

COREL®



# WordPerfect® **OFFICE X5**

User Guide for VBA



# Contents

---

<b>Introduction</b> .....	<b>1</b>
About VBA in WordPerfect Office X5 .....	1
About this guide .....	1
About additional resources .....	3
About Corel .....	3
<b>Understanding VBA</b> .....	<b>5</b>
What is VBA? .....	5
What is automation? .....	6
Who is VBA designed for? .....	6
How does VBA compare with other programming languages? .....	7
What are the main elements used in VBA? .....	8
What is an object model? .....	9
How is VBA code structured? .....	10
<b>Getting started with VBA</b> .....	<b>19</b>
Using the Visual Basic toolbar .....	19
Using the VB Editor .....	20
Using the Project Explorer in the VB Editor .....	21
Using the Properties window in the VB Editor .....	22
Using the Code window in the VB Editor .....	23
Using the toolbars in the VB Editor .....	26
Using the Object Browser in the VB Editor .....	27
<b>Working with macros</b> .....	<b>33</b>
Creating macros .....	33
Running macros .....	35
Debugging macros .....	35
Using the debugging windows .....	37
<b>Creating user-interfaces for macros</b> .....	<b>41</b>
Creating dialog boxes for macros .....	41
Coding dialog boxes .....	44
Designing dialog boxes .....	45

Providing help for macros .....	50
<b>Organizing and deploying macros .....</b>	<b>53</b>
Organizing macros.....	53
Deploying macros .....	53
<b>Appendix: About the object models .....</b>	<b>55</b>
About the WordPerfect object model.....	55
About the Quattro Pro object model .....	56
About the Presentations object model .....	56
<b>Glossary.....</b>	<b>57</b>
<b>Index .....</b>	<b>61</b>



# Introduction

---

This guide is intended as a resource for developing and distributing Microsoft® Visual Basic® for Applications (VBA) solutions in Corel® WordPerfect® Office X5.

This introductory section provides information about the following:

- VBA in WordPerfect Office X5
- this guide
- additional resources
- Corel

## About VBA in WordPerfect Office X5

Although Corel® WordPerfect® X5, Corel® Quattro Pro® X5, and Corel® Presentations™ X5 support VBA version 6.3, it is important to note that VBA is not included with WordPerfect Office X5. You can use VBA in WordPerfect Office X5 only if you have installed a software product that includes VBA 6.3.

VBA comes with a fully integrated development environment (IDE) that provides contextual pop-up lists, syntax highlighting, line-by-line debugging, and visual designer windows. These helpful prompts and aids create a particularly friendly learning environment for inexperienced developers.

## About this guide

This guide was designed as a resource for the following:

- exploring the VBA IDE and many of its advanced features
- understanding the most important WordPerfect Office X5 functions and how to use them
- examining how to package and deploy VBA solutions developed for WordPerfect Office X5

This guide should be used by anyone who is interested in automating simple and complex tasks in WordPerfect Office X5 or who is developing commercial solutions

that integrate with WordPerfect Office X5. It is assumed that the reader already has experience with at least one other procedural programming language, such as BASIC, Microsoft Visual Basic, C, C++, Java™, Pascal, Cobol, or Fortran. This guide does not describe the basics of procedural programming, such as functions, conditional branching, and looping. Non-programmers should learn the basics of programming in a language such as Visual Basic or VBA before using this document to develop WordPerfect Office X5 solutions.

## Contents

This guide is organized into the following chapters, which deal with specific aspects of automating tasks and building solutions in WordPerfect Office X5:

- “Understanding VBA” on page 5 — provides a brief introduction to VBA.
- “Getting started with VBA” on page 19 — lets you explore the VBA workspace in WordPerfect, Quattro Pro, and Presentations.
- “Working with macros” on page 33 — shows you how to create, run, and debug macros.
- “Creating user-interfaces for macros” on page 41 — demonstrates how to provide dialog boxes, toolbars and buttons, user interaction, and help for your macros.
- “Organizing and deploying macros” on page 53 — helps you organize and deploy the macros you create.

This guide also provides an appendix (page 55), which provides information on the object models for WordPerfect Office X5.

Finally, a glossary (page 57) defines the terms that are used in this guide.

## Conventions

The following conventions are used in this guide:

When you see	This is
--------------	---------



A note — presents information such as conditions for performing a procedure



A tip — presents information such as procedure shortcuts, variations, or benefits

---

In addition, this guide uses a few formatting conventions:

- User-interface items are displayed in **boldface**.
- Glossary items are displayed in *italics*.

