft.com/library`, although the terminology used and the code samples tend to be targeted at the C++ developer. A Google search will usually locate documentation more appropriate for the Visual Basic and VBA developer, but is unlikely to be as complete as MSDN. If you're using API calls found on a Web site, the Web page will hopefully explain what they do, but it is a good idea to always check the official documentation for the functions to see whether any limitations or other remarks may affect your usage.

Unfortunately, the MSDN library's search engine is significantly worse than using Google to search the MSDN site. We find that Google always gives us more relevant pages than MSDN's search engine. To use Google to search MSDN, browse to `http://www.google.com` and click the Advanced Search link. Type in the search criteria and then in the Domain edit box type msdn.microsoft.com to restrict the search to MSDN.

Finding Declarations

It is not uncommon to encounter code snippets on the Internet that include incorrect declarations for API functions—such as declaring an argument's data type as Integer or Boolean when it should be Long. Although using the declaration included in the snippet will probably work (hopefully the author tested it), it might not work for the full range of possible arguments that the function accepts and in rare cases may cause memory corruption and data loss. The official VBA-friendly declarations for many of the more commonly used API functions can be found in the win32api.txt file, which is included with a viewer in the Developer Editions of Office 97–2002, Visual Basic 6 and is available for download from `http://support.microsoft.com/?kbid=178020`. You'll notice from the download page that the file hasn't been updated for some time. It therefore doesn't include the declarations and constants added in recent versions of Windows. If you're using one of those newer declarations, you'll have to trust the Web page author, examine a number of Web pages to check that they all use the same declaration or create your own VBA-friendly declaration by following the steps we described in the *Excel 2002 VBA Programmers Reference*.

Finding the Values of Constants

Most API functions are passed constants to modify their behavior or specify the type of value to return. For example, the GetSystemMetrics function shown previously accepts a parameter to specify which metric we want, such as SM_CXSCREEN to get the width of the screen in pixels or SM_CYSCREEN to get the height. All of the appropriate constants are shown on the MSDN page for that declaration. For example, the GetSystemMetrics function is documented at `http://msdn.microsoft.com/library/en-us/sysinfo/base/getsystemmetrics.asp` and shows more than 70 valid constants.

Although many of the constants are included in the win32api.txt file mentioned earlier, it does not include constants added for recent versions of Windows. The best way to find these values is by downloading and installing the core Platform SDK from `http://www.microsoft.com/msdownload/platformsdk/sdkupdate/`. This includes all the C++ header files that were used to build the DLLs, in a subdirectory called *include*. The files in this directory can be searched using normal Windows file searching to find the file that contains the constant we're interested in. For example, searching for SM_CXSCREEN gives the file winuser.h. Opening that file and searching within it gives the following lines:

```
#define SM_CXSCREEN             0
#define SM_CYSCREEN             1
```

These constants can then be included in your VBA module by declaring them as Long variables with the values shown:

```
Const SM_CXSCREEN As Long = 0
Const SM_CYSCREEN As Long = 1
```

Sometimes, the values will be shown in hexadecimal form, such as 0x8000, which can be converted to VBA by replacing the *0x* with *&h* and adding a further *&* on the end, such that

```
#define KF_UP                0x8000
```

becomes

```
Const KF_UP As Long = &h8000&
```

Understanding Handles

Within VBA, we're used to setting a variable to reference an object using code like

```
Set wkbBackDrop = Workbooks("Backdrop.xls")
```

and releasing that reference by setting the variable to Nothing (or letting VBA do that for us when it goes out of scope at the end of the procedure). Under the covers, the thing that we see as the Backdrop.xls workbook is just an area of memory containing data structured in a specific way that only Excel understands. When we set the variable equal to that object, it is just given the memory location of that data structure. The Windows operating system works in a very similar way, but at a much more granular level; almost everything within Windows is maintained as a small data structure somewhere. If we want to work with the item that is represented by that structure (such as a window), we need to get a reference to it and pass that reference to the appropriate API function. These references are known as ***handles*** and are just ID numbers that Windows uses to identify the data structure. Variables used to store handles are usually given the prefix h and are declared As Long.

When we ask for the handle to an item, some functions—such as FindWindow—give us the handle to a shared data structure; there is only one data structure for each window, so every call to FindWindow with the same parameters will return the same handle. In these cases, we can just discard the handle when we're finished with it. In most situations, however, Windows allocates an area of memory, creates a new data structure for us to use and returns the handle to that structure. In these cases, we ***must*** tidy up after ourselves, by explicitly telling Windows that we've finished using the handle (and by implication, the memory used to store the data structure that the handle points to). If we fail to tidy up correctly, each call to our routine will use another bit of memory until Windows crashes—this is known as a ***memory leak***. The most common cause of memory leaks is forgetting to include tidy-up code within a routine's error handler. The MSDN documentation will tell you whether you need to release the handle and which function to call to do it.

Encapsulating API Calls

GetSystemMetrics is one of the few API calls that can easily be used in isolation—it has a meaningful name, takes a single parameter, returns a simple result and doesn't require any preparation or cleanup. So long as you can remember what SM_CXSCREEN is asking for, it's extremely easy to call this function; `GetSystemMetrics(SM_CXSCREEN)` gives us the width of the screen in pixels.

In general practice, however, it is a very good idea to wrap your API calls inside their own VBA functions and to place those functions in modules dedicated to specific areas of the Windows API, for the following reasons:

- The VBA routine can include some validity checks before trying to call the API function. Passing invalid data to API functions will often result in a crash.
- Most of the textual API functions require string variables to be defined and passed in, which are then populated by the API function. Using a VBA routine hides that complexity.

- Many API functions accept parameters that we don't need to use. A VBA routine can expose only the parameters that are applicable to our application.
- Few API functions can be used in isolation; most require extra preparatory and clean up calls. Using a VBA routine hides that complexity.
- The API declarations themselves can be declared Private to the module in which they're contained, so they can be hidden from use by other developers who may not understand how to use them; their functionality can then be exposed through more friendly VBA routines.
- Some API functions, such as the encryption or Internet functions, require an initial set of preparatory calls to open resources, a number of routines that use those resources and a final set of routines to close the resources and tidy up. Such routines are ideally encapsulated in a class module, with the Class_Initialize and Class_Terminate procedures used to ensure the resources are opened and closed correctly.
- By using dedicated modules for specific areas of the Windows API, we can easily copy the routines between applications, in the knowledge that they are self-contained.

When you start to include lots of API calls in your application, it quickly becomes difficult to keep track of which constants belong to which functions. We can make the constants much easier to manage if we encapsulate them in an enumeration and use that enumeration for our VBA function's parameter, as shown in Listing 9-1. By doing this, the applicable constants are shown in the Intellisense list when the VBA function is used, as shown in Figure 9-1. The ability to define enumerations was added in Excel 2000.

Listing 9-1 Encapsulating the GetSystemMetrics API Function and Related Constants

```
'Declare all the API-specific items Private to the module
Private Declare Function GetSystemMetrics Lib "user32" _
        (ByVal nIndex As Long) As Long
Private Const SM_CXSCREEN As Long = 0
Private Const SM_CYSCREEN As Long = 1

'Wrap the API constants in a public enumeration,
```

```
'so they appear in the Intellisense dropdown
Public Enum SystemMetricsConstants
  smScreenWidth = SM_CXSCREEN
  smScreenHeight = SM_CYSCREEN
End Enum

'Wrapper for the GetSystemMetrics API function,
'using the SystemMetricsConstants enumeration
Public Function SystemMetrics( _
       ByVal uIndex As SystemMetricsConstants) As Long

  SystemMetrics = GetSystemMetrics(uIndex)
End Function
```

Figure 9-1 By Using the Enumeration, the Relevant Constants Appear in the Intellisense Drop-Down

Working with the Screen

The procedures included in this section all relate to the Windows screen and can be found in the MScreen module of the API Examples.xls workbook.

Reading the Screen Resolution

The GetSystemMetrics API function has been used to illustrate the general concepts above. It can be used to discover many of the simpler aspects of the operating system, from whether a mouse or network is present to the height of the standard window title bar. By far its most common use in Excel is to find the screen resolution, to check that it is at least a minimum size (for example, 800×600) or to work out which userform to display if you have different layouts optimized for different resolutions. The code in Listing 9-2 wraps the GetSystemMetrics API function, exposing it as separate ScreenWidth and ScreenHeight functions.

Listing 9-2 Reading the Screen Resolution

```
'Declare all the API-specific items Private to the module
Private Declare Function GetSystemMetrics Lib "user32" _
        (ByVal nIndex As Long) As Long
Private Const SM_CXSCREEN = 0     'Screen width
Private Const SM_CYSCREEN = 1     'Screen height

'The width of the screen, in pixels
Public Function ScreenWidth() As Long
  ScreenWidth = GetSystemMetrics(SM_CXSCREEN)
End Function

'The height of the screen, in pixels
Public Function ScreenHeight() As Long
  ScreenHeight = GetSystemMetrics(SM_CYSCREEN)
End Function
```

Finding the Size of a Pixel

In general, Excel measures distances in points, whereas most API functions use pixels and many ActiveX controls (such as the Microsoft Flexgrid) use twips. A point is defined as being 1/72 (logical) inches, and a twip is defined as 1/20th of a point. To convert between pixels and points, we need to know how many pixels Windows is displaying for each logical inch. This is the DPI (dots per inch) set by the user in *Control Panel > Display > Settings > Advanced > General > Display*, which is usually set at either Normal size (96 DPI) or Large size (120 DPI). In versions of Windows prior to XP, this was known as Small Fonts and Large Fonts. The value of this setting can be found using the GetDeviceCaps API function, which is used to examine the detailed capabilities of a specific graphical device, such as a screen or printer.

Device Contexts

One of the fundamental features of Windows is that applications can interact with all graphical devices (screens, printers, or even individual picture files) in a standard way. This is achieved by operating through a layer of indirection called a device context, which represents a drawing layer. An application obtains a reference (handle) to the drawing layer for a specific device (for example, the screen), examines its capabilities (such as the size of a dot, whether it can draw curves and how many colors it supports), draws onto the drawing layer and then releases the reference. Windows takes care of exactly how the drawing layer is represented on the graphical device. In this example, we're only examining the screen's capabilities.

The code to retrieve the size of a pixel is shown in Listing 9-3. Remember that when adding this code to an existing module, the declarations must always be placed at the top of the module.

Listing 9-3 Finding the Size of a Pixel

```
Private Declare Function GetDC Lib "user32" _
        (ByVal hwnd As Long) As Long

Private Declare Function GetDeviceCaps Lib "gdi32" _
        (ByVal hDC As Long, ByVal nIndex As Long) As Long

Private Declare Function ReleaseDC Lib "user32" _
        (ByVal hwnd As Long, ByVal hDC As Long) As Long

Private Const LOGPIXELSX = 88    'Pixels/inch in X

'A point is defined as 1/72 inches
Private Const POINTS_PER_INCH As Long = 72

'The size of a pixel, in points
Public Function PointsPerPixel() As Double

  Dim hDC As Long
  Dim lDotsPerInch As Long

  hDC = GetDC(0)
  lDotsPerInch = GetDeviceCaps(hDC, LOGPIXELSX)
  PointsPerPixel = POINTS_PER_INCH / lDotsPerInch
  ReleaseDC 0, hDC

End Function
```

The first thing to notice about this routine is that we cannot just call GetDeviceCaps directly; we need to give it a handle to the screen's device context. This handle is obtained by calling the GetDC function, where the zero parameter conveniently gives us the device context for the screen. We then call GetDeviceCaps, passing the constant LOGPIXELSX, which asks for the number of pixels per logical inch horizontally. (For screens, the horizontal and vertical DPI is the same, but it might not be for printers, which is why circles on screen often print out as ovals.) With Normal size chosen, we get 96 dots per inch. We divide the 72 points per inch by the 96 DPI, telling us that a dot (that is, pixel) is 0.75 points; so if we want to move something in Excel by one pixel, we need to change its Top or Left by 0.75. With Large Size selected, a pixel is 0.6 points.

Every time we use GetDC to obtain a handle to a device context, we use up a small amount of Window's graphical resources. If we didn't release the handle after using it, we would eventually use up all of Window's graphical resources and crash. To avoid that, we have to be sure to release any resources we obtain, in this case by calling ReleaseDC.

Working with Windows

Everything that we see on the screen is either a window or is contained within a window, from the Windows desktop to the smallest popup tooltip. Consequently, if we want to modify something on the screen, we always start by locating its window. The windows are organized into a hierarchy, with the desktop at the root. The next level down includes the main windows for all open applications and numerous system-related windows. Each application then owns and maintains its own hierarchy of windows. Every window is identified by its window handle, commonly referred to as ***hWnd***. By far the best tool for locating and examining windows is the Spy++ utility that is included with Visual Studio. Figure 9-2 shows the Spy++ display for the window hierarchy of a typical Excel session.

Window Classes

As well as showing the hierarchy, the Spy++ display shows three key attributes for each window: the handle (in hexadecimal), the caption and the class. Just like class modules, a window class defines a type of window. Some classes, such as the ComboBox class, are provided by the Windows operating system, but most are defined as part of an application. Each window class is usually associated with a specific part of an application, such as *XLMAIN* being Excel's main application window. Table 9-1 lists the window classes shown in the Spy++ hierarchy and their uses, plus some other window classes commonly encountered during Excel application development.

