
C H A P T E R

1

DLLs and APIs

*Moving from DOS to
Windows*

*Dynamic Link Libraries
(DLLs)*

*Application
Programmer's Interface
(API)*

*The Different Flavors of
Windows*

*The Different Flavors of
Visual Basic*



IN 1991, MICROSOFT RELEASED VERSION 1.0 OF A NEW LANGUAGE CALLED Visual Basic. By doing so, they have literally revolutionized Windows programming, and in many ways advanced the art of programming in general. By some estimates over 1 million copies of Visual Basic have been sold, making it one of the most popular programming languages of all time. Perhaps the most important reason for this is that Visual Basic is the first programming tool that truly allows people to write Windows applications without being a Windows expert.

Moving from DOS to Windows

Visual Basic is a comprehensive programming language. It is certainly adequate for a great many applications. There is a good chance, however, that sooner or later you will run into a requirement that Visual Basic does not support directly. For example, sound support, I/O port access, and many bitmap operations are not part of the Visual Basic language.

But this is not really a flaw in Visual Basic. No language can contain every command and function that might be needed by every programmer. In the DOS world, this problem is solved by creating and using libraries of functions that extend the capabilities of the language. These functions can be linked into your programs as needed.

In the Windows world, things are a bit more complex. For one thing, the operating environment is considerably more complicated than DOS—the simple act of displaying a line on the screen involves the use of objects known as Windows handles, display contexts, and pens. The process of linking in external functions is also different—Windows applications often use a technique known as dynamic linking instead of the static linking common in DOS (linking will be described in detail later in this chapter).

As long as you are using only Visual Basic functions and commands, this complexity remains mostly in the background. As soon as you are ready to go beyond the built-in features of the language, however, you are sure to face some of these issues.

Don't panic.

Like any serious programming language, Visual Basic is highly extensible. While a great deal of press has been devoted to the third-party add-ons and libraries available for VB, the fact is that the greatest add-on library of them all is included automatically with every copy of the language. That is Windows itself.

There has been a great deal of talk about how complex Windows programming is—and it is, for people writing entire programs in C++, C, or Pascal. As a Visual Basic programmer, though, you already have a good general understanding of how Windows works (though you may not realize it). That is because Visual Basic is very closely tied to Windows. VB events correspond closely to Windows messages. VB properties often correspond closely to



Windows styles and properties. In a sense, Visual Basic is a programmable shell for Windows.

The first part of this book covers the relationship between Windows and Visual Basic. This chapter focuses on the transition from DOS to Windows in general, the 32-bit versions of Windows in particular, and discusses the subject of extensibility. Chapter 2 looks inside Windows and shows how it corresponds to Visual Basic. Chapter 3 is a thorough review of everything you need to know to access Windows functions and DLL functions from Visual Basic. Chapter 4 examines specific techniques for writing programs that will work on the different types of Windows operating systems and goes into further detail on how to translate typical C language documentation into Visual Basic.

To appreciate the differences between DOS and Windows programming, it is a good idea to take a moment and review the three big differences between DOS and Windows itself.

First, Windows is multitasking—it's able to run more than one program at the same time. Multitasking poses an interesting set of problems. Which program gets access to the screen? How is memory shared? If more than one program is using the screen, which program gets keyboard input? Which one gets mouse input? What if two programs want to use the same serial port? Almost every difference between DOS and Windows programming relates in some way to multitasking.

Second, Windows is event driven. Many DOS programs are written as loops that perform I/O operations as needed. The programmer decides when it is time to check for keyboard or mouse input. Windows programs can receive external events at almost any time and should not use continuous loops, which on some operating systems can tie up the system and prevent other applications from running.

Third, Windows is device independent. DOS programs frequently need to take into account every possible printer or graphics device type. It is not unusual for a DOS program to ship with dozens of printer drivers, and separate graphics drivers for EGA, VGA, monochrome, and Super VGA graphics mode—and sometimes even for different brands of VGA cards! Windows provides a graphics device interface (called GDI) that makes it possible to support virtually any graphics device—Windows takes care of providing the individual drivers. GDI translates graphics commands so that they appear identical (or at least as similar as possible) on all devices.

Many of the programming issues that will be discussed here derive from these differences. The development of dynamic linking is one such issue.

Dynamic Link Libraries (DLLs)

Linking refers to the process in which external functions are incorporated into an application. There are two types of linking: *static* and *dynamic*. Static



linking takes place during creation of the program. Dynamic linking takes place when the program is running.