- Information that can be substituted, such as a path or filename, is displayed in *<italics and between angle brackets>*.
- Coding is formatted in a monospace font.

## About additional resources

The main Help files for WordPerfect, Quattro Pro, and Presentations provide basic information on using VBA. To access these Help files, additional resources that install with WordPerfect Office X5, and even more resources on the Web, please see the Reference Center that installs with WordPerfect Office X5.

## To access the Reference Center

- 1 Do one of the following:
  - On the Windows® taskbar, click **Start ▶ Programs ▶ WordPerfect Office X5 ▶ Reference Center**.
  - In WordPerfect, Quattro Pro, or Presentations, click **Help ▶ Reference Center**.
- 2 Click any tab to display the available documentation for that category.
- 3 Click any category entry to display its associated documentation.

## About Corel

Corel is one of the world's top software companies, with more than 100 million active users in over 75 countries. We develop software that helps people express their ideas and share their stories in more exciting, creative, and persuasive ways. Through the years, we've built a reputation for delivering innovative, trusted products that are easy to learn and use, helping people achieve new levels of productivity. The industry has responded with hundreds of awards for software innovation, design, and value.

Our award-winning product portfolio includes some of the world's most widely recognized and popular software brands, including CorelDRAW® Graphics Suite, Corel® Painter™, Corel DESIGNER® Technical Suite, Corel® PaintShop Photo™ Pro, Corel® VideoStudio®, Corel® WinDVD®, Corel® WordPerfect® Office, WinZip®, and the recently released Corel® Digital Studio™ 2010. Our global headquarters are in Ottawa, Canada, with major offices in the United States, United Kingdom, Germany, China, Taiwan, and Japan.







# Understanding VBA

---

Before getting started with VBA in WordPerfect Office X5, it's important to understand a little bit about VBA in general.

This chapter answers the following questions:

- What is VBA?
- What is automation?
- Who is VBA designed for?
- How does VBA compare with other programming languages?
- What are the main elements used in VBA?
- What is an object model?
- How is VBA code structured?

## What is VBA?

*Visual Basic for Applications* (more commonly known as *VBA*) is a built-in programming language that can automate repetitive functions and create intelligent solutions in WordPerfect, Quattro Pro, and Presentations.

VBA is both a language and an editor. It is not possible to have the language without the editor, nor is it possible to edit VBA in anything but the VB Editor or to run VBA programs without the VB Editor.

VBA is developed by Microsoft and is built into almost all of its desktop applications, including Microsoft® Office. VBA is licensed by Microsoft to other companies, including Corel (in CorelDRAW Graphics Suite, Corel DESIGNER Technical Suite, and Corel WordPerfect Office), Autodesk, Inc. (in AutoCAD®), and IntelliCAD Technology Consortium (in IntelliCAD®). This makes Corel applications compatible with a wide array of applications that support VBA.



For a complete list of applications that support VBA, consult the Microsoft Web site.

It is not necessary for an application to support VBA in order for the WordPerfect Office X5 VBA engine to control that application. That means you can

build solutions in WordPerfect Office X5 that access databases specialized content editors, XML documents, and more.

## What is automation?

Most actions that you can do in WordPerfect Office X5 can be done programmatically through VBA. This programmability of is called *automation*. Automating repetitive tasks can save time and reduce effort, while automating complex tasks can make possible the otherwise impossible.

In its simplest form, automation is simply recording a sequence of actions so that you can play them back time and again. The term *macro* has come to include any code that is accessible to VBA while running within the process, even though some of that code might be far more advanced than a mere set of recorded actions. For the purposes of this guide, a macro refers to VBA *functions* and *subroutines* (which are explained in “Building functions and subroutines” on page 12).

While it is possible to record a sequence of actions in WordPerfect Office X5, the real power of automation and VBA is that these recordings can be edited to provide conditional and looping execution.

## Who is VBA designed for?

VBA can be used by both non-programmers and programmers alike.

### VBA for non-programmers

VBA is based on the successful Microsoft Visual Basic (VB) programming language. The main difference between VBA and VB is that you cannot create stand-alone executable (EXE) files using VBA, whereas you can with VB. That is to say, using VBA, you can create only programs that run inside the host application (in this case, WordPerfect, Quattro Pro, or Presentations).