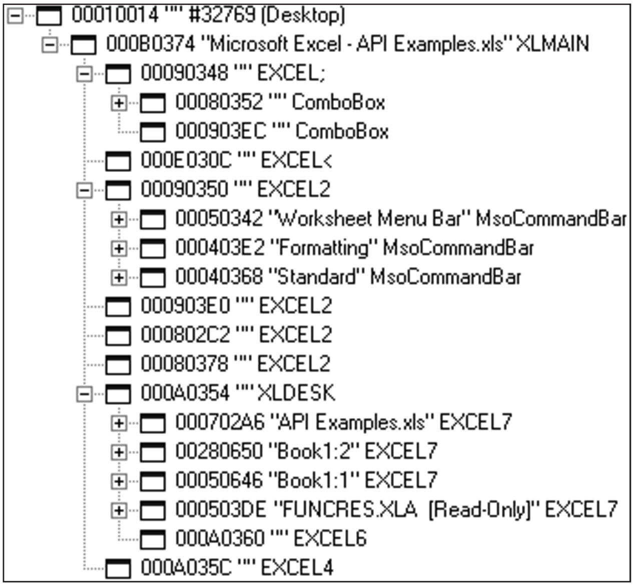


Figure 9-2 The Spy++ Display of the Excel Window Hierarchy

Table 9-1 Excel Window Classes and Their Uses

Window Class	Usage
XLMAIN	The main Excel application window.
EXCEL;	The left half of the formula bar, including the Name drop-down.
ComboBox	A standard Windows combo box (in this case, it's the Name drop-down).
EXCEL<	The edit box section of the formula bar.
EXCEL2	The four command bar docking areas (top, left, right and bottom).
MsoCommandBar	A command bar.
XLDESK	The Excel desktop.
EXCEL7	A workbook window. In this example, Book1 has two windows open.
EXCELE	A window used to provide in-sheet editing of embedded charts.
EXCEL4	The status bar.

Finding Windows

The procedures shown in the sections that follow can be found in the MWindows module of the API Examples.xls workbook.

To work with a window, we first need to find its handle. In Excel 2002, the hWnd property was added to the Application object, giving us the handle of the main Excel application window. In previous versions and for all other top-level windows (that is, windows that are direct children of the desktop), we can use the FindWindow API call, which is defined as follows:

```
Declare Function FindWindow Lib "user32" Alias "FindWindowA" _
        (ByVal lpClassName As String, _
         ByVal lpWindowName As String) As Long
```

To use the FindWindow function, we need to supply a class name and/or a window caption. We can use the special constant vbNullString for either, which tells the function to match on any class or caption. The function searches through all the immediate children of the desktop window (known as ***top-level windows***), looking for any that have the given class and/or caption that we specified. To find the main Excel window in versions prior to Excel 2002, we might use the following:

```
hWndExcel = FindWindow("XLMAIN", Application.Caption)
```

ANSI vs. Unicode and the Alias Clause

You might have noticed that the declaration for FindWindow contains an extra clause that we haven't used before—the ***Alias*** clause. All Windows API functions that have textual parameters come in two flavors: Those that operate on ANSI strings have an *A* suffix, whereas those that operate on Unicode strings have a *W* suffix. So while all the documentation and searches on MSDN talk about FindWindow, the Windows DLLs do not actually contain a function of that name—they contain two functions called FindWindowA and FindWindowW. We use the Alias statement to provide the actual name (case sensitive) for the function contained in the DLL. In fact, as long as we provide the correct name in the Alias clause, we can give it any name we like:

```
Declare Function Foo Lib "user32" Alias "FindWindowA" _
        (ByVal lpClassName As String, _
```

```
        ByVal lpWindowName As String) As Long

   ApphWnd = Foo("XLMAIN", Application.Caption)
```

Although VBA stores strings internally as Unicode, it always converts them to ANSI when passing them to API functions. This is usually sufficient, and it is quite rare to find examples of VB or VBA calling the Unicode versions. In some cases, however, we need to support the full Unicode character set and can work around VBA's conversion behavior by calling the W version of the API function and using StrConv to do an extra ANSI-to-Unicode conversion within our API function calls:

```
Declare Function FindWindow Lib "user32" Alias "FindWindowW" _
       (ByVal lpClassName As String, _
        ByVal lpWindowName As String) As Long

   ApphWnd = FindWindow(StrConv("XLMAIN", vbUnicode), _
            StrConv(Application.Caption, vbUnicode))
```

Finding Related Windows

The problem with the (very common) usage of FindWindow to get the main Excel window handle is that if we have multiple instances of Excel open that have the same caption, there is no easy way to tell which one we get, so we might end up modifying the wrong instance! It is a common problem if the user typically doesn't have his workbook windows maximized, because all instances of Excel will then have the same caption of "Microsoft Excel."

A more robust and foolproof method is to use the FindWindowEx function to scan through all children of the desktop window, stopping when we find one that belongs to the same process as our current instance of Excel. FindWindowEx works in exactly the same way as FindWindow, but we provide the parent window handle and the handle of a child window to start searching after (or zero to start with the first). Listing 9-4 shows a specific ApphWnd function, which calls a generic FindOurWindow function, which uses the following API functions:

- GetCurrentProcessID to retrieve the ID of the instance of Excel running the code

- GetDesktopWindow to get the handle of the desktop window, that we pass to FindWindowEx to look through its children (because all application windows are children of the desktop)
- FindWindowEx to find the next window that matches the given class and caption
- GetWindowThreadProcessID to retrieve the ID of the instance of Excel that owns the window that FindWindowEx found

Listing 9-4 Foolproof Way to Find the Excel Main Window Handle

```
'Get the handle of the desktop window
Declare Function GetDesktopWindow Lib "user32" () As Long

'Find a child window with a given class name and caption
Declare Function FindWindowEx Lib "user32" _
         Alias "FindWindowExA" _
        (ByVal hWnd1 As Long, ByVal hWnd2 As Long, _
         ByVal lpsz1 As String, ByVal lpsz2 As String) _
         As Long

'Get the process ID of this instance of Excel
Declare Function GetCurrentProcessId Lib "kernel32" () _
         As Long

'Get the ID of the process that a window belongs to
Declare Function GetWindowThreadProcessId Lib "user32" _
        (ByVal hWnd As Long, ByRef lpdwProcessId As Long) _
         As Long

'Foolproof way to find the main Excel window handle
Function ApphWnd() As Long

  'Excel 2002 and above have a property for the hWnd
  If Val(Application.Version) >= 10 Then
    ApphWnd = Application.hWnd
  Else
    ApphWnd = FindOurWindow("XLMAIN", Application.Caption)
  End If

End Function
```

```
'Finds a top-level window of the given class and caption
'that belongs to this instance of Excel, by matching the
'process IDs
Function FindOurWindow( _
         Optional sClass As String = vbNullString, _
         Optional sCaption As String = vbNullString)

  Dim hWndDesktop As Long
  Dim hWnd As Long
  Dim hProcThis As Long
  Dim hProcWindow As Long

  'Get the ID of this instance of Excel, to match to
  hProcThis = GetCurrentProcessId

  'All top-level windows are children of the desktop,
  'so get that handle first
  hWndDesktop = GetDesktopWindow

  Do
    'Find the next child window of the desktop that
    'matches the given window class and/or caption.
    'The first time in, hWnd will be zero, so we'll get
    'the first matching window. Each call will pass the
    'handle of the window we found the last time,
    'thereby getting the next one (if any)
    hWnd = FindWindowEx(hWndDesktop, hWnd, sClass, _
                        sCaption)

    'Get the ID of the process that owns the window
    GetWindowThreadProcessId hWnd, hProcWindow

    'Loop until the window's process matches this process,
    'or we didn't find a window
  Loop Until hProcWindow = hProcThis Or hWnd = 0

  'Return the handle we found
  FindOurWindow = hWnd

End Function
```

The FindOurWindow function can also be used to safely find any of the top-level windows that Excel creates, such as userforms.

After we've found Excel's main window handle, we can use the FindWindowEx function to navigate through Excel's window hierarchy. Listing 9-5 shows a function to return the handle of a given Excel workbook's window. To get the window handle, we start at Excel's main window, find the desktop (class XLDESK) and then find the window (class EXCEL7) with the appropriate caption.

Listing 9-5 Function to Find a Workbook's Window Handle

```
Private Declare Function FindWindowEx Lib "user32" _
        Alias "FindWindowExA" _
       (ByVal hWnd1 As Long, ByVal hWnd2 As Long, _
        ByVal lpsz1 As String, ByVal lpsz2 As String) _
        As Long

'Function to find the handle of a given workbook window
Function WorkbookWindowhWnd(wndWindow As Window) As Long

  Dim hWndExcel As Long
  Dim hWndDesk As Long

  'Get the main Excel window
  hWndExcel = ApphWnd

  'Find the desktop
  hWndDesk = FindWindowEx(hWndExcel, 0, _
                          "XLDESK", vbNullString)

  'Find the workbook window
  WorkbookWindowhWnd = FindWindowEx(hWndDesk, 0, _
                          "EXCEL7", wndWindow.Caption)

End Function
```

Windows Messages

At the lowest level, windows communicate with each other and with the operating system by sending simple messages. Every window has a main message-handling procedure (commonly called its wndproc) to which messages are sent. Every message consists of four elements: the handle of the window to which the message is being sent, a message ID and two numbers that provide extra information about the message (if required). Within each wndproc, there is a huge case statement that works out what to do for each message ID. For example, the system will send the WM_PAINT message to a window when it requires the window to redraw its contents.

It will probably come as no surprise that we can also send messages directly to individual windows, using the SendMessage function. The easiest way to find which messages can be sent to which window class is to search the MSDN library using a known constant and then look in the See Also list for a link to a list of related messages. Look down the list for a message that looks interesting, then go to its details page to see the parameters it requires. For example, if we look again at Figure 9-1, we can see that the EXCEL; window contains a combo box. This combo box is actually the Name drop-down to the left of the formula bar. Searching the MSDN library (using Google) with the search term "combo box messages" gives us a number of relevant hits. One of them takes us to `msdn.microsoft.com/library/en-us/shellcc/platform/commctls/comboboxes/comboboxes.asp`. Looking down the list of messages we find the CB_SETDROPPEDWIDTH message that we can use to change the width of the drop-down portion of the Name box. In Listing 9-6, we use the SendMessage function to make the Name drop-down 200 pixels wide, enabling us to see the full text of lengthy defined names.

Listing 9-6 Changing the Width of the Name Drop-Down List

```
Private Declare Function FindWindowEx Lib "user32" _
        Alias "FindWindowExA" _
       (ByVal hWnd1 As Long, ByVal hWnd2 As Long, _
        ByVal lpsz1 As String, ByVal lpsz2 As String) _
        As Long

Private Declare Function SendMessage Lib "user32" _
        Alias "SendMessageA" _
       (ByVal hwnd As Long, ByVal wMsg As Long, _
        ByVal wParam As Long, Byval lParam As Long) _
        As Long

'Not included in win32api.txt, but found in winuser.h
```

```
Private Const CB_SETDROPPEDWIDTH As Long = &H160&

'Make the Name dropdown list 200 pixels wide
Sub SetNameDropdownWidth()

  Dim hWndExcel As Long
  Dim hWndFormulaBar As Long
  Dim hWndNameCombo As Long

  'Get the main Excel window
  hWndExcel = ApphWnd

  'Get the handle for the formula bar window
  hWndFormulaBar = FindWindowEx(hWndExcel, 0, _
                  "EXCEL;", vbNullString)

  'Get the handle for the Name combobox
  hWndNameCombo = FindWindowEx(hWndFormulaBar, 0, _
                 "combobox", vbNullString)

  'Set the dropdown list to be 200 pixels wide
  SendMessage hWndNameCombo, CB_SETDROPPEDWIDTH, 200, 0

End Sub
```

Changing the Window Icon

When creating a dictator application, the intent is usually to make it look as though it is a normal Windows application and not necessarily running within Excel. Two of the giveaways are the application and worksheet icons. These can be changed to our own icons using API functions. We first use the ExtractIcon function to get a handle to an icon from a file, then send that icon handle to the window in a WM_SETICON message, as shown in Listing 9-7. The SetIcon routine is given a window handle and the path to an icon file, so it can be used to set either the application's icon or a workbook window's icon. For best use, the icon file should contain both 32×32 and 16×16 pixel versions of the icon image. Note that when setting the workbook window's icon, Excel doesn't refresh the image to the left of the menu bar until a window is maximized or minimized/restored, so you may need to toggle the WindowState to force the update.

Listing 9-7 Setting a Window's Icon

```
Private Declare Function ExtractIcon Lib "shell32.dll" _
        Alias "ExtractIconA" _
       (ByVal hInst As Long, _
        ByVal lpszExeFileName As String, _
        ByVal nIconIndex As Long) As Long

Private Declare Function SendMessage Lib "user32" _
        Alias "SendMessageA" _
       (ByVal hwnd As Long, ByVal wMsg As Long, _
        ByVal wParam As Long, Byval lParam As Long) _
        As Long

Private Const WM_SETICON As Long = &H80

'Set a window's icon
Sub SetIcon(ByVal hWnd As Long, ByVal sIcon As String)

  Dim hIcon As Long

  'Get the icon handle
  hIcon = ExtractIcon(0, sIcon, 0)

  'Set the big (32x32) and small (16x16) icons
  SendMessage hWnd, WM_SETICON, 1, hIcon
  SendMessage hWnd, WM_SETICON, 0, hIcon

End Sub
```

Changing Windows Styles

If you look at all the windows on your screen, you might notice that they all look a little different. Some have a title bar, some have minimize and maximize buttons, some have an [x] to close them, some have a 3D look, some are resizable, some are a fixed size and so on. All of these things are individual attributes of the window and are stored as part of the window's data structure. They're all on/off flags stored as bits in two Long numbers. We can use the GetWindowLong function to retrieve a window's style settings, switch individual bits on or off and write them back using SetWindowLong. Modifying windows styles in this way is most often done for userforms and is covered in *Chapter 10 — Userform Design and Best Practices*.

Working with the Keyboard

The behavior of many of Excel's toolbar buttons and some of the dialog buttons changes if the Shift key is held down when the button is clicked. For example, the Increase decimal toolbar button normally increases the number of decimal places shown in a cell, but decreases the number of decimal places if it is clicked with the Shift key held down. Similarly, when closing Excel, if you hold down the Shift key when clicking the No button on the Save Changes? dialog, it acts like a "No to All" button. We can do exactly the same in our applications by using API functions to examine the state of the keyboard. The procedures included in this section can be found in the MKeyboard module of the API Examples.xls workbook.

Checking for Shift, Ctrl, Alt, Caps Lock, Num Lock and Scroll Lock

The GetKeyState API function tells us whether a given key on the keyboard is currently held down or "on" (in the case of Caps Lock, Num Lock and Scroll Lock). The function is used by passing a code representing the key we're interested in and returns whether the key is being held down or is "on." Listing 9-8 shows a function to determine whether one of the six "special" keys is currently pressed. Note that we have again encapsulated the key code constants inside a more meaningful enumeration.