Static Linking

Programming languages are typically extended in two ways. First, most languages provide access to the underlying operating system. In DOS, this is accomplished with interrupt 21 calls to DOS. Second, most languages allow you to create libraries of functions that can be merged into your program. These functions then appear to the programmer as if they were built into the language.

Program modules containing functions are precompiled into object (.OBJ) files. These object files are often grouped into library (.LIB) files using a program called a librarian (such as LIB.EXE).

When it is time to create a final executable version of an application, a program known as a linker scans your application's object files for references to functions that are not defined. It then searches through any specified library files for the missing functions. The linker extracts any program modules containing the required functions and copies them into the new executable file, "linking" them to your program. This process, shown in Figure 1.1, is known as *static linking* because all of the addressing information needed by your program to access the library functions is fixed when the executable file is created, and remains unchanged (static) when the program is running.

Linkers traditionally include entire modules when linking in an executable file, thus functions F2 and F5 in Figure 1.1 are included in the application even though they are not called by module YourApp. Newer compilers and linkers, especially those used by the C++ language, allow inclusion of functions on an individual basis.

Static linking has one minor drawback. Consider the example in Figure 1.2. Imagine you have a function called ShowMessage that displays a message on the screen and requests a user prompt. Say this function is 20K in size and you use it in five different programs.

As Figure 1.2 shows, 80K of disk space is essentially wasted in copies of the same function. If a typical application uses library functions totaling 100K, it is clear that this can quickly add up. Since Windows applications tend to make heavy use of common functionality and function libraries, it is not unusual for applications to use many hundreds of kilobytes of functions and grow into the megabyte range.

Still, it's only disk space. It's almost impossible today to find a disk with a capacity under 500 megabytes, so this really isn't much of a problem.

That is, until you start writing Windows applications. Windows is multitasking, meaning that it is quite possible for all five programs to be running at the same time. Now that 80K of wasted space is tying up scarce memory as well.



Figure 1.1
Static linking

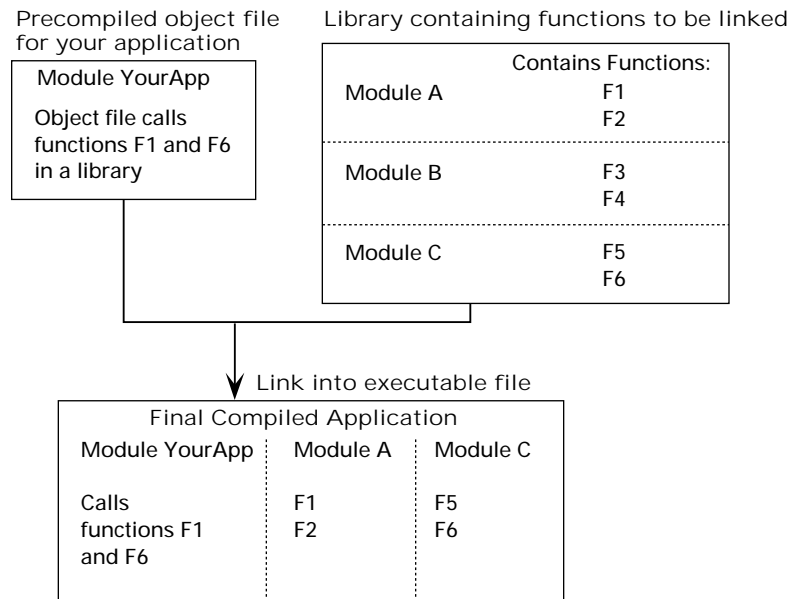
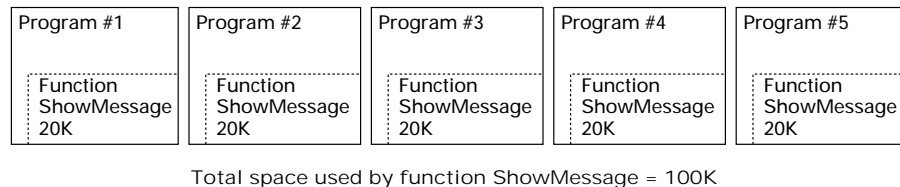


Figure 1.2
Disk space used under static linking



Another problem that arises with Windows is the issue of accessing the underlying system. With DOS applications, a few dozen operating system commands can be accessed through a simple interrupt scheme. Windows has hundreds of commands. Microsoft addressed these problems by implementing a technique called *dynamic linking* which is described in the next section.