VB is a “visual” version of the BASIC programming language. This means that it is a very easy language to learn, particularly because it provides visual cues within the editor. Microsoft has added a great deal to the original BASIC language, and it is now a powerful and fast language (although not as powerful as Java or C++, nor as quick as C).

The aim of this guide is not to teach you how to become a programmer but instead to teach experienced programmers how to apply their skills to developing useful solutions

within WordPerfect Office X5. If you are not a programmer, you may find it useful to refer to the many books that have been written about VBA and VB before continuing to read this guide.

## **VBA for programmers**

VBA is an in-process automation controller. In other words, VBA can be used to control the features of WordPerfect Office X5 that can be automated, and VBA runs efficiently by bypassing the interprocess synchronization mechanisms. However, the automation that the in-process VBA can access can also be accessed by the following:

- external out-of-process automation controllers (OLE clients)
- applications that are developed in programming languages (such as VB, Visual C++®, Windows® Script Host, and C++) that can be used to develop OLE clients
- the VBA engines of other applications

## **How does VBA compare with other programming languages?**

VBA has many similarities with most modern, procedural programming languages, including Java and JavaScript®, C and C++, and Windows Script Host. However, VBA runs as an in-process automation controller, whereas the other languages (apart from JavaScript) are used to compile stand-alone applications.

### **VBA compared with Java and JavaScript**

VBA is similar to Java and JavaScript in that it is a high-level, procedural programming language that has full garbage collection and very little memory-pointer support. (See “Using memory pointers and memory allocation” on page 14 for more information.) In addition, code developed in VBA — much like code developed in Java and JavaScript — supports on-demand compilation and can be executed without being compiled.

VBA has another similarity with JavaScript in that it cannot be executed as a standalone application. JavaScript is embedded within Web pages, as a mechanism for manipulating the Web browser’s document object model (or “DOM”). Likewise, VBA programs are executed inside a host environment — in this case, WordPerfect, Quattro Pro, or Presentations — so as to manipulate the host’s *object model* (which is discussed in “What is an object model?” on page 9).

Most VBA applications can be compiled to P-code so as to make them run more quickly, although the difference is hardly noticeable given the sophistication of today’s computer hardware. Java can be similarly compiled; JavaScript, however, cannot.

Finally, whereas VBA uses a single equals sign (=) for both comparison and assignment, Java and JavaScript use a single equals sign (=) for assignment and two equals signs (==) for Boolean comparison. (For more information on Boolean comparison and assignment in VBA, see “Using Boolean comparison and assignment” on page 15.)

### **VBA compared with C and C++**

Visual Basic — similarly to C and C++ — uses functions. In VB, functions can be used to return a value but subroutines cannot. In C and C++, however, functions are used regardless of whether you want to return a value. (For more information on functions and subroutines, see “Building functions and subroutines” on page 12.)

VBA allocates and frees memory “transparently.” In C and C++, however, the developer is responsible for most memory management. This makes using strings in VBA even simpler than using the `CString` class in C++.

Finally, whereas VBA uses a single equals sign (=) for both comparison and assignment, C and C++ use a single equals sign (=) for assignment and two equals signs (==) for Boolean comparison. (For more information on Boolean comparison and assignment in VBA, see “Using Boolean comparison and assignment” on page 15.)

### **VBA compared with Windows Script Host**

Windows Script Host (WSH) is a useful addition to Windows for doing occasional scripting and automation of Windows tasks. WSH is an out-of-process automation controller that can be used to control WordPerfect Office X5. However, because WSH scripts cannot be compiled (and must be interpreted as they are executed) and must be run out of process, they tend to be slow.

WSH is a host for a number of scripting languages, each of which has its own syntax. However, the standard language used by WSH is a macro language resembling Visual Basic, so for standard scripts, the syntax is the same as in VBA.

## **What are the main elements used in VBA?**

If you’ve ever developed object-oriented code in C++, Borland® Delphi®, or Java, you’re already familiar with the concepts of “classes,” “objects,” “properties,” and “methods,” but let’s re-examine them in greater detail as they apply to VBA.

A *class* is a description of something. For example, the class “car” is a small vehicle with an engine and four wheels.

An *object* is an instance of a class. If we extend the car metaphor, then the actual, physical car that you go out and purchase for the purposes of driving is an object (that is, an instance of the class “car”).