Listing 9-8 Checking Whether a Key Is Held Down

```
Private Declare Function GetKeyState Lib "user32" _
        (ByVal vKey As Long) As Integer

Private Const VK_SHIFT As Long = &H10
Private Const VK_CONTROL As Long = &H11
Private Const VK_MENU As Long = &H12
Private Const VK_CAPITAL = &H14
Private Const VK_NUMLOCK = &H90
Private Const VK_SCROLL = &H91

Public Enum GetKeyStateKeyboardCodes
  gksKeyboardShift = VK_SHIFT
  gksKeyboardCtrl = VK_CONTROL
  gksKeyboardAlt = VK_MENU
```

```
  gksKeyboardCapsLock = VK_CAPITAL
  gksKeyboardNumLock = VK_NUMLOCK
  gksKeyboardScrollLock = VK_SCROLL
End Enum

Public Function IsKeyPressed _
       (ByVal lKey As GetKeyStateKeyboardCodes) As Boolean

  Dim iResult As Integer

  iResult = GetKeyState(lKey)

  Select Case lKey
  Case gksKeyboardCapsLock, gksKeyboardNumLock, _
       gksKeyboardScrollLock

    'For the three 'toggle' keys, the 1st bit says if it's
    'on or off, so clear any other bits that might be set,
    'using a binary AND
    iResult = iResult And 1

  Case Else
    'For the other keys, the 16th bit says if it's down or
    'up, so clear any other bits that might be set, using a
    'binary AND
    iResult = iResult And &H8000
  End Select

  IsKeyPressed = (iResult <> 0)

End Function
```

Bit Masks

The value obtained from the call to GetKeyState should not be interpreted as a simple number, but as its binary representation where each individual bit represents whether a particular attribute is on or off. This is one of the few functions that return a 16-bit Integer value, rather than the more common 32-bit Long. The MSDN documentation for GetKeyState says that “If the high-order bit is 1, the key is down, otherwise the key is up. If the low-order bit is 1, the key is on, otherwise the key is off.” The first sentence is applicable for all keys (down/up), whereas the second is only applicable to the Caps Lock, Num Lock and Scroll Lock keys. It is possible for both bits to be set, if the Caps Lock key is held down and "on." The low-order bit is the rightmost bit, and the high-order bit is the leftmost (16th) bit. To examine whether a specific bit has been set, we have to apply a ***bit mask***, to zero-out the bits we're not interested in, by performing a binary AND between the return value and a binary value that has a single 1 in the position we're interested in. In the first case, we're checking for a 1 in the first bit, which is the number 1. In the second case, we're checking for a 1 in the 16th bit, i.e. the binary number 1000 0000 0000 0000, which is easiest to represent in code as the hexadecimal number &h8000. After we've isolated that bit, a zero value means off/up and a nonzero value means on/down.

Testing for a Key Press

As mentioned previously, at the lowest level, windows communicate through messages sent to their wndproc procedure. When an application is busy (such as Excel running some code), the wndproc only processes critical messages (such as the system shutting down). All other messages get placed in a queue and are processed when the application next has some spare time. This is why using SendKeys is so unreliable; it's not until the code stops running (or issues a DoEvents statement) that Excel checks its message queue to see whether there are any key presses to process.

We can use Excel's message queuing to allow the user to interrupt our code by pressing a key. Normally, if we want to allow the user to stop a lengthy looping process, we can either show a modeless dialog with a Cancel button (as explained in *Chapter 10 — Userform Design and Best Practices*), or allow the user to press the Cancel key to jump into the routine's error handler (as explained in *Chapter 12 — VBA Error Handling*). An easier way is to check Excel's message queue during each iteration of the loop to see whether the user has pressed a key. This is achieved using the PeekMessage API function:

```
Declare Function PeekMessage Lib "user32" _
         Alias "PeekMessageA" _
        (ByRef lpMsg As MSG, _
         ByVal hWnd As Long, _
         ByVal wMsgFilterMin As Long, _
         ByVal wMsgFilterMax As Long, _
         ByVal wRemoveMsg As Long) As Long
```

Structures

If you look at the first parameter of the PeekMessage function, you'll see it is declared As MSG and is passed ByRef. MSG is a windows ***structure*** and is implemented in VBA as a user-defined type. To use it in this case, we declare a variable of that type and pass it in to the function. The function sets the value of each element of the UDT, which we then read. Many API functions use structures as a convenient way of passing large amounts of information into the function, instead of having a long list of parameters. Many messages that we send using the SendMessage function require a structure to be passed as the final parameter (as opposed to a single Long value). In those cases, we use a different form of the SendMessage declaration, where the final parameter is declared As Any and is passed ByRef:

```
Declare Function SendMessageAny Lib "user32" _
         Alias "SendMessageA" _
        (ByVal hwnd As Long, ByVal wMsg As Long, _
         ByVal wParam As Long, _
         ByRef lParam As Any) As Long
```

When we use this declaration, we're actually sending a pointer to the memory where our UDT is stored. If we have an error in the definition of our UDT, or if we use this version of the declaration to send a message that is not expecting a memory pointer, the call will at best fail and possibly crash Excel.

Listing 9-9 shows the full code to check for a key press.

Listing 9-9 Testing for a Key Press

```
'Type to hold the coordinates of the mouse pointer
Private Type POINTAPI
  x As Long
  y As Long
End Type

'Type to hold the Windows message information
Private Type MSG
  hWnd As Long       'the window handle of the app
  message As Long    'the type of message (e.g. keydown)
  wParam As Long     'the key code
  lParam As Long     'not used
  time As Long       'time when message posted
```

```
  pt As POINTAPI    'coordinate of mouse pointer
End Type

'Look in the message buffer for a message
Private Declare Function PeekMessage Lib "user32" _
         Alias "PeekMessageA" _
        (ByRef lpMsg As MSG, ByVal hWnd As Long, _
         ByVal wMsgFilterMin As Long, _
         ByVal wMsgFilterMax As Long, _
         ByVal wRemoveMsg As Long) As Long

'Translate the message from a key code to a ASCII code
Private Declare Function TranslateMessage Lib "user32" _
        (ByRef lpMsg As MSG) As Long

'Windows API constants
Private Const WM_CHAR As Long = &H102
Private Const WM_KEYDOWN As Long = &H100
Private Const PM_REMOVE As Long = &H1
Private Const PM_NOYIELD As Long = &H2

'Check for a key press
Public Function CheckKeyboardBuffer() As String

  'Dimension variables
  Dim msgMessage As MSG
  Dim hWnd As Long
  Dim lResult As Long

  'Get the window handle of this application
  hWnd = ApphWnd

  'See if there are any "Key down" messages
  lResult = PeekMessage(msgMessage, hWnd, WM_KEYDOWN, _
            WM_KEYDOWN, PM_REMOVE + PM_NOYIELD)

  'If so ...
  If lResult <> 0 Then

    '... translate the key-down code to a character code,
    'which gets put back in the message queue as a WM_CHAR
    'message ...
    lResult = TranslateMessage(msgMessage)
```

```
    '... and retrieve that WM_CHAR message
    lResult = PeekMessage(msgMessage, hWnd, WM_CHAR, _
              WM_CHAR, PM_REMOVE + PM_NOYIELD)

    'Return the character of the key pressed,
    'ignoring shift and control characters
    CheckKeyboardBuffer = Chr$(msgMessage.wParam)
  End If

End Function
```

When we press a key on the keyboard, the active window is sent a WM_KEYDOWN message, with a low-level code to identify the physical key pressed. The first thing we need to do, then, is to use PeekMessage to look in the message queue to see whether there are any pending WM_KEYDOWN messages, removing it from the queue if we find one. If we found one, we have to translate it into a character code using TranslateMessage, which sends the translated message back to Excel's message queue as a WM_CHAR message. We then look in the message queue for this WM_CHAR message and return the character pressed.

Working with the File System and Network

The procedures included in this section can be found in the MFileSys module of the API Examples.xls workbook.

Finding the User ID

Excel has its own user name property, but does not tell us the user's network logon ID. This ID is often required in Excel applications for security validation, auditing, logging change history and so on. It can be retrieved using the API call shown in Listing 9-10.

Listing 9-10 Reading the User's Login ID

```
Private Declare Function GetUserName Lib "advapi32.dll" _
        Alias "GetUserNameA" _
       (ByVal lpBuffer As String, _
        ByRef nSize As Long) As Long
```

```
'Get the user's login ID
Function UserName() As String

  'A buffer that the API function fills with the login name
  Dim sBuffer As String * 255

  'Variable to hold the length of the buffer
  Dim lStringLength As Long

  'Initialize to the length of the string buffer
  lStringLength = Len(sBuffer)

  'Call the API function, which fills the buffer
  'and updates lStringLength with the length of the login ID,
  'including a terminating null - vbNullChar - character
  GetUserName sBuffer, lStringLength

  If lStringLength > 0 Then
    'Return the login id, stripping off the final vbNullChar
    UserName = Left$(sBuffer, lStringLength - 1)
  End If

End Function
```

Buffers

Every API function that returns textual information, such as the user name, does so by using a buffer that we provide. A buffer comprises a String variable initialized to a fixed size and a Long variable to tell the function how big the buffer is. When the function is called, it writes the text to the buffer (including a final Null character) and (usually) updates the length variable with the number of characters written. (Some functions return the text length as the function's result instead of updating the variable.) We can then look in the buffer for the required text. Note that VBA stores strings in a very different way than the API functions expect, so whenever we pass strings to API functions, VBA does some conversion for us behind the scenes. For this to work properly, we ***always*** pass strings by value (ByVal) to API functions, even when the function updates the string. Some people prefer to ignore the buffer length information, looking instead for the first vbNullChar character in the buffer and assuming that's the end of the retrieved string, so you may encounter usage like that shown in Listing 9-11.

Listing 9-11 Using a Buffer, Ignoring the Buffer Length Variable

```
'Get the user's login ID, without using the buffer length
Function UserName2() As String
  Dim sBuffer As String * 255
  GetUserName sBuffer, 255
  UserName2 = Left$(sBuffer, InStr(sBuffer, vbNullChar) - 1)
End Function
```

Changing to a UNC Path

VBA's intrinsic ChDrive and ChDir statements can be used to change the active path prior to using `Application.GetOpenFilename`, such that the dialog opens with the correct path preselected. Unfortunately, that can only be used to change the active path to local folders or network folders that have been mapped to a drive letter. Note that once set, the VBA CurDir function will return a UNC path. We need to use API functions to change the folder to a network path of the form \\server\share\path, as shown in Listing 9-12. In practice, the SetCurDir API function is one of the few that can be called directly from your code.

Listing 9-12 Changing to a UNC Path

```
Private Declare Function SetCurDir Lib "kernel32" _
        Alias "SetCurrentDirectoryA" _
       (ByVal lpPathName As String) As Long

'Change to a UNC Directory
Sub ChDirUNC(ByVal sPath As String)

  Dim lReturn As Long

  'Call the API function to set the current directory
  lReturn = SetCurDir(sPath)

  'A zero return value means an error
  If lReturn = 0 Then
    Err.Raise vbObjectError + 1, "Error setting path."
  End If

End Sub
```

Locating Special Folders

Windows maintains a large number of special folders that relate to either the current user or the system configuration. When a user is logged in to Windows with relatively low privileges, such as the basic User account, it is highly likely that the user will only have full access to his personal folders, such as his *My Documents* folder. These folders can usually be found under *C:\Documents and Settings\UserName*, but could be located anywhere. We can use an API function to give us the correct paths to these special folders, using the code shown in Listing 9-13. Note that this listing contains a subset of all the possible folder constants. The full list can be found by searching MSDN for "CSIDL Values." The notable exception from this list is the user's Temp folder, which can be found by using the GetTempPath function. Listing 9-13 includes a special case for this folder, so that it can be obtained through the same function.

Listing 9-13 Locating a Windows Special Folder

```
Private Declare Function SHGetFolderPath Lib "shell32" _
        Alias "SHGetFolderPathA" _
       (ByVal hwndOwner As Long, ByVal nFolder As Long, _
        ByVal hToken As Long, ByVal dwFlags As Long, _
        ByVal pszPath As String) As Long

Private Declare Function GetTempPath Lib "kernel32" _
        Alias "GetTempPathA" _
       (ByVal nBufferLength As Long, _
        ByVal lpBuffer As String) As Long

'More Commonly used CSIDL values.
'For the full list, search MSDN for "CSIDL Values"
Private Const CSIDL_PROGRAMS As Long = &H2
Private Const CSIDL_PERSONAL As Long = &H5
Private Const CSIDL_FAVORITES As Long = &H6
Private Const CSIDL_STARTMENU As Long = &HB
Private Const CSIDL_MYDOCUMENTS As Long = &HC
Private Const CSIDL_MYMUSIC As Long = &HD
Private Const CSIDL_MYVIDEO As Long = &HE
Private Const CSIDL_DESKTOPDIRECTORY As Long = &H10
Private Const CSIDL_APPDATA As Long = &H1A
Private Const CSIDL_LOCAL_APPDATA As Long = &H1C
Private Const CSIDL_INTERNET_CACHE As Long = &H20
```

```
Private Const CSIDL_WINDOWS As Long = &H24
Private Const CSIDL_SYSTEM As Long = &H25
Private Const CSIDL_PROGRAM_FILES As Long = &H26
Private Const CSIDL_MYPICTURES As Long = &H27

'Constants used in the SHGetFolderPath call
Private Const CSIDL_FLAG_CREATE As Long = &H8000&
Private Const SHGFP_TYPE_CURRENT = 0
Private Const SHGFP_TYPE_DEFAULT = 1
Private Const MAX_PATH = 260

'Public enumeration to give friendly names for the CSIDL values
Public Enum SpecialFolderIDs
  sfAppDataRoaming = CSIDL_APPDATA
  sfAppDataNonRoaming = CSIDL_LOCAL_APPDATA
  sfStartMenu = CSIDL_STARTMENU
  sfStartMenuPrograms = CSIDL_PROGRAMS
  sfMyDocuments = CSIDL_PERSONAL
  sfMyMusic = CSIDL_MYMUSIC
  sfMyPictures = CSIDL_MYPICTURES
  sfMyVideo = CSIDL_MYVIDEO
  sfFavorites = CSIDL_FAVORITES
  sfDesktopDir = CSIDL_DESKTOPDIRECTORY
  sfInternetCache = CSIDL_INTERNET_CACHE
  sfWindows = CSIDL_WINDOWS
  sfWindowsSystem = CSIDL_SYSTEM
  sfProgramFiles = CSIDL_PROGRAM_FILES

  'There is no CSIDL for the temp path,
  'so we need to give it a dummy value
  'and treat it differently in the function
  sfTemporary = &HFF
End Enum

'Get the path for a Windows special folder
Public Function SpecialFolderPath( _
       ByVal uFolderID As SpecialFolderIDs) As String

  'Create a buffer of the correct size
  Dim sBuffer As String * MAX_PATH
  Dim lResult As Long

  If uFolderID = sfTemporary Then
```

```
    'Use GetTempPath for the temporary path
    lResult = GetTempPath(MAX_PATH, sBuffer)

    'The GetTempPath call returns the length and a
    'trailing \ which we remove for consistency
    SpecialFolderPath = Left$(sBuffer, lResult - 1)
  Else
    'Call the function, passing the buffer
    lResult = SHGetFolderPath(0, _
              uFolderID + CSIDL_FLAG_CREATE, 0, _
              SHGFP_TYPE_CURRENT, sBuffer)

    'The SHGetFolderPath function doesn't give us a
    'length, so look for the first vbNullChar
    SpecialFolderPath = Left$(sBuffer, _
                        InStr(sBuffer, vbNullChar) - 1)
  End If

End Function
```

The observant among you might have noticed that we've now come across all three ways in which buffers are filled by API functions:

- GetUserName returns the length of the text by modifying the input parameter.
- GetTempPath returns the length of the text as the function's return value.
- SHGetFolderPath doesn't return the length at all, so we search for the first vbNullChar.