Dynamic Linking

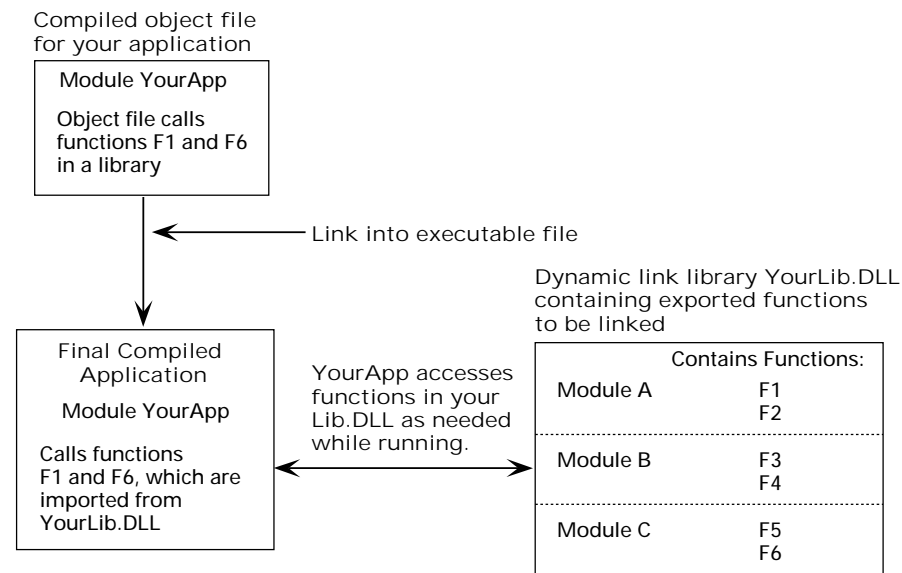
With dynamic linking, program modules containing functions are also pre-compiled into object (.OBJ) files. Instead of grouping them into library files, they are linked into a special form of Windows executable file known as a *dynamic link library* (DLL). When a DLL is created, the programmer specifies

which of the functions included should be accessible from other running applications. This is known as *exporting* the function.

When you create a Windows-executable file, the linker scans your program's object files and makes a list of those functions that are not present and the DLL in which they can be found. The process of specifying where each function can be found is known as *importing* the function.

The dynamic linking process is shown in Figure 1.3.

Figure 1.3
Dynamic linking

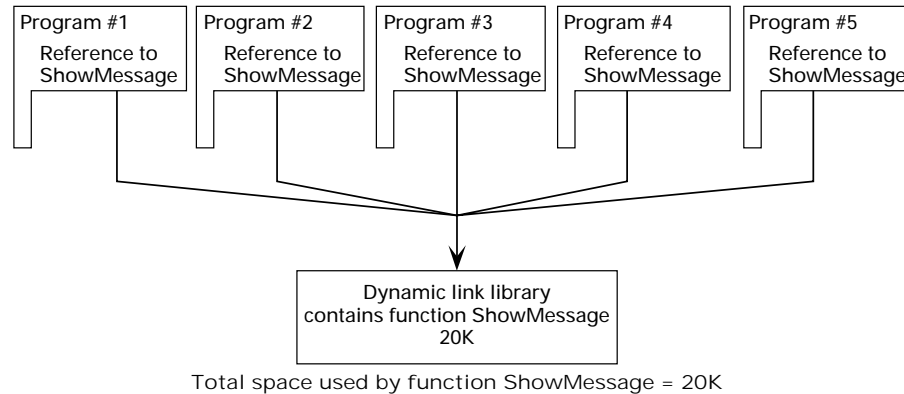


When your application runs, any time it needs a function that is not in the executable file, Windows loads the dynamic link library so that all of its functions become accessible to your application. At that time, the address of each function is resolved and dynamically linked into your application—hence the term *dynamic* linking. Figure 1.4 updates the example shown in Figure 1.2, illustrating how memory is saved by having all applications share the same dynamic link library.

Dynamic link libraries typically have the extension .DLL, but this is not a requirement. Visual Basic custom controls and OLE controls are also DLLs (though they have some special features) and use the extension .VBX or .OCX. Windows device drivers are DLLs and typically have the extension .DRV. Some Windows system DLLs use the standard executable extension .EXE, especially in the 16-bit environments.