Most classes have *properties*. For example, the properties of the class “car” are that it is small, it has an engine, and it has four wheels. Every instance of the class “car” (that is, every object in that class) also has properties such as color, speed, and number of seats. Some properties, called “read-only” properties, are fixed by the design of the class; for example, the number of wheels or seats does not (usually) vary from car to car. However, other properties can be changed after the object has been created; for example, the speed of the car can go up and down, and, with a bit of help, its color can be changed.

A *method* is an operation that the object can have performed on itself. In the example of the class “car,” the car can be made to go faster and slower, so two methods for the class are “accelerate” and “decelerate.”

Objects are often made up of other smaller objects. For example, a car contains four objects of the class “wheel,” two objects of the class “headlight,” and so on. Each of these child objects has the same properties and methods of its class-type. This parent/child relationship of objects is an important one to recognize, particularly when referencing an individual object.

Some classes “inherit” features from their parents.

## **What is an object model?**

VBA relies on an application’s *object model* for communicating with that application and modifying its documents. Without an object model, VBA cannot query or change an application’s documents.

Object models in software provide a high level of structure to the relationship between parent and child objects.

Remember, though, that the object model is the map that the VBA language uses to access the various members — objects, methods, and properties — of a document, and to make changes to those members. Without the object model, it is simply impossible to gain access to the objects in the document.

## Understanding object hierarchy

In any object model, each object is a child of another object, which is a child of another object. Also, each object has child members of its own — properties, objects, and methods. All of this comprises an object hierarchy that is the object model.

In order to “drill down” through the layers of hierarchy to get to the object or member that you want, you must use a standard notation. In VBA, as in many object-oriented languages, the notation is to use a period ( . ) to indicate that the object on the right is a member (or child) of the object on the left.

It is not usually necessary to use the full hierarchical (or fully qualified) reference to an object or its properties. Some of the object-syntax in the fully qualified reference is mandatory or required; however, other syntax is optional (because a shortcut object for it is available, or because it is implicit or implied), and so it can either be included for clarity or omitted for brevity.

A *shortcut object* is merely a syntactical replacement for the long-hand version of the object.

For detailed information on the WordPerfect Office X5 object models, see the Appendix: About the object models.

## How is VBA code structured?

Because VBA is a procedural language that shares much in common with all procedural languages, your current knowledge should help you get off to a quick start with VBA.

This section examines the following topics on VBA structure and syntax:

- Declaring variables
- Building functions and subroutines
- Ending lines
- Including comments
- Using memory pointers and memory allocation
- Defining scope
- Using Boolean comparison and assignment
- Using logical and bitwise operators
- Providing message boxes and input boxes



The VB Editor formats all of the code for you (as discussed in “Formatting code automatically” on page 23). The only custom formatting that you can do is to change the size of the indentations.

VBA can create object-oriented classes, although these are a feature of the language and are not discussed in detail in this guide.

## Declaring variables

In VBA, the construction for declaring *variables* is as follows:

```
Dim foobar As Integer
```

The built-in data types are Byte, Boolean, Integer, Long, Single, Double, String, Variant, and several other less-used types including Date, Decimal, and Object.

Variables can be declared anywhere within the body of a function, or at the top of the current module. However, it is generally a good practice to declare a variable before it is used; otherwise, the compiler interprets it as a Variant, and inefficiencies can be incurred at run time.



Booleans take False to be zero and True to be any other value, although converting from a Boolean to a Long results in True being converted to a value of -1.



To get more information about one of the built-in data types, type it into the code window, select it, and then press **F1**.

Data structures can be built by using the following syntax:

```
Public Type fooType
    item1 As Integer
    item2 As String
End Type
Dim myTypedItem As fooType
```

The items within a variable declared as type fooType are accessed using dot notation:

```
myTypedItem.item1 = 5
```

## Declaring strings

Strings in VBA are much simpler than in C. In VBA, strings can be added together, truncated, searched forwards and backwards, and passed as simple arguments to functions.

To add two strings together, simply use the concatenation operator (&) or the addition operator (+):

```
Dim string1 As String, string2 As String
string2 = string1 & " more text" + " even more text"
```

In VBA, there are many functions for manipulating strings, including `InStr()`, `Left()`, `Mid()`, `Right()`, `Len()`, and `Trim()`.