Deleting a File to the Recycle Bin

The VBA Kill statement is used to delete a file, but does not send it to the recycle bin for potential recovery by the user. To send a file to the recycle bin, we need to use the SHFileOperation function, as shown in Listing 9-14:

Listing 9-14 Deleting a File to the Recycle Bin

```
'Structure to tell the SHFileOperation function what to do
Private Type SHFILEOPSTRUCT
  hwnd As Long
```

```
  wFunc As Long
  pFrom As String
  pTo As String
  fFlags As Integer
  fAnyOperationsAborted As Boolean
  hNameMappings As Long
  lpszProgressTitle As String
End Type

Private Declare Function SHFileOperation Lib "shell32.dll" _
        Alias "SHFileOperationA" _
       (ByRef lpFileOp As SHFILEOPSTRUCT) As Long

Private Const FO_DELETE = &H3
Private Const FOF_SILENT = &H4
Private Const FOF_NOCONFIRMATION = &H10
Private Const FOF_ALLOWUNDO = &H40

'Delete a file, sending it to the recycle bin
Sub DeleteToRecycleBin(ByVal sFile As String)

    Dim uFileOperation As SHFILEOPSTRUCT
    Dim lReturn As Long

    'Fill the UDT with information about what to do
    With FileOperation
        .wFunc = FO_DELETE
        .pFrom = sFile
        .pTo = vbNullChar
        .fFlags = FOF_SILENT + FOF_NOCONFIRMATION + _
                  FOF_ALLOWUNDO
    End With

    'Pass the UDT to the function
    lReturn = SHFileOperation(FileOperation)

    If lReturn <> 0 Then
      Err.Raise vbObjectError + 1, "Error deleting file."
    End If

End Sub
```

There are two things to note about this function. First, the function uses a user-defined type to tell it what to do, instead of the more common method of having multiple input parameters. Second, the function returns a value of zero to indicate success. If you recall the SetCurDir function in Listing 9-12, it returns a value of zero to indicate failure! The only way to know which to expect is to check the Return Values section of the function's information page on MSDN.

Browsing for a Folder

All versions of Excel have included the GetOpenFilename and GetSaveAsFilename functions to allow the user to select a filename to open or save. Excel 2002 introduced the common Office FileDialog object, which can be used to browse for a folder, using the code shown in Listing 9-15, which results in the dialog shown in Figure 9-3.

Listing 9-15 Using Excel 2002's FileDialog to Browse for a Folder

```
'Browse for a folder, using the Excel 2002 FileDialog
Sub BrowseForFolder()

  Dim fdBrowser As FileDialog

  'Get the File Dialog object
  Set fdBrowser = Application.FileDialog(msoFileDialogFolderPicker)

  With fdBrowser

    'Initialize it
    .Title = "Select Folder"
    .InitialFileName = "c:\"

    'Display the dialog
    If .Show Then
      MsgBox "You selected " & .SelectedItems(1)
    End If
  End With

End Sub
```

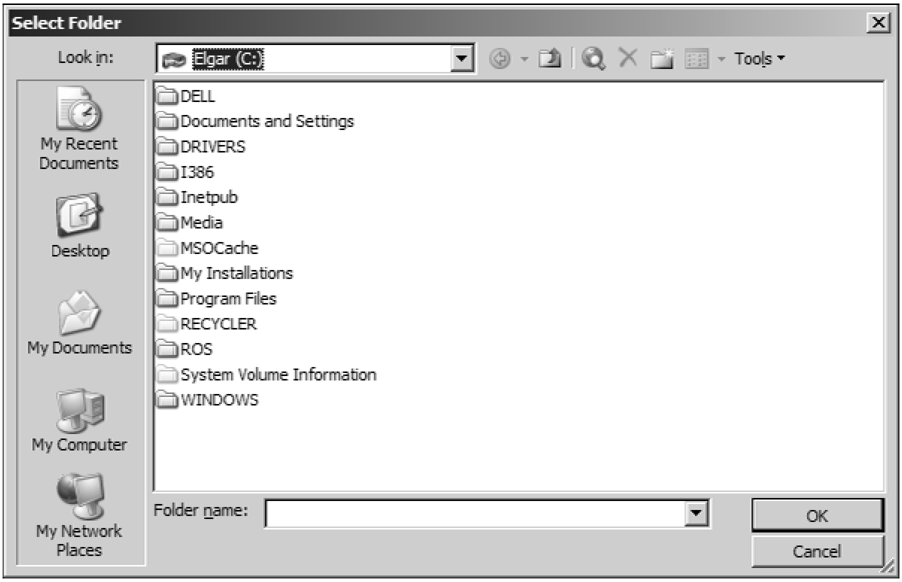


Figure 9-3 The Standard Office 2002 Folder Picker Dialog

We consider this layout far too complicated, when all we need is a simple tree view of the folders on the computer. We can use API functions to show the standard Windows Browse for folder dialog shown in Figure 9-4, which our users tend to find much easier to use. The Windows dialog also gives us the option to display some descriptive text to tell our users what they should be selecting.

Callbacks

So far, every function we've encountered just does its thing and returns its result. However, a range of API functions (including the SHBrowseForFolder function that we're about to use) interact with the calling program while they're working. This mechanism is known as a ***callback***. Excel 2000 added a VBA function called AddressOf, which provides the address in memory where a given procedure can be found. This address is passed to the API function, which calls back to the procedure found at that address as required. For example, the EnumWindows function iterates through all the top-level windows, calling back to the procedure with the details of each window it finds. Obviously, the procedure being called must be defined exactly as Windows expects it to be so the API function can pass it the correct number and type of parameters.

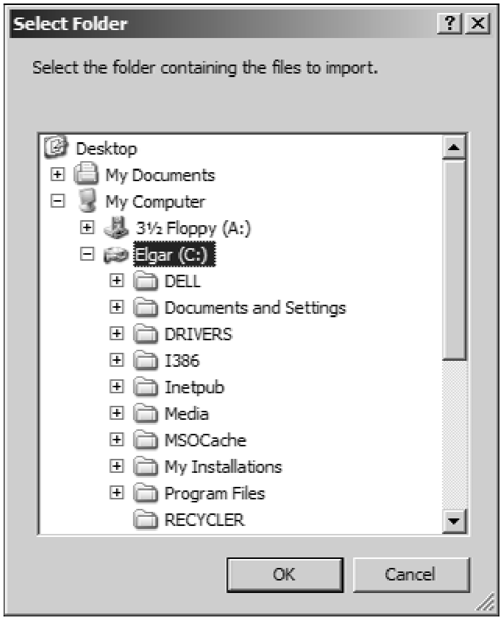


Figure 9-4 The Standard Windows Folder Picker Dialog

The SHBrowseForFolder function uses a callback to tell us when the dialog is initially shown, enabling us to set its caption and initial selection, and each time the user selects a folder, enabling us to check the selection and enable/disable the OK button. The full text for the function is contained in the MBrowseForFolder module of the API Examples.xls workbook and a slightly simplified version is shown in Listing 9-16.

Listing 9-16 Using Callbacks to Interact with the Windows File Picker Dialog

```
'UDT to pass information to the SHBrowseForFolder function
Private Type BROWSEINFO
  hOwner As Long
  pidlRoot As Long
  pszDisplayName As String
  lpszTitle As String
  ulFlags As Long
  lpfn As Long
  lParam As Long
  iImage As Long
End Type
```

```
'Commonly used ulFlags constants

'Only return file system directories.
'If the user selects folders that are not
'part of the file system (such as 'My Computer'),
'the OK button is grayed.
Private Const BIF_RETURNONLYFSDIRS As Long = &H1

'Use a newer dialog style, which gives a richer experience
Private Const BIF_NEWDIALOGSTYLE As Long = &H40

'Hide the default 'Make New Folder' button
Private Const BIF_NONEWFOLDERBUTTON As Long = &H200


'Messages sent from dialog to callback function

Private Const BFFM_INITIALIZED = 1
Private Const BFFM_SELCHANGED = 2


'Messages sent to browser from callback function
Private Const WM_USER = &H400

'Set the selected path
Private Const BFFM_SETSELECTIONA = WM_USER + 102

'Enable/disable the OK button
Private Const BFFM_ENABLEOK = WM_USER + 101


'The maximum allowed path
Private Const MAX_PATH = 260

'Main Browse for directory function
Declare Function SHBrowseForFolder Lib "shell32.dll" _
         Alias "SHBrowseForFolderA" _
        (ByRef lpBrowseInfo As BROWSEINFO) As Long

'Gets a path from a pidl
Declare Function SHGetPathFromIDList Lib "shell32.dll" _
         Alias "SHGetPathFromIDListA" _
        (ByVal pidl As Long, _
         ByVal pszPath As String) As Long
```

```
'Used to set the browse dialog's title
Declare Function SetWindowText Lib "user32" _
         Alias "SetWindowTextA" _
        (ByVal hwnd As Long, _
         ByVal lpString As String) As Long

'A versions of SendMessage, to send strings to the browser
Private Declare Function SendMessageString Lib "user32" _
         Alias "SendMessageA" (ByVal hwnd As Long, _
         ByVal wMsg As Long, ByVal wParam As Long, _
         ByVal lParam As String) As Long

'Variables to hold the initial options,
'set in the callback function
Dim msInitialPath As String
Dim msTitleBarText As String

'The main function to initialize and show the dialog
Function GetDirectory(Optional ByVal sInitDir As String, _
          Optional ByVal sTitle As String, _
          Optional ByVal sMessage As String, _
          Optional ByVal hwndOwner As Long, _
          Optional ByVal bAllowCreateFolder As Boolean) _
          As String

  'A variable to hold the UDT
  Dim uInfo As BROWSEINFO

  Dim sPath As String
  Dim lResult As Long

  'Check that the initial directory exists
  On Error Resume Next
  sPath = Dir(sInitDir & "\*.*", vbNormal + vbDirectory)
  If Len(sPath) = 0 Or Err.Number <> 0 Then sInitDir = ""
  On Error GoTo 0

  'Store the initials setting in module-level variables,
  'for use in the callback function
  msInitialPath = sInitDir
  msTitleBarText = sTitle

  'If no owner window given, use the Excel window
```

```
   'N.B. Uses the ApphWnd function in MWindows
   If hwndOwner = 0 Then hwndOwner = ApphWnd

   'Initialise the structure to pass to the API function
   With uInfo
      .hOwner = hwndOwner
      .pszDisplayName = String$(MAX_PATH, vbNullChar)
      .lpszTitle = sMessage
      .ulFlags = BIF_RETURNONLYFSDIRS + BIF_NEWDIALOGSTYLE _
         + IIf(bAllowCreateFolder, 0, BIF_NONEWFOLDERBUTTON)

      'Pass the address of the callback function in the UDT
      .lpfn = LongToLong(AddressOf BrowseCallBack)
   End With

   'Display the dialog, returning the ID of the selection
   lResult = SHBrowseForFolder(uInfo)

   'Get the path string from the ID
   GetDirectory = GetPathFromID(lResult)

End Function


'Windows calls this function when the dialog events occur
Private Function BrowseCallBack (ByVal hwnd As Long, _
        ByVal Msg As Long, ByVal lParam As Long, _
        ByVal pData As Long) As Long

   Dim sPath As String

   'This is called by Windows, so don't allow any errors!
   On Error Resume Next

   Select Case Msg
   Case BFFM_INITIALIZED
      'Dialog is being initialized,
      'so set the initial parameters

      'The dialog caption
      If msTitleBarText <> "" Then
        SetWindowText hwnd, msTitleBarText
      End If
```

```
    'The initial path to display
    If msInitialPath <> "" Then
      SendMessageString hwnd, BFFM_SETSELECTIONA, 1, _
                        msInitialPath
    End If

  Case BFFM_SELCHANGED
    'User selected a folder
    'lParam contains the pidl of the folder, which can be
    'converted to the path using GetPathFromID
    'sPath = GetPathFromID(lParam)

    'We could put extra checks in here,
    'e.g. to check if the folder contains any workbooks,
    'and send the BFFM_ENABLEOK message to enable/disable
    'the OK button:
    'SendMessage hwnd, BFFM_ENABLEOK, 0, True/False
  End Select

End Function


'Converts a PIDL to a path string
Private Function GetPathFromID(ByVal lID As Long) As String

  Dim lResult As Long
  Dim sPath As String * MAX_PATH

  lResult = SHGetPathFromIDList(lID, sPath)

  If lResult <> 0 Then
    GetPathFromID = Left$(sPath, InStr(sPath, Chr$(0)) - 1)
  End If

End Function

'VBA doesn't let us assign the result of AddressOf
'to a variable, but does allow us to pass it to a function.
'This 'do nothing' function works around that problem
Private Function LongToLong(ByVal lAddr As Long) As Long
  LongToLong = lAddr
End Function
```

Let's take a closer look at how this all works. First, most of the shell functions use things called PIDLs to uniquely identify folders and files. For simplicity's sake, you can think of a PIDL as a handle to a file or folder, and there are API functions to convert between the PIDL and the normal file or folder name.