Figure 1.4
Disk space used
under dynamic
linking



Visual Basic and DLLs

Visual Basic provides two mechanisms for programmers to specify import information for a program. The `Declare` statement tells Windows which DLL contains a desired function and what parameters it expects. You can also add a reference to a Windows API type library which contains the same information. The tradeoffs involved with these two choices is discussed further in Chapter 3.

Visual Basic does not allow you to directly export functions that can be directly called from other applications. DLLs created using Visual Basic expose OLE automation interfaces—not individual functions.

For those situations where you just have to create your own dynamic link libraries that export functions, you will either need a third party tool that adds this capability to Visual Basic (such as Desaware's SpyWorks) or a more traditional development system and language. The most common tool for DLL development at this time is Visual C++ from Microsoft. This is a rapidly changing field, however, and new tools appear frequently, so you may want to do additional research if you choose to take this approach.

Application Programmer's Interface (API)

API is one of those acronyms that seems to be used primarily to intimidate people. An API is simply a set of functions available to an application programmer. The DOS interrupt functions can technically be considered the DOS API. If you write database programs in dBase, the dBase functions you use can be considered the dBase API.



The term is most often used to describe a set of functions that are part of one application but are being accessed by another application. When a Visual Basic program uses OLE Automation to execute an Excel spreadsheet function, you can say that it is accessing the Excel API.

So, the Windows API refers to the set of functions that are part of Windows and are accessible to any Windows application. It is difficult to overstate the significance of this concept. Consider the following example.

Bring up the Windows Program Manager (NT) or Windows Explorer (Windows 95) and invoke the About command from the Help menu. A dialog box will come up showing information about the system, including the amount of physical memory that is free or available.

Obviously there is a method within Windows for determining these values. As it turns out, that function is called `GlobalMemoryStatus` and is exported by Windows. Thus it is part of the API and is available to any Windows application.

Try the following trivial Visual Basic program:

```
' Create a new project
' In the global module place the following statements:
Type MEMORYSTATUS
    dwLength As Long          ' 32
    dwMemoryLoad As Long      ' percent of memory in use
    dwTotalPhys As Long       ' bytes of physical memory
    dwAvailPhys As Long       ' free physical memory bytes
    dwTotalPageFile As Long   ' bytes of paging file
    dwAvailPageFile As Long   ' free bytes of paging file
    dwTotalVirtual As Long    ' user bytes of address space
    dwAvailVirtual As Long    ' free user bytes
End Type
Declare Sub GlobalMemoryStatus Lib "kernel32" (lpmmemstat As MEMORYSTATUS)
' In the form_Click event for form1 place the following lines:
Dim ms As MEMORYSTATUS
ms.dwLength = Len(ms)
GlobalMemoryStatus ms
Print "Total physical memory: "; ms.dwTotalPhys
Print "Available physical memory: "; ms.dwAvailPhys
```

Now run the program. When you click anywhere on the form, it will display the physical memory statistics for the system. Almost every aspect of the Windows environment and the API functions associated with it are accessible from Visual Basic.

The Windows API and Visual Basic

When you consider that the Windows API has literally thousands of functions, it becomes apparent that there is a great deal of capability available.



This may seem overwhelming—after all, learning thousands of functions can be somewhat time-consuming, almost as time-consuming as trying to figure out which ones to use in any given case. Fortunately, there are a number of factors that help bring this down to size. From the point of view of the Visual Basic programmer, the Windows API functions can be divided into four categories:

1. *API functions that correspond to Visual Basic features.* Many API functions are already built into Visual Basic, so there is no need to access them via the Declare statement. These functions will be covered briefly, as they are of little use to the VB programmer.
2. *API functions that cannot be used from Visual Basic.* There are a number of API functions that, for reasons that will become clear later, simply cannot be accessed from Visual Basic. In most cases, this is because they require parameters that are incompatible with VB. In other cases, they implement operations that are necessary for an independent Windows program that VB handles behind the scenes. Most of these will also be mentioned briefly in Appendix E, but will not be covered in any detail.
3. *API functions that are useful for Visual Basic programmers.* Most of these functions will be covered in detail with examples.
4. *API functions that cannot be called directly from Visual Basic, but can be accessed through a simple interface that performs certain parameter conversions or allows access to information that cannot be obtained directly.* The CD-ROM provided with this book includes the dynamic link library APIGID32.DLL (described in Appendix A), which provides support for these functions; thus they will also be covered in detail with examples. As an added bonus, the APIGID32.DLL library also contains a number of extra functions (such as port I/O support) that help fill in some of the few remaining gaps.