## Declaring arrays

To declare an *array*, use parentheses — that is, the ( and ) symbols:

```
Dim barArray (4) As Integer
```

The value defines the index of the last item in the array. Because array indexes are zero-based by default, there are five elements in the preceding sample array (that is, elements 0 thru 4, inclusive).

Arrays can be resized by using `ReDim`. For example, the following code adds an extra element to `barArray`, but preserves the existing contents of the original five elements:

```
ReDim Preserve barArray (6)
```

Upper and lower bounds for an array can be determined at run time with the functions `UBound()` and `LBound()`.

Multi-dimensional arrays can be declared by separating the dimension indexes with commas:

```
Dim barArray (4, 3)
```

## Building functions and subroutines

VBA uses both *functions* and *subroutines* (or “subs”). Functions can be used to return a value, whereas subs cannot.

In VBA, functions and subs do not need to be declared before they are used, nor before they are defined. In fact, functions and subs need to be declared only if they actually exist in external system dynamic-linked libraries (DLLs).

Typical functions in a language such as Java or C++ can be structured as follows:



```

void foo( string stringItem ) {
    // The body of the function goes here
}
double bar( int numItem ) { return 23.2; }

```

In VBA, however, functions are structured as in the following example:

```

Public Sub foo (stringItem As String)
    ' The body of the subroutine goes here
End Sub
Public Function bar (numItem As Integer) As Double bar = 23.2
End Function

```

To force a function or sub to exit immediately, you can use `Exit Function` or `Exit Sub` (respectively).

## Declaring enumerated types

To declare an *enumerated type*, use the following construction:

```

Public Enum fooEnum
    ItemOne
    ItemTwo
    ItemThree
End Enum

```



The first item in an enumerated type is assigned, by default, a value of zero.

## Ending lines

In VBA, each statement must exist on its own line, but no special character is required to denote the end of each line. (This is in contrast to the many programming languages that use the semicolon to separate individual statements.)

To break a long VBA statement over two or more lines, each of the lines (other than the last line) must end with an underscore ( `_` ) preceded by at least one space:

```

newString = fooFunction ("This is a string", _
                        5, 10, 2)

```

It is also possible to combine several statements in a single line by separating them with colons:

```
a = 1 : b = 2 : c = a + b
```



A line cannot end with a colon. Lines that end with a colon are labels used by the `Goto` statement.

## Including comments

Comments in VBA — similarly to in C++ and Java — can be created only at the end of a line. Comments begin with an apostrophe ( `'` ) and terminate at the end of the line.

Each line of a multi-line comment must begin with its own apostrophe:

```
a = b ' This is a really interesting piece of code that
      ' needs so much explanation that I have had to break
      ' the comment over multiple lines.
```

To comment out large sections of code, use the following code (similarly to in C or C++):

```
#If 0 Then ' That's a zero, not the letter 'oh'.
  ' All this code will be ignored by
  ' the compiler at run time!
#End If
```

## Using memory pointers and memory allocation

VBA does not support C-style memory pointers. Memory allocation and garbage collection are automatic and transparent, just as in Java and JavaScript (and some C++ code).

## Passing values “by reference” and “by value”

Most languages, including C++ and Java, pass an argument to a procedure as a copy of the original. If the original must be passed, then one of two things can happen:

- a memory pointer is passed that directs to the original in memory
- a reference to the original is passed

The same is true in VB, except that passing a copy of the original is called *passing by value* and passing a reference to the original is called *passing by reference*.

By default, function and subroutine parameters are passed by reference. This means that a reference to the original variable is passed in the procedure's argument, and so changing that argument's value within the procedure, in effect, changes the original variable's value as well. This is a great way of returning more than one value from a function or sub. To explicitly annotate the code to indicate that an argument is being passed by reference, you can prefix the argument with `ByRef`.

If you want to prevent the procedure from changing the value of the original variable, you can force the copying of an argument. To do this, prefix the argument with `ByVal`, as shown in the example that follows. This `ByRef/ByVal` functionality is similar to the ability of C and C++ to pass a copy of a variable, or to pass a pointer to the original variable.

```
Private Sub fooFunc (ByVal int1 As Integer, _  
                    ByRef long1 As Long, _  
                    long2 As Long) ' Passed ByRef by default
```

In the preceding example, arguments `long1` and `long2` are both, by default, passed by reference. Modifying either argument within the body of the function modifies the original variable; however, modifying `int1` does not modify the original because it is a copy of the original.