The GetDirectory function is the main function in the module and is the function that should be called to display the dialog. It starts by validating the (optional) input parameters, then populates the BROWSEINFO user-defined type that is used to pass all the required information to the SHBrowseForFolder function. The ***hOwner*** element of the UDT is used to provide the parent window for the dialog, which should be the handle of the main Excel window, or the handle of the userform window if showing this dialog from a userform. The ***ulFlags*** element is used to specify detailed behavior for the dialog, such as whether to show a Make Folder button. The full list of possible flags and their purpose can be found on MSDN by searching for the SHBrowseForFolder function. The ***lpfn*** element is where we pass the address of the callback function, BrowseCallBack. We have to wrap the AddressOf value in a simple LongToLong function, because VB doesn't let us assign the value directly to an element of a UDT.

After the UDT has been initialized, we pass it to the SHBrowseForFolder API function. That function displays the dialog and Windows calls back to our BrowseCallBack function, passing the BFFM_INITIALIZED message. We respond to that message by setting the dialog's caption (using the SetWindowText API function) and the initial folder selection (by sending the BFFM_SETSELECTIONA message back to the dialog with the path string).

Every time the user clicks a folder, it triggers a Windows callback to our BrowseCallBack function, passing the BFFM_SELCHANGED message and the ID of the selected folder. All the code to respond to that message is commented out in this example, but we could add code to check whether the folder is a valid selection for our application (such as whether it contains any workbooks) and enable/disable the OK button appropriately (by sending the BFFM_ENABLEOK message back to the dialog).

When the user clicks the OK or Cancel button, the function returns the ID of the selected folder and execution continues back in the GetDirectory function. We get the textual path from the returned ID and return it to the calling code.

Practical Examples

All the routines included in this chapter have been taken out of actual Excel applications, so are themselves practical examples of API calls.

The PETRAS application files for this chapter can be found on the CD in the folder *\Application\Ch09—Understanding and Using Windows API Calls* and now includes the following files:

- **PetrasTemplate.xlt**—The timesheet template
- **PetrasAddin.xla**—The timesheet data-entry support add-in
- **PetrasReporting.xla**—The main reporting application
- **PetrasConsolidation.xlt**—A template to use for new results workbooks
- **Debug.ini**—A dummy file that tells the application to run in debug mode
- **PetrasIcon.ico**—A new icon file, to use for Excel's main window

PETRAS Timesheet

Until this chapter, the location used by the Post to Network routine has used Application.GetOpenFilename to allow the user to select the directory to save the timesheet workbook to. The problem with that call is that the directory must already contain at least one file. In this chapter, we add the BrowseForFolder dialog and use that instead of GetOpenFilename, which allows empty folders to be selected.

We've also added a new feature to the timesheet add-in. In previous versions you were prompted to specify the consolidation location the first time you posted a timesheet workbook to the network. When you selected a location, that location was stored in the registry and from there on out the application simply read the location from the registry whenever you posted a new timesheet.

What this didn't take into account is the possibility that the consolidation location might change. If it did, you would have no way, short of editing the application's registry entries directly, of switching to the new location. Our new Specify Consolidation Folder feature enables you to click a button on the toolbar and use the Windows browse for folders dialog to modify the consolidation folder. The SpecifyConsolidationFolder procedure is shown in Listing 9-17 and the updated toolbar is shown in Figure 9-5.

Listing 9-17 The New SpecifyConsolidationFolder Procedure

```
Public Sub SpecifyConsolidationFolder()

    Dim sSavePath As String

    InitGlobals

    ' Get the current consolidation path.
    sSavePath = GetSetting(gsREG_APP, gsREG_SECTION, _
            gsREG_KEY, "")

    ' Display the browse for folders dialog with the initial
    ' path display set to the current consolidation folder.
    sSavePath = GetDirectory(sSavePath, _
            gsCAPTION_SELECT_FOLDER, gsMSG_SELECT_FOLDER)

    If Len(sSavePath) > 0 Then
        ' Save the selected path to the registry.
        If Right$(sSavePath, 1) <> "\" Then _
            sSavePath = sSavePath & "\"
        SaveSetting gsREG_APP, gsREG_SECTION, _
            gsREG_KEY, sSavePath
    End If

End Sub
```

Table 9-2 summarizes the changes that have been made to the timesheet add-in for this chapter.

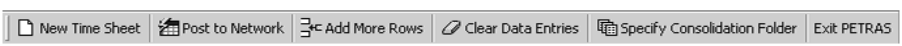


Figure 9-5 The Updated PETRAS Timesheet Toolbar

Table 9-2 Changes to the PETRAS Timesheet Add-in to Use the BrowseForFolder Routine

Module	Procedure	Change
MBrowseForFolder (new module)		Included the entire MBrowseForFolder module shown in Listing 9-16
MEntryPoints	PostTimeEntriesToNetwork	Added call to the GetDirectory function in MBrowseForFolder
	SpecifyConsolidationFolder	New feature to update the consolidation folder location

PETRAS Reporting

The changes made to the central reporting application for this chapter are to display a custom icon for the application and to enable the user to close all the results workbooks simultaneously, by holding down the Shift key while clicking the *File* > *Close* menu. The detailed changes are shown in Table 9-3, and Listing 9-18 shows the new MenuFileClose routine that includes the check for the Shift key.

Table 9-3 Changes to the PETRAS Reporting Application for Chapter 9

Module	Procedure	Change
MAPIWrappers (new module)	ApphWnd	Included Listing 9-4 to obtain the handle of Excel's main window
MAPIWrappers (new module)	SetIcon	Included Listing 9-7 to display a custom icon, read from the new PetrasIcon.ico file.
MAPIWrappers	IsKeyPressed	Included Listing 9-8 to check for the Shift key held down when clicking *File* > *Close*
MGlobals		Added a constant for the icon filename
MWorkspace	ConfigureExcelEnvironment	Added a call to SetIcon
MEntryPoints	MenuFileClose	Added check for Shift key being held down, shown in Listing 9-17, doing a Close All if so

Listing 9-18 The New MenuFileClose Routine, Checking for a Shift+Close

```
'Handle the File > Close menu
Sub MenuFileClose()

  Dim wkbWorkbook As Workbook

  'Ch09+
  'Check for a Shift+Close
  If IsKeyPressed(gksKeyboardShift) Then

    'Close all results workbooks
    For Each wkbWorkbook In Workbooks
      If IsResultsWorkbook(wkbWorkbook) Then
        CloseWorkbook wkbWorkbook
      End If
    Next
  Else
    'Ch09-

    'Close only the active workbook
    If IsResultsWorkbook(ActiveWorkbook) Then
      CloseWorkbook ActiveWorkbook
    End If
  End If

End Sub
```

Later chapters, particularly *Chapter 10 — Userform Design and Best Practices*, use more of the routines and concepts introduced in this chapter.

Conclusion

The Excel object model provides an extremely rich set of tools for us to use when creating our applications. By including calls to Windows API functions, we can enhance our applications to give them a truly professional look and feel.

This chapter has explained most of the uses of API functions that are commonly encountered in Excel application development. All the fundamental concepts have been explained and you should now be able to interpret and understand new uses of API functions as you encounter them.

All of the example routines included in this chapter have been taken from actual Excel applications and are ready for you to use in your own workbooks.

CHAPTER 15

Advanced Charting Techniques

Only a few minutes are required to learn the basics of Excel's charting module, but many frustrating hours are required to get a chart looking "just right." Most people create charts using one of the built-in chart types, but are unable to modify them to meet their exact requirements. This chapter introduces and explains the fundamental techniques we can use to impose our will on Excel's charting engine to produce charts that look exactly how we want them to.

The chapter focuses solely on the technical aspects of working with the chart engine. We do not investigate which chart type should be used in any given situation, nor the pros and cons of whether 3D charts can be used to present data accurately, nor whether you should use as few or as many of the colorful formatting options that Excel supports.

Fundamental Techniques

Combining Chart Types

When most people create charts, they start the Chart Wizard and browse through all the standard and custom chart types shown in Step 1, trying to find one that most closely resembles the look they're trying to achieve. More often than not, there isn't a close enough match and they end up thinking that Excel doesn't support the chart they're trying to create. In fact, we can include any number of column, bar, line, XY and/or area series within the same chart. All of the choices on the Custom Types tab of Step 1 of the Chart Wizard are no more than preformatted combinations of these basic styles, with a bit of formatting thrown in. Instead of relying on these custom types, we can usually get better results (and a greater understanding of the chart engine) by creating these combination charts ourselves. Unfortunately, we can't combine the different 3D styles, pie charts or bubble charts with other types.

Let's start by creating a simple column/line combination chart for the data shown in Figure 15-1, where we want the 2004 sales to be shown as columns, with the forecast shown as lines.

The easiest way to start is by selecting the data region, `A3:C8` and create a simple column chart from it, as shown in Figure 15-2. We usually find it easiest to start with a column chart, but perhaps that's because it's the default selection in the Chart Wizard, so we can create the chart by selecting the source data, clicking the Chart Wizard toolbar button and then the Finish button on the Chart Wizard.

	A	B	C
1			
2	2004 Fruit Sales		
3		Sales	Forecast
4	Apples	1000	900
5	Oranges	1200	1400
6	Peaches	600	800
7	Pears	700	1100
8	Bananas	1100	1000

Figure 15-1 The Sample Data to Plot as a Combination Column/Line Chart

	A	B	C	D	E	F	G
1							
2	2004 Fruit Sales						
3		Sales	Forecast				
4	Apples	1000	900				
5	Oranges	1200	1400				
6	Peaches	600	800				
7	Pears	700	1100				
8	Bananas	1100	1000				

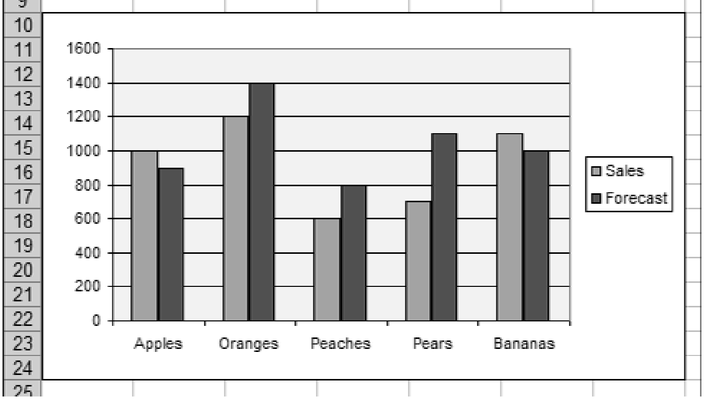


Figure 15-2 The Chart Wizard Created a Standard Column Chart

To change the Forecast values from a column to a line, select the series, click the *Chart > Chart Type* menu item and select one of the 2D Line chart types, choosing to apply the chart type to the selected series, as shown in Figure 15-3.

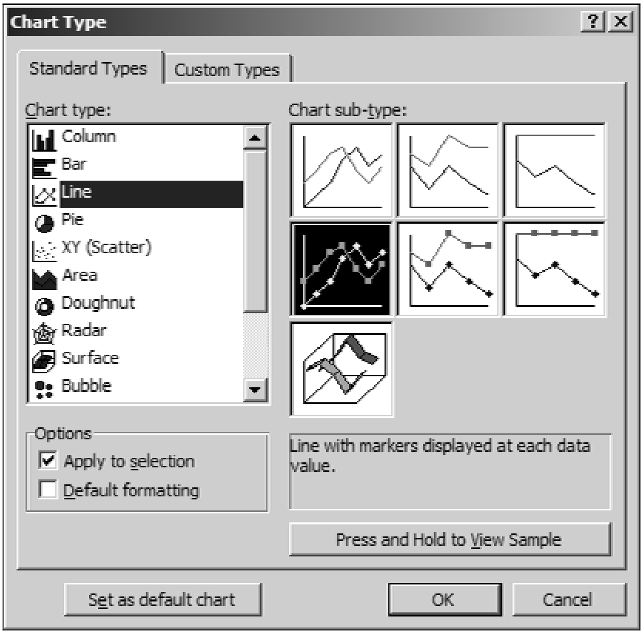


Figure 15-3 Selecting the New Type for the Selected Series

When you click OK, the Forecast series will display as a line, while the Sales series remains as the original column, as shown in Figure 15-4. (We've also modified the format of the Forecast line to make it stand out in the book.)

That's just about all there is to it. Start with a simple column chart with multiple series, select each series in turn, use the *Chart > Chart Type* menu to change its type and then apply the required formatting. The possible combinations are limited only by our imagination and the legibility of the final chart!