The Different Flavors of Windows

Between 1991 and 1995, Visual Basic evolved from version 1.0 to 3.0, and Windows evolved from version 3.0 to 3.11. This period saw a dramatic improvement in features and performance in both Windows and Visual Basic, yet the Windows API itself remained fundamentally the same. True, the number of functions increased as Microsoft added new dynamic link libraries to Windows that extended the capability of the operating system, but the core API remained constant, and you could write your Visual Basic programs to take advantage of the capability provided by the Windows API knowing that your program would run on any Windows system.



In 1995, with the appearance of Visual Basic 4.0, this situation changed. The root of this change lies in this simple, but crucial fact: The Windows API is *not* the same thing as Windows. Or to put it differently: an API is not an operating system. Up until Visual Basic 4.0, Windows programmers did not have to worry about this difference. Windows was essentially a single operating system—from the programmer’s point of view the differences between Windows 3.0 and Windows 3.11 were minor. That single operating system had a single API.

Visual Basic 4.0 provides support for three different operating systems: Windows 3.x, Windows NT, and Windows 95, and is likely to support additional operating systems soon. As the underlying operating systems have evolved, the Windows API has evolved as well. Instead of a single 16-bit API, there is now a 16-bit Windows API called the “Win16” API, and several variations of a 32-bit API as well.

These changes mean that it is no longer possible to write code for a single operating system and API. As long as you are writing pure Visual Basic code, VB hides most of the differences between environments from you. However, in order to preserve compatibility with older operating systems, it also prevents you from taking advantage of many of the new operating system and API features. Once you start accessing the Windows API, it becomes necessary to consider which operating system and API you intend your program to support. In some cases, you may need to write functions that perform differently depending on the operating environment. Fortunately, Visual Basic 4.0 contains new features to make this possible. Visual Basic 5.0, on the other hand, only supports 32-bit operating systems and thus only accesses the Win32 API.

Table 1.1 lists the versions of the Windows API that work with each of the Microsoft operating systems that are available today.

Table 1.1 The Windows APIs

| API | Description |
|-------|--|
| Win16 | The original Windows 16-bit API. This is the native API for Windows and Windows for Workgroups 3x. It is also supported by Windows 95 and Windows NT in 16-bit mode. You can access this API from the 16-bit editions of Visual Basic (including VB 3.0 and 4.0). Documentation on using this API from Visual Basic can be found in <i>The Visual Basic Programmer’s Guide to the Windows API</i> —the predecessor to this book. |
| Win32 | The full 32-bit API. This is the native API for Windows NT. You can access this API from any 32-bit edition of Visual Basic or Visual Basic for Applications. |



Table 1.1 The Windows APIs (Continued)

| API | Description |
|----------------------|---|
| Win32c Windows 95 | This is a subset of Win32 that is supported by Windows 95 (the “c” stands for the original code name of Windows 95, which was Chicago). You can call any Win32 function under this API, but many of the functions are not implemented—especially those that relate to NT-specific features that are not built into Windows 95. There may be some operating system-specific API functions in Win32c that are not yet incorporated into the latest implementation of Win32 on Windows NT. |
| Win32s | This is a smaller subset of Win32 that is supported under Windows and Windows for Workgroups 3x. It is not supported by Visual Basic 4.0 or 5.0. |

This book will focus on the Win32 API and will emphasize those features supported in Win32c, as it seems likely that Windows 95 will become the dominant operating system platform for the foreseeable future. It will not cover Win32s at all, as this API subset is unlikely to see widespread use. You will also learn how to write programs that are compatible with multiple operating systems, including how to write Visual Basic 4.0 programs that can compile successfully on both Win16 and Win32 platforms.

Let me stress once again the importance of the fact that a particular API is NOT the same thing as an operating system. For example, in many cases functions or constants in this book are defined as Windows 95 only. This does not necessarily mean that the function will not work on Windows NT. Win32 is rapidly incorporating many new Windows 95 features and can run the Windows 95 desktop—meaning that many Windows 95-only functions are available in some NT environments.

The Major Windows DLLs

The core functionality of the Win32 API is divided into three major dynamic link libraries and a number of smaller DLLs as shown in Table 1.2.