## Defining scope

You can define the *scope* of a data type or procedure (or even an object). Data types, functions, and subs (and members of classes) that are declared as `private` are visible only within that module (or file), while functions that are declared as `public` are visible throughout all the modules; however, you may have to use fully qualified referencing if the modules are almost out of scope — for example, if you are referencing a function in a different project.

Unlike C, VBA does not use braces — that is, the `{` and `}` symbols — to define local scope. Local scope in VBA is defined by an opening function or sub definition statement (that is, `Function` or `Sub`) and a matching `End` statement (that is, `End Function` or `End Sub`). Any variables declared within the function are available only within the scope of the function itself.

## Using Boolean comparison and assignment

In VB, Boolean comparison and assignment are both performed by using a single equals sign (`=`):

```
If a = b Then c = d
```

This is in contrast to many other languages that use a double equals sign for a Boolean comparison and a single equals sign for assignment:

```
if( a == b ) c = d;
```

The following code, which is valid in C, C++, Java, and JavaScript, is invalid in VBA:

```
if( ( result = fooBar( ) ) == true )
```

This would have to be written in VBA as the following:

```
result = fooBar( )
```

```
If result = True Then
```

For other Boolean comparisons, VBA uses the same operators as other languages (except for the operators for “is equal to” and “is not equal to”). All the Boolean-comparison operators are provided in the following table:

<b>Comparison</b>	<b>VBA operator</b>	<b>C-style operator</b>
Is equal to	=	==
Is not equal to	<>	!=
Is greater than	>	>
Is less than	<	<
Is greater than or equal to	>=	>=
Is less than or equal to	<=	<=

The result of using one of the Boolean operators is always either `True` or `False`.

## Using logical and bitwise operators

In VBA, logical operations are performed by using the keywords `And`, `Not`, `Or`, `Xor`, `Imp`, and `Eqv`, which perform the logical operations AND, NOT, OR, Exclusive-OR, logical implication, and logical equivalence (respectively). These operators also perform Boolean comparisons.

The following code shows a comparison written in C or a similar language:

```
if( ( a && b ) || ( c && d ) )
```

This would be written as follows in VBA:

```
If ( a And b ) Or ( c And d ) Then
```

Alternatively, the above could be written in the following full long-hand form:

```
If ( a And b = True ) Or ( c And d = True ) = True Then
```

The following table provides a comparison of the four common VBA logical and bitwise operators, and the C-style logical and bitwise operators used by C, C++, Java, and JavaScript:

VBA operator	C-style bitwise operator	C-style Boolean operator
And	&	&&
Not	~	!
Or		
Xor	^	

## Providing message boxes and input boxes

You can present simple messages to the user by using the `MsgBox` function:

```
Dim retval As Long
retval = MsgBox("Click OK if you agree.", _
               vbOKCancel, "Easy Message")
If retval = vbOK Then
    MsgBox "You clicked OK.", vbOK, "Affirmative"
End If
```

You can also get strings from the user by using `InputBox` function:

```
Dim inText As String
inText = InputBox("Input some text:", "type here")
If Len(inText) > 0 Then
    MsgBox "You typed the following: " & inText & "."
End If
```

If the user clicks **Cancel**, the length of the string returned in `inText` is zero.





## Getting started with VBA

---

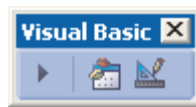
Now that you understand a bit about VBA, you're ready to get started with macros.

This chapter covers the following topics:

- Using the Visual Basic toolbar
- Using the VB Editor
- Using the Project Explorer in the VB Editor
- Using the Properties window in the VB Editor
- Using the Code window in the VB Editor
- Using the toolbars in the VB Editor
- Using the Object Browser in the VB Editor

### Using the Visual Basic toolbar

WordPerfect and Quattro Pro feature a toolbar that provides easy access to the VB Editor.