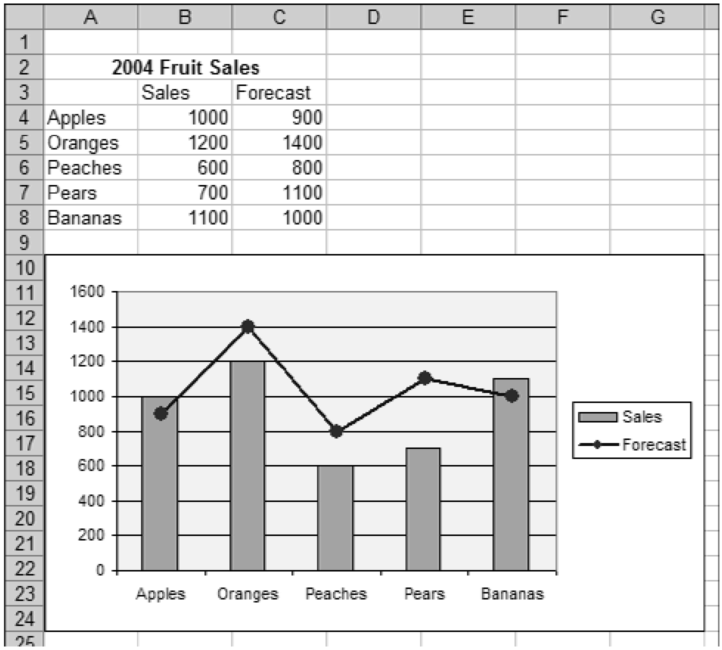


Figure 15-4 The Resulting Combination Column/Line Chart

Using Multiple Axes

When we create one of the standard 2D charts, the plot area can have two sets of axes. The primary axes are usually displayed on the bottom and left, whereas the secondary axes are usually displayed on the top and right. If we have more than one series on the chart, we can choose which set of axes to use for each series by double-clicking the series and making our choice on the Axis tab of the Format Data Series dialog. When instructed to place a series on the secondary axis, Excel usually only displays a secondary Y axis on the chart. This can be changed using the *Chart > Chart Options* menu command, clicking the Axes tab and choosing whatever combination of primary and secondary axes are desired. When two series are plotted on different axes, the axes are scaled independently. Care must be taken to ensure that it is obvious to the viewer which series is plotted on which axis, by adding relevant axis labels and matching them to the series labels, as shown in Figure 15-5.

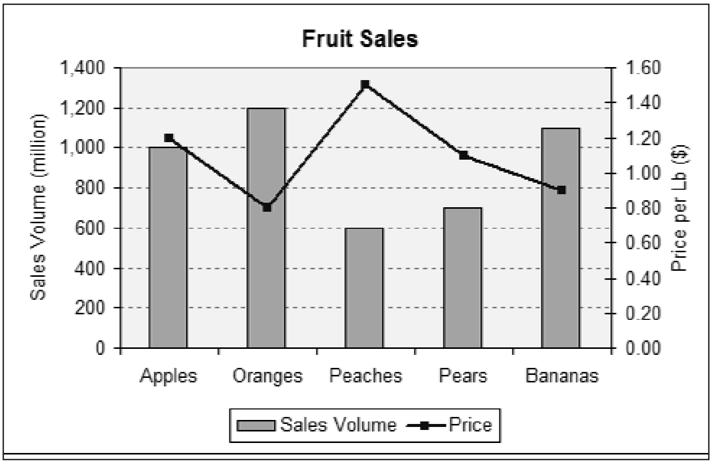


Figure 15-5 Using Labels and Axis Titles to Clearly Identify Which Series Applies to Which Axis

Using Defined Names to Link Charts to Data

A key point to understand is that our charts do not have to refer directly to the cells containing their data. The source data for a chart series is provided by the =SERIES() function, which can be seen in the formula bar when a series is selected. The SERIES() function has the following format:

```
=SERIES(Name, XValues, YValues, PlotOrder)
```

Each of the four parameters can be a constant or array of constants, a direct range reference or a reference to a defined name. All the lines in Listing 15-1 are examples of valid functions.

Listing 15-1 Examples of Valid SERIES() Functions

```
=SERIES(Sheet1!$B$1,Sheet1$A$2:$A$20,Sheet1!$B2:$B20,1)
=SERIES("Sales",Sheet1$A$2:$A$20,Sheet1!$B2:$B20,1)
=SERIES("Horizontal Line",{0,1},{123,123},1)
=SERIES("Book Names",Book1.xls!chtXName,Book1.xls!chtYName,1)
=SERIES("Sheet Names",Sheet1!chtXName,Sheet1!chtYName,1)
```

The last two versions of the SERIES() formula use workbook-level and sheet-level defined names respectively instead of direct cell references. This indirection enables us to use the defined names' definitions to modify the ranges or arrays passed to the chart, as shown in the following examples.

Setting Up the Defined Name Links

When you use a defined name in a SERIES formula, for best results you should begin with a name that references a worksheet range directly. After you have this working correctly, you can modify the name to perform more complex operations. Sometimes, if the formula for the defined name is particularly complex, or if we make an error in its definition, the charting module will refuse to accept the name in the SERIES() function. By starting with a very simple definition for the names, we are able to add them to the SERIES() function without problem.

Figure 15-6 shows a simple line chart, with the series selected and the SERIES() function displayed in the formula bar.

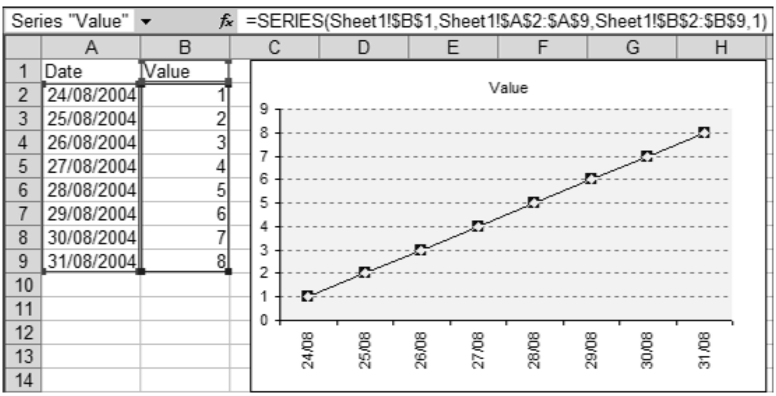


Figure 15-6 A Simple Line Chart

To change the chart to use defined names, we first create two defined names, for the Date and Value ranges. Select *Insert > Name > Define* from the menu and create the following two names:

Name: `Sheet1!chtDates`
Refers to: `=Sheet1!$A$2:$A$9`

Name: `Sheet1!chtValues`
Refers to: `=Sheet1!$B$2:$B$9`

Now select the chart series and edit the SERIES() formula to read as follows:

```
=SERIES("Value",Sheet1!chtDates,Sheet1!chtValues,1)
```

That's it! The chart series is now linked to the defined names and the defined names refer to the source data ranges. Obviously, if we had more series in our chart, we would have to create extra names for the values for each additional series. Now that we've set up the linkage, we can modify the Refers To: formulas for the names (their ***definitions***) to create some interesting and time-saving effects.

Auto-Expanding Charts

One of the most frequently asked questions in the microsoft.public.excel.charting newsgroup is how to get a chart to automatically include new data as it's typed in. In Excel 2003, if we create a List from the data range and set either the chart or the defined names to refer to an entire column of the List, the reference will automatically be adjusted to include any new data. In previous versions, or if we prefer not to convert the range to a List in Excel 2003, we can use defined names to do the automatic updating.

The trick is to use a combination of the OFFSET() and COUNTA() functions in the definition of the name used for the X values, then define the name used for the Y values as an offset from the X values range. Select a cell in the worksheet, then choose *Insert > Name > Define*. Change the definition of the chtDates range to be the following by selecting the existing chtDates entry, typing the new definition and clicking the Add button:

Name: `Sheet1!chtDates`
Refers to: `=OFFSET(Sheet1!$A$2,0,0,COUNTA`
`(Sheet1!$A:$A)-1,1)`

The OFFSET() function has the following parameters:

```
=OFFSET(SourceRange, RowsToMoveDown, ColumnsToMoveAcross,
              NumberOfRowsToInclude, NumberOfColumnsToInclude)
```

The COUNTA() function returns the number of non-blank cells in the range, which in our case includes the header row. We therefore subtract one to get the number of data items. Putting the two together gives us a reference that starts in `A2`, moves down zero rows and across zero columns (so remains in `A2`), has a number of rows equal to the count of our data items and is one column wide. While in the Define Name dialog with the chtDates name selected, if we tab into the Refers to: box, Excel will highlight the resulting range with its "dancing ants," as shown in Figure 15-7.

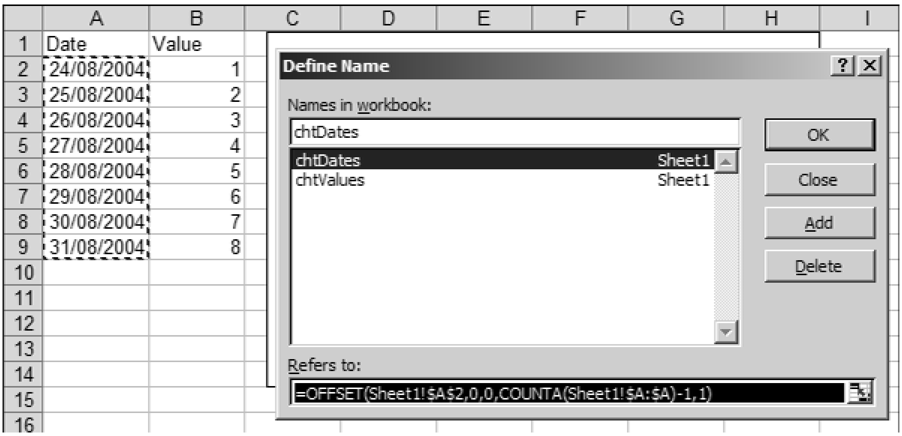


Figure 15-7 Excel's Dancing Ants Showing the Range Referred to by the Defined Name

While we're in the Define Name dialog, we need to modify the definition of the chtValues name. The easiest way to do that is to again use the OFFSET() function, but this time to start at the range referred to by the chtDates name and move one column across, keeping the same height and width:

Name: `Sheet1!chtValues`
Refers to: `=OFFSET(Sheet1!chtDates,0,1)`

After clicking OK to apply those changes and return to the worksheet, the chart should be showing exactly the same as before—the new definitions resolve to the same ranges we started off with. The difference now is that if we type a new data point in row 10, it will automatically appear on the chart (assuming calculation is set to Automatic)!

To recap, it works because the COUNTA() function contained within the definition of the chtDates range returns the number of items in column A, which now includes the new entry. That feeds into the OFFSET() function, making it include the new entry in its resulting reference (now `A2:A10`). The chtValues range is updated to refer to one column across from the expanded chtDates range, so becomes `B2:B10` and both those names feed into the chart series =SERIES() function, making the chart redraw to include the new data. The functions used in the defined name assume that the source data is contiguous, starting in cell A2. Blank cells will result in an incorrectly calculated range. More precise formulas are outside the scope of this book, but can easily be found by searching the Google newsgroup archives.

It is fundamental to the rest of this section that you fully understand the mechanism we're using. If anything is unclear, take some time to go through the example, perhaps trying to create an auto-expanding chart with two or three data series.

Scrolling and Zooming a Time Series

In the auto-expanding chart, we were only updating one of the OFFSET() function's parameters. If we modify both the row offset and number of rows, we can provide a simple, codeless mechanism for our users to scroll and zoom through a time series. In the worksheet shown in Figure 15-8, we've added two scrollbars from the Forms toolbar below the chart, set their Min and Max values to correspond to the number of data points and linked their values to the cells in column D, using two defined names ZoomVal and ScrollVal to refer to cells `D24` and `D25` respectively.

In the definition for the chtDates name for this example, the ScrollVal figure is used for the row offset and the ZoomVal figure provides the number of data points to include in the range:

Name: `Sheet1!chtDates`
Refers to: `=OFFSET(Sheet1!$A$1,Sheet1!ScrollVal,0,`
`Sheet1!ZoomVal,1)`

The chtValues definition is the same as before, `=OFFSET(chtDates,0,1)`.

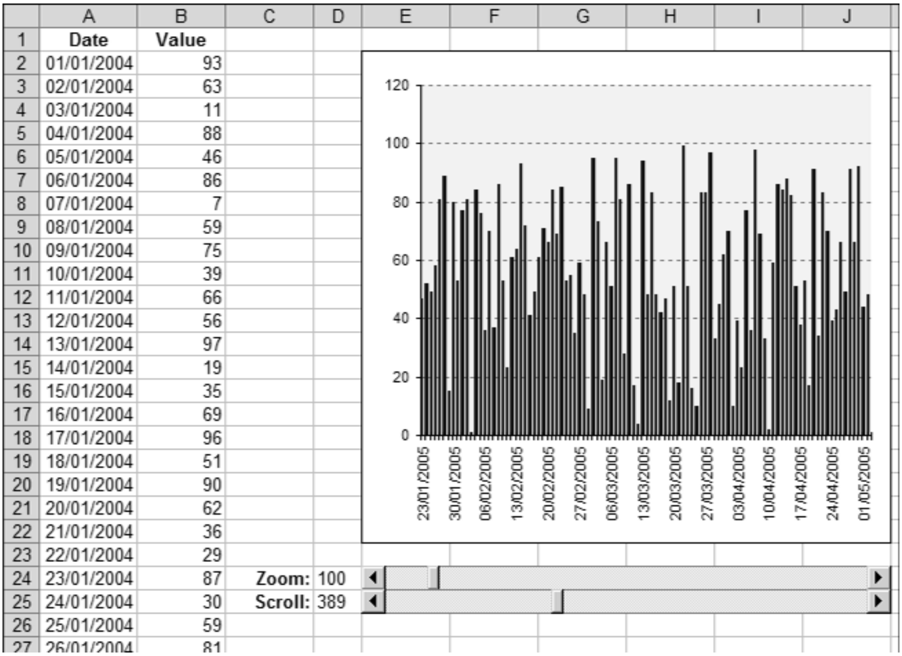


Figure 15-8 Allowing the User to Zoom and Scroll Through Time-Series Data

Transforming Coordinate Systems

In the previous two examples, we've used the OFFSET() function in the defined name to change the range of values drawn on the chart, but keeping the actual data intact. We can also use defined names to modify the data itself prior to plotting it, such as transforming between polar and x, y coordinate systems. In polar coordinates, a point's location is defined by its angle and distance from the origin, rather than the distance-along and distance-up of the standard XY chart. Excel does not have a built-in chart type that will plot data in polar coordinates, but we can use defined names to convert the (angle, length) polar coordinate to (x, y), which can then be drawn on a standard XY chart. We're going to show you how to create the chart shown in Figure 15-9 from the data shown beside it by using defined names. In this example, the length figures are calculated from the angle using the formula `a*sin(a)`.