Table 1.2 Windows DLLs

| DLL Name | Description |
|--------------|--|
| KERNEL32.DLL | Low-level operating functions. Memory management, task management, resource handling, and related operations are found here. |
| USER32.DLL | Functions relating to Windows management. Messages, menus, cursors, carets, timers, communications, and most other nondisplay functions can be found here. |



Table 1.2 Windows DLLs (Continued)

| DLL Name | Description |
|---|--|
| GDI32.DLL | The Graphics Device Interface library. This DLL contains functions relating to device output. Most drawing, display context, metafile, coordinate, and font functions are in this DLL. |
| COMDLG32.DLL LZ32.DLL VERSION.DLL | These DLLs provide additional capabilities including support for common dialogs, file compression, and version control. In some cases the capabilities are accessible directly; in others you will need to use APIGID32.DLL. |
| APIGID32.DLL | Provided on the CD-ROM that comes with this book. This DLL provides VB interfaces for some incompatible Windows API functions, plus a few additional utility functions. |

A Universe of Extension Libraries

You have already seen how Windows is made up of several core dynamic link libraries. One of the curious effects of this operating system architecture is that it makes it possible to change Windows incrementally. When Microsoft wanted to add electronic mail capability to the operating system, they did not need to update the entire Windows system, all they had to do was add a new dynamic link library. As it turns out, most of the new features that have appeared in Windows over the past few years have actually taken the form of new DLLs, and with these new DLLs came new extensions to the API. Table 1.3 lists some of the most important extension DLLs and the functionality that they provide.

Table 1.3 Major Extension Libraries

| DLL Name | Description |
|--------------|--|
| COMCTL32.DLL | This DLL implements a new set of Windows controls, such as the tree list and rich text edit control. This DLL was initially created for Windows 95, but is now available for Windows NT as well. |
| MAPI32.DLL | This DLL provides a set of functions that lets any application work with electronic mail. |
| NETAPI32.DLL | This DLL provides a set of API functions for the access and control of networks. |
| ODBC32.DLL | This is one of the DLLs that implements ODBC—Open Database Connectivity. These functions provide a standard API that can be used to work with different types of databases. |
| WINMM.DLL | This DLL provides access to a system's multimedia capabilities. |



The Windows API (be it Win16 or Win32) is evolving rapidly. The complexity and number of functions are growing so quickly that it is nearly impossible to keep up with them. Fortunately, it is not necessary to become an expert in the entire Windows API to use it effectively. It is important to become acquainted with the general architecture of Windows and the fundamental concepts that make it possible to understand the various API functions. It is also important to learn how to read API documentation and function declarations so that you can create the declarations necessary to access those functions from Visual Basic. The rest of Part 1 tackles these two critical subjects.

The Different Flavors of Visual Basic

Even as Windows was evolving, an interesting thing was happening to Visual Basic itself. You see, Visual Basic actually consists of two separate parts that closely interact. There is the form engine—the part of the language that manages windows and controls and provides a development environment—and the language engine—the part of Visual Basic that compiles the code that you write into a low-level pseudo code, then executes it.

In versions 1.0 through 3.0 of Visual Basic, parts of the language engine were written in assembly language, which provided good performance but made it very unportable and difficult to extend to 32 bits. At the same time, a group at Microsoft was working on a new language engine called “Object Basic,” which was designed from scratch to be portable, to support both 16- and 32-bit platforms, and to include support for the new OLE technology. The resulting language engine, now referred to as VBA (Visual Basic for Applications), first saw use in Microsoft Excel, and is gradually spreading to other Microsoft applications. It is becoming, in effect, the “macro” language for Microsoft’s applications. This term, however, is misleading—“macro” language suggests that it is limited as compared to a real language, while in reality VBA is a full-featured Basic language.

How full featured? It turns out that VBA is the language engine for Visual Basic 4.0 and above.

Now add to this statement the additional fact that the ability of Visual Basic to call Windows API and DLL functions is part of the VBA language engine, and a rather astonishing truth emerges: Virtually all of the techniques described in this book can be applied not only to Visual Basic, but to every Microsoft tool or application that uses the VBA language engine!

This book will continue to focus on the standalone Visual Basic language due to time and space limitations, but several examples for other Microsoft VBA-based applications can be found in the VBPG32\SAMPLES\VBA directory on the CD-ROM that comes with this book.