*The Visual Basic toolbar*

The toolbar buttons provide the following functions:

- playing macros
- opening the VB Editor
- switching the VB Editor between its modes for designing and running macros

### To display the Visual Basic toolbar in WordPerfect or Quattro Pro

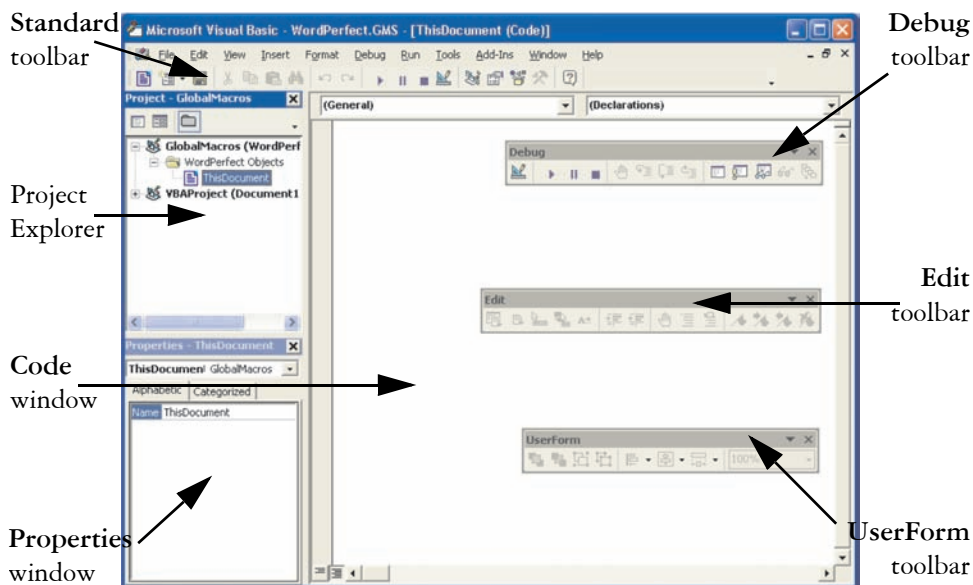
- 1 Click View ► Toolbars.
- 2 Enable the Visual Basic check box, and then click OK.

## Using the VB Editor

The VB Editor that is included with VBA is similar to the one included with full Visual Basic.

The VB Editor lets you develop code and dialog boxes, browse the object tree and the modules within each project, set individual properties for objects, and debug code. However, it's important to note that the VB Editor for VBA cannot compile executable (EXE) program files.

The VB Editor features several windows and toolbars, all of which are discussed in this section. The three available windows are the Project Explorer (see page 21), the **Properties** window (see page 22), and the **Code** window (see page 23). The four available toolbars (see page 26), are the **Standard** toolbar, the **Debug** toolbar, the **Edit** toolbar, and the **UserForm** toolbar, of which you will use the **Standard** and **Debug** toolbars most often.



*The VB Editor*

The VB Editor also lets you access the Object Browser (see page 27).

You can invoke the VB Editor from within WordPerfect, Quattro Pro, or Presentations. Although this starts VBA as a new application in Windows, it runs within the WordPerfect, Quattro Pro, or Presentations process.



## To start the VB Editor

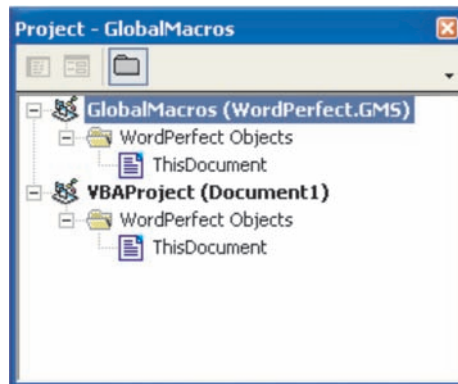
- Click Tools ► Visual Basic ► Visual Basic Editor, or press Alt + F11.



To switch between the VB Editor and WordPerfect, Quattro Pro, or Presentations, use the Windows taskbar, or press Alt + F11 or Alt + Tab.




## Using the Project Explorer in the VB Editor




The Project Explorer is essential for navigating VBA projects and their constituent documents/objects, forms, *modules*, and *class modules*.



*The Project Explorer with a module selected*

Each type of item in the Project Explorer has an icon assigned to it:

Icon	Meaning
	project
	folder
	document/object

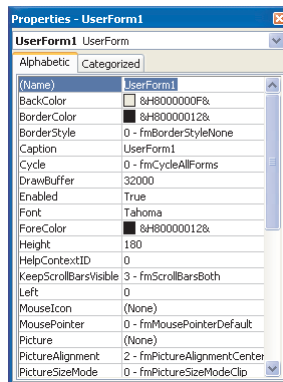
Icon	Meaning
	form
	module
	class module

## To display or hide the Project Explorer

- Click View ► Project Explorer, or press Ctrl + R.