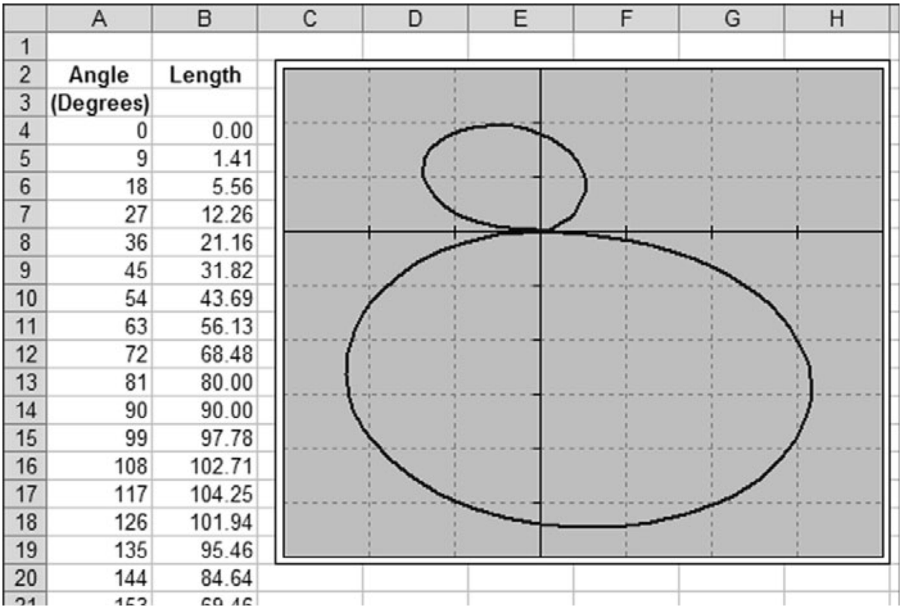


Figure 15-9 Plotting Polar Coordinates on an XY Scatter Chart

To demonstrate how the various uses of defined names can be combined, we'll implement two levels of indirection. The first level will use the technique from the *Auto-Expanding Charts* section above to automatically handle changing data sets, while a second level will perform the coordinate transformation.

The names to handle the automatic updates are defined as follows:

Name: `Sheet1!datAngle`
Refers to: `=OFFSET(Sheet1!$A$3,1,0, COUNTA(Sheet1!$A$3:$A$5000)-1,1)`

Name: `Sheet1!datLength`
Refers to: `=OFFSET(Sheet1!datAngle,0,1)`

The observant reader might have noticed that we're using a slight different version of the OFFSET() function in the definition for the datAngle name. The version shown here is slightly more robust, as it counts within a specific range of 5,000 cells, starting with the data header cell. You may have seen a variation on this technique in which the entire column address was used in the COUNTA function. By limiting the range in the way we do here, it doesn't matter whether the user changes the contents of the cells above the data range, such as adding extra titles to the sheet.

With the datAngle and datLength names referring to our source data, we can define two more names to convert from the polar to x, y coordinates:

```
Name:        Sheet1!chtX
Refers to:   =Sheet1!datLength*
             COS(Sheet1!datAngle*PI()/180)

Name:        Sheet1!chtY
Refers to:   =Sheet1!datLength*
             SIN(Sheet1!datAngle*PI()/180)
```

The chart series can then use the chtX and chtY names for the X and Y data:

```
=SERIES("Polar Plot",Sheet1!chtX,Sheet1!chtY,1)
```

Charting a Function

So we've used defined names to change the range of cells to plot and to manipulate the data in that range before we plot it. In *Chapter 14 — Data Manipulation Techniques*, we introduced array formulas and explained how they can be used to perform calculations on arrays of data. We also showed a specific array formula that is often used to generate a number sequence for use in other array formulas. What we didn't mention was that we can also use array formulas in our defined names and refer to them from charts! Figure 15-10 shows a worksheet that uses array formulas in defined names to plot a mathematical function over a range of x values, without needing to read any data from the worksheet.

This worksheet combines a number of Excel tricks to generate the x axis values and use them to calculate the y axis results. We create a defined named to generate the values for the x axis and give it the name x, for reasons explained below:

```
Name:        Sheet1!x
Refers to:   =$C$6+(ROW(OFFSET($A$1,0,0,$C$8,1))-
             1)*($C$7-$C$6)/($C$8-1)
```

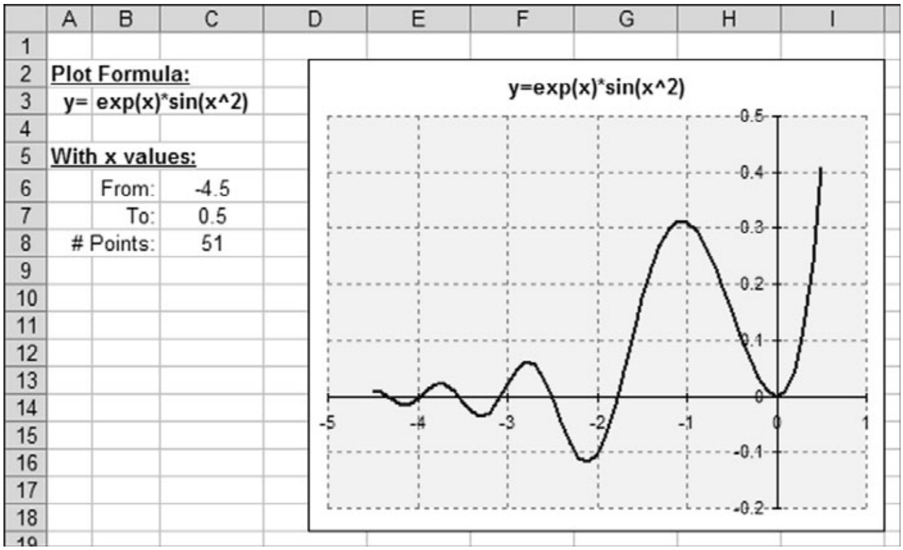


Figure 15-10 Using Array Formulas in Defined Names to Generate and Plot Data

Working through the parts of this array formula:

- `OFFSET($A$1,0,0, $C$8,1)` gives the range A1:A51.
- `ROW(OFFSET($A$1,0,0, $C$8,1))` converts the range to the array {1, 2, 3, …, 50, 51}.
- `(ROW(OFFSET($A$1,0,0, $C$8,1))-1)` subtracts 1 from each item in the array, giving {0, 1, 2, …, 49, 50}.
- `($C$7-$C$6)/($C$8-1)` calculates the x axis increment for each point, giving 0.1 in our example.
- `(ROW(OFFSET($A$1,0,0,$C$8,1))-1)*($C$7-$C$6)/($C$8-1)` multiplies each item in the array by the x axis increment, giving the array {0, 0.1, 0.2, …, 4.9, 5.0}.
- `$C$6+(ROW(OFFSET($A$1,0,0,$C$8,1))-1)*($C$7-$C$6)/($C$8-1)` adds the array to the required x value start point, resulting in the range of x values to use in the chart {–4.5, –4.4, –4.3, … 0.49, 0.50}.

Unfortunately, if we try to include Sheet1!x in the chart SERIES() function, we get an error about an incorrect range reference. To create the chart, we use the workaround described at the start of this section, by creating two names chtX and chtY that point to worksheet cells, use them to create the chart and then change them to their real definitions:

Name: `Sheet1!chtX`
Refers to: `=Sheet1!x`

Name: `Sheet1!chtY`
Refers to: `=EVALUATE(Sheet1!$B$3&"+x*0")`

The definition for chtX is just a workaround for Excel not allowing us to use the `x` name in the chart itself. The definition for chtY needs some explaining! Cell `B3` contains the equation to be plotted, `exp(x)*sin(x^2)`, as text. The EVALUATE function is an XLM macro function, equivalent to the VBA Application.Evaluate method, but which can be called from within a defined name. XLM functions were the programming language for Excel 4, replaced by VBA in Excel 5, but still supported in Excel 2003. The documentation for the XLM functions can be downloaded from the Microsoft Web site, by searching for "macrofun.exe" or "xlmacro.exe." At the time of writing, one version of the file is available from `http://support.microsoft.com/?kbid=128175`.

EVALUATE() evaluates the expression it's given, returning a numeric result. In our case, when the expression is evaluated, Excel replaces the x's in the formula with the array of values produced by our Sheet1!x defined name (which is exactly why we called it x) and returns an array containing the result of the function for each of our x axis values. These arrays are plotted on the chart, to give the line for the equation. The `&"+x*0"` part of the chtY definition works around an error in Excel that sometimes causes trig functions to not evaluate as array formulas, by forcing the entire formula to be evaluated as an array.

Faking It

A chart is a visual artifact, designed to impart information to the viewer in a graphical manner. As such, we should mainly be interested in whether the final chart looks correct and performs its purpose of providing clear information. We should not be too bothered about whether the chart has been constructed according to a notional set of generally approved guidelines. In other words, we often need to cheat by using some of the chart engine's features in "creative and imaginative" ways. This section explains a few ways in which we can get creative with Excel's chart engine, by using some of its features in ways they were probably not designed to be used.

Error Bars

When is a line not a line? When it's an error bar! From a purely visual perspective, an error bar is a horizontal or vertical line emanating from a data point, so if we ever have the need to draw horizontal or vertical lines around our data points, we might consider using error bars for those lines. A great example is the step chart shown in Figure 15-11, where the vertical lines show the change in an item's price during a day and the horizontal lines connect the end price from one day to the start price for the next day.

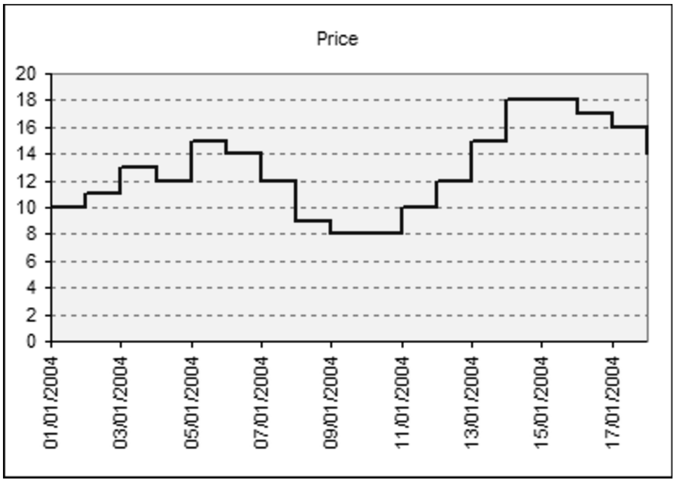


Figure 15-11 A Step Chart

Because Excel doesn't include a built-in Step Chart type, many people believe that Excel can't create them. There are quite a few ways in which it can be done, but the easiest is probably to use an XY chart with both vertical and horizontal error bars. The basic data for the chart consists of a list of dates and end-of-day prices, with a calculated field for the change in price from the end of the previous day. From this basic data, we start with a normal XY chart to plot the price against the date, as shown in Figure 15-12.

Below each data point, we want to display a vertical line equal to the change in price for that day, which we do by specifying a custom minus error value in the Y Error Bars tab of the Format Data Series Dialog, as shown in Figure 15-13.

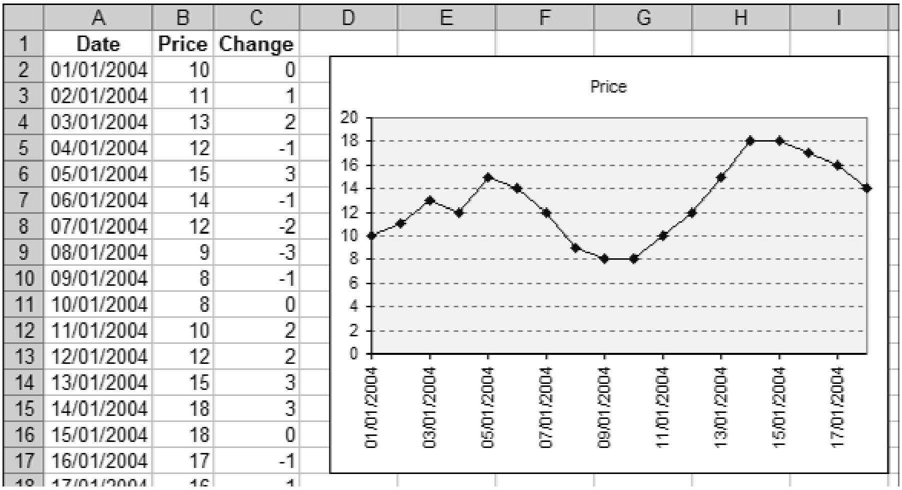


Figure 15-12 Start with a Normal XY (Scatter) Chart of Price vs. Date

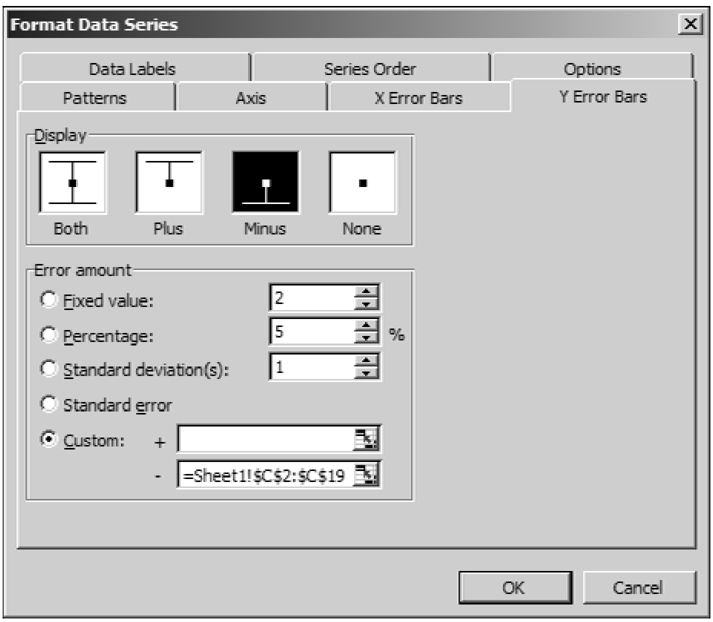


Figure 15-13 Add a Custom Minus Y Error Bar for the Day's Change in Price

The horizontal lines need to join each data point to the bottom of the subsequent point's error bar. That sounds difficult, but because these are daily prices all you need to do is add Plus markers to the X error bars with a fixed value setting of 1. With the error bars configured, you should be seeing a chart something like that shown in Figure 15-14.

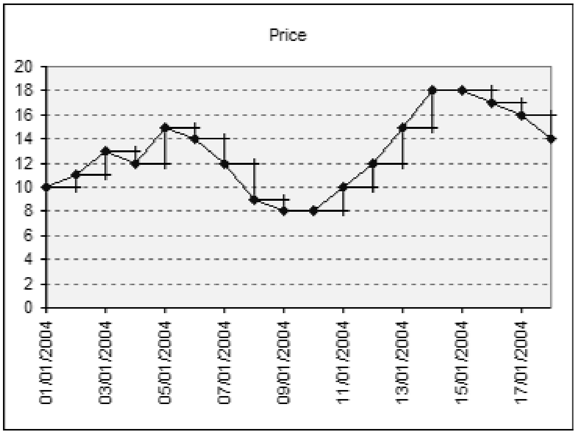


Figure 15-14 The Chart with the Additional Error Bars

All that remains is to double-click the error bar lines and use the Patterns tab to change their color, thickness and marker style, and then double-click the original XY line and format that to have no line and no marker. The result appears to be the step chart from Figure 15-11, even though it's actually only error bars being drawn.

Dummy XY Series

When is an axis not an axis? When it's an XY series with data labels! Excel's value axes are either boringly linear or logarithmic. They do not support breaks in the axis, nor scales that vary along the axis nor many other complex-axis effects. Figure 15-15 shows a chart with a variable Y axis, where the bottom half of the chart plots values from 0 to 100 in steps of 20, but the top half plots 100 to 1,000 in steps of 200:

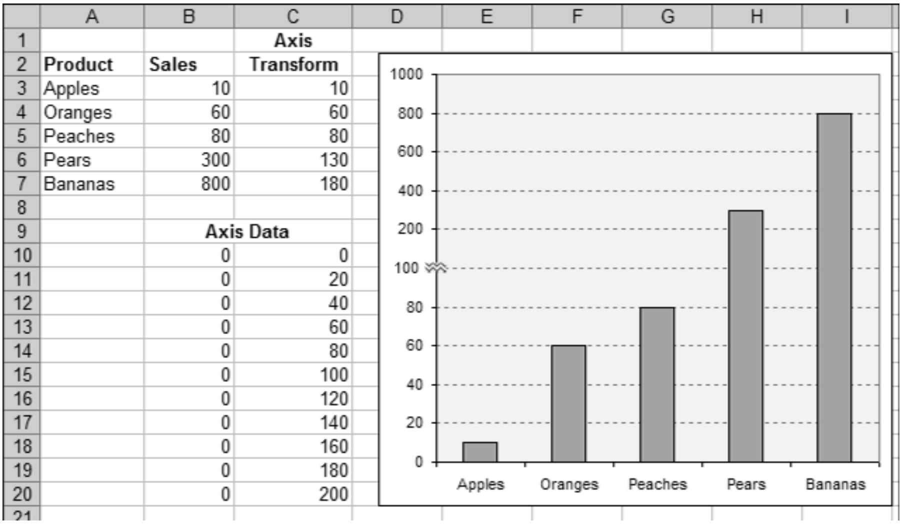


Figure 15-15 Chart with a Complex Axis Scale

In this chart, the real Y axis goes from zero to 200, but we've added a dummy XY series using the data from `B10:C20`, added data labels to the XY series, set them to display to the left of the point and customized their text to that shown in the figure. The result appears to be a complex axis scale that varies up the chart. The final step is to transform the real sales data in `B3:B7` into the correct values for Excel to plot on its linear 0 to 200 scale, which is done using a simple mapping formula in `C3:C7` of `=IF(B3<=100,B3,100+B3/10)`, which is the data that Excel plots.

We can use this technique to implement any axis scale of our choosing, such as including breaks in our axes, plotting using logarithmic, hyperbolic or probability scales or even including multiple dummy XY series to make the chart appear to have many axes (as long as the user can determine which series is plotted against which axis). This effect can be misleading, if it is not clearly shown that a break in the axis scale exists. The chart in Figure 15-15 looks linear along its entire range, but if plotted on a true linear scale, it would resemble a boomerang with a large angle in the middle. An easy way to indicate a break in the axis is to set an individual point's data marker using a custom image, as we have done. Draw the image using Paint or other graphics program, copy it to the clipboard, select the data point and paste the image.

VBA Techniques

So far, we've concentrated on the techniques we can use to get the most out of Excel's charting engine through the user interface. In this section, we examine how we can use VBA to manipulate charts.

Converting Between Chart Coordinate Systems

When using VBA to work with charts, there are (at least) four different coordinate systems that we often need to convert between:

- The chart series data displayed inside the plot area is in the axis coordinates if it's an XY Scatter chart.
- The mouse pointer coordinates given in the MouseMove etc. events are measured in pixels, with the origin in the top-left corner of the ChartObject window.
- The coordinates of any drawing objects added to the chart are in points, with the origin being the top left of the chart area, slightly inside the ChartObject window.
- The coordinates used by the GET.CHART.ITEM XLM function to locate the vertices of chart objects are in points, but with the origin in the bottom-left corner of the chart area. See the *Locating Chart Items* section later for an example of its use.

Furthermore, if the chart is embedded on a worksheet, the worksheet zoom factor affects the mouse pointer coordinates, but not the data nor location of any drawing objects on the chart.

Listing 15-2 shows a MouseMove event for a chart, within which we convert the X, Y mouse coordinates given to the event into both data coordinates (displayed in the status bar) and drawing object coordinates (which we use to move an oval to follow the mouse pointer). Note that this code uses the PointsPerPixel function defined in *Chapter 9 — Understanding and Using Windows API Calls*:

Listing 15-2 Converting from Mouse Coordinates to Data and Drawing Object Coordinates

```
Private Sub mchtChart_MouseMove(ByVal Button As Long, _
    ByVal Shift As Long, ByVal X As Long, ByVal Y As Long)
```

```
Dim dZoom As Double
Dim dXVal As Double
Dim dYVal As Double
Dim dPixelSize As Double

On Error Resume Next

'The active window zoom factor
dZoom = ActiveWindow.Zoom / 100

'The pixel size, in points
dPixelSize = PointsPerPixel

'Mouse coordinates to (XY) Data coordinates
With mchtChart
  dXVal = .Axes(xlCategory).MinimumScale + _
     (.Axes(xlCategory).MaximumScale - _
      .Axes(xlCategory).MinimumScale) * _
     (X * dPixelSize / dZoom - _
      (.PlotArea.InsideLeft + .ChartArea.Left)) / _
     .PlotArea.InsideWidth

  dYVal = .Axes(xlValue).MinimumScale + _
     (.Axes(xlValue).MaximumScale - _
      .Axes(xlValue).MinimumScale) * _
     (1 - (Y * dPixelSize / dZoom - _
           (.PlotArea.InsideTop + .ChartArea.Top)) / _
     .PlotArea.InsideHeight)
End With

Application.StatusBar = "(" & Application.Round(dXVal, 2) _
     & ", " & Application.Round(dYVal, 2) & ")"

'Mouse coordinates to Drawing Object Points

'We'll only move the oval if the Shift key is pressed
If Shift = 1 Then
  With mchtChart
    dXVal = (X * dPixelSize / dZoom - .ChartArea.Left)
    dYVal = (Y * dPixelSize / dZoom - .ChartArea.Top)

    With .Shapes("ovlPointer")
      .Left = dXVal - .Width / 2
      .Top = dYVal - .Height / 2
```

```
      End With
    End With
  End If

End Sub
```

Locating Chart Items

Sometimes, however hard we try, the only way to get a chart looking exactly how we want it is to add drawing objects to it, such as rectangles, lines, arrows and so on. As soon as we do that, we hit the problem of trying to identify where in the drawing object coordinate space an item on the chart is located, such as the top middle of a specific column in a column chart.

That level of positional information cannot be obtained through the Excel object model, but can be obtained by calling on the long-disused XLM function GET.CHART.ITEM. This function has the following parameters:

```
GET.CHART.ITEM(x_y_index, point_index, item_text)
```

Where:

- `x_y_index` is 1 to return the x position and 2 to return the y position.
- `point_index` depends on the item we're looking at, but is a number from 1 to 8 to identify a specific vertex within the item. For example, 2 is the upper middle of any rectangular item, such as a column in a column chart.
- `item_text` identifies the item we're interested in, such as "Plot" for the plot area, or "S2P4" for the fourth data point in the second series in the chart.

The full list of available parameters can be found in the XLM Macros help file available for download from the Microsoft Web site at `http://support.microsoft.com/?kbid=128175`. The only caveat with using GET.CHART.ITEM is that the chart must be active for it to work. The code in Listing 15-3 moves an arrow on a chart to be from the top-left corner of the inside of the plot area (using normal VBA positioning) to the top middle of the third column of a column chart, resulting in the chart shown in Figure 15-16.

Listing 15-3 Using GET.CHART.ITEM to Locate a Chart Item's Vertices

```
Private Sub cmdMoveArrow_Click()

  Dim rngActive As Range
  Dim dXVal As Double
  Dim dYVal As Double
  Dim chtChart As Chart

  Set rngActive = ActiveCell

  'We have to activate the chart to use GET.CHART.ITEM
  Me.ChartObjects(1).Activate

  'Find the XY position of the middle top of the third column
  'in the data series,
  'returned in XLM coordinates
  dXVal = ExecuteExcel4Macro("GET.CHART.ITEM(1,2,""S1P3"")")
  dYVal = ExecuteExcel4Macro("GET.CHART.ITEM(2,2,""S1P3"")")

  'Get the Chart
  Set chtChart = Me.ChartObjects(1).Chart
  With chtChart

    'Convert the XLM coordinates to Drawing Object coordinates
    'The x values are the same, but the Y values need to be
    'flipped
    dYVal = .ChartArea.Height - dYVal

    'Move and size the Arrow
    .Shapes("linArrow").Left = .PlotArea.InsideLeft
    .Shapes("linArrow").Top = .PlotArea.InsideTop
    .Shapes("linArrow").Width = dXVal - .Shapes("linArrow").Left
    .Shapes("linArrow").Height = dYVal - .Shapes("linArrow").Top
  End With

  rngActive.Activate

End Sub
```

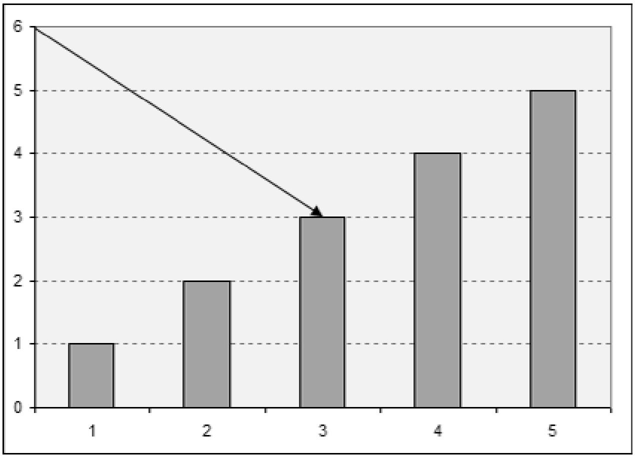


Figure 15-16 Moving an Arrow to Point to the Top Middle of a Column

Calculating Reasonable Axis Scales

Often when we're controlling charts through VBA, we need to set our own values for the axis scales. The code in Listing 15-4 calculates tidy Minimum, Maximum and MajorUnit values. It is a different algorithm than the one Excel uses to determine chart axis scales, but is one that we have found to give pleasant-looking results.

Listing 15-4 Function to Calculate Reasonable Chart Axes Scales

```
Public Type CHART_SCALE
  dMin As Double
  dMax As Double
  dScale As Double
End Type

Public Function ChartScale(ByVal dMin As Double, _
        ByVal dMax As Double) As CHART_SCALE

    Dim dPower As Double, dScale As Double

    'Check if the max and min are the same
    If dMax = dMin Then
        dScale = dMax
        dMax = dMax * 1.01
```

```
        dMin = dMin * 0.99
    End If

    'Check if dMax is bigger than dMin - swap them if not
    If dMax < dMin Then
        dScale = dMax
        dMax = dMin
        dMin = dScale
    End If

    'Make dMax a little bigger and dMin a little smaller
    If dMax > 0 Then
        dMax = dMax + (dMax - dMin) * 0.01
    Else
        dMax = dMax - (dMax - dMin) * 0.01
    End If
    If dMin > 0 Then
        dMin = dMin - (dMax - dMin) * 0.01
    Else
        dMin = dMin + (dMax - dMin) * 0.01
    End If

    'What if they are both 0?
    If (dMax = 0) And (dMin = 0) Then dMax = 1

    'This bit rounds the maximum and minimum values to
    'reasonable values to chart.
    'Find the range of values covered
    dPower = Log(dMax - dMin) / Log(10)
    dScale = 10 ^ (dPower - Int(dPower))

    'Find the scaling factor
    Select Case dScale
    Case 0 To 2.5
        dScale = 0.2
    Case 2.5 To 5
        dScale = 0.5
    Case 5 To 7.5
        dScale = 1
    Case Else
        dScale = 2
    End Select
```

```
    'Calculate the scaling factor (major unit)
    dScale = dScale * 10 ^ Int(dPower)

    'Round the axis values to the nearest scaling factor
    ChartScale.dMin = dScale * Int(dMin / dScale)
    ChartScale.dMax = dScale * (Int(dMax / dScale) + 1)
    ChartScale.dScale = dScale

End Function
```

Conclusion

Although Excel's charting engine has a relatively poor reputation among users, most of that is due to a lack of knowledge about how to exploit the engine, rather than a lack of features. Yes, we would like to see significant improvements in the quality of the graphics, proper support for true 3D contour and XYZ scatter plots and a general overhaul of the user interface to make the advanced techniques shown in this chapter much more discoverable for the average user.

However, after we've spent the time to explore the charting engine and fully understand the techniques introduced here, we realize that the limits of Excel's charting capabilities are to be found in our imagination and creativity, rather than with Excel.

Index

P