## Using the Properties window in the VB Editor

The **Properties** window lists all of the editable properties for the currently selected object. Many objects in VBA — including projects, modules, and forms and their controls — have property sheets that can be modified.



*The Properties window, showing the properties of a form*

The **Properties** window is automatically updated when you select an object, or when you change the properties of the selected object by using other methods (for example, by using the mouse to move and resize form controls).

## To display or hide the Properties window

- Click View ► Properties window, or press F4.

## Using the Code window in the VB Editor

The Code window is where you spend most of your time when working on macros. A standard code editor in the style of Microsoft® Visual Studio®, the Code window lets you format code automatically, color and check syntax automatically, jump to definitions, and use contextual pop-up lists and automatic completion.

If you are already familiar with any of the Microsoft Visual Studio editors, the VB Editor's Code window will be entirely familiar to you.

### Formatting code automatically

The VB Editor formats code automatically for you — even the capitalization of keywords, functions, subroutines, and variables is taken care of by the VB Editor, irrespective of what you type.

You cannot custom-format code, although you can set the indentation for each line, as well as the placing of custom line breaks.

When it comes to calling functions and subs, you must adhere to the following rules:

- If you are calling a function and you are using the returned value, the parentheses around the parameters are mandatory (just as in most modern programming languages):  
`a = fooFunc (b, c)`
- However, if the returned value from a function call is being discarded, or if you are calling a sub, the parentheses must be left out (unlike in most other languages):  
`barFunc d, e`  
`fooBarSub f`
- If you prefer always to see the parentheses, use the `Call` keyword before the function or sub call:  
`Call barFunc (d, e)`  
`Call fooBarSub (f)`

## Coloring syntax automatically

When you develop code in the **Code** window, the editor colorizes each word according to its classification:

- VBA keywords and programming statements are usually displayed in blue.
- Comments are displayed in green.
- All other text is displayed in black.

This colorization makes the code much easier to read.

The **Code** window also uses the following colorization techniques:

- Lines of code containing errors are displayed in red.
- Selected text is white on blue.
- The line where execution paused for debugging is shown as a yellow highlight.
- Breakpoints that you set for debugging purposes are shown as a red dot in the left margin with the code in white on a red background.
- Bookmarks (which you set in the code) are indicated by a blue dot in the left margin.



Breakpoints (along with bookmarks) are lost when you quit the application. For more information on them, see “Setting breakpoints” on page 36.



You can modify the default colors for syntax highlighting by clicking **Tools ▶ Options**, clicking the **Editor** format tab, and making your changes.

## Checking syntax automatically

Every time you move the cursor out of a line of code, the editor checks the syntax of the code in that line; if it finds an error, it changes the color of the text of that line to red and displays a pop-up warning. This real-time checking is useful (particularly when you are learning VBA) because it indicates many possible errors in the code without having you run the code.



You can disable pop-up warnings by clicking **Tools ▶ Options**, clicking the **Editor** tab, and then disabling the **Auto syntax check** check box. The VB Editor still checks the syntax and highlights erroneous lines in red, but it stops displaying a warning when you paste text from another line of code.

## Jumping to definitions

You can jump directly to the definition of a variable, procedure, or object by right-clicking the item in the **Code** window and then clicking **Definition**. This takes you either to the definition of the variable or function in the code or to the object's definition in the Object Browser.



To return to where you requested the definition, right-click, and then click **Last position in the Code window**.

## Using contextual pop-up lists and automatic completion

As you write procedures and define variables, the VB Editor adds these items to an internal list that already contains all of its built-in keywords and enumerated values. As you type, the VB Editor presents you with a list of candidate words that you may want to insert at the current position; this list is contextual, so the VB Editor usually presents only the words that are valid for the current position.

This list makes code development quicker and more convenient, particularly because you do not need to remember every function and variable name but can instead choose them from the list provided. If you type the first few characters of the word you want to use, the list advances to the nearest candidate that matches the characters you've entered. Select the word you want to use, and either type the character that you want to have follow the word (typically a space, line feed, parenthesis, period, or comma) or press **Tab** or **Ctrl + Enter** to enter only the word.



To force the pop-up menu display, you can press **Ctrl + Spacebar**. The menu scrolls to the word that most closely matches the characters that you have typed so far. This technique is also useful for filling parameter lists when calling a function or subroutine. If there is only one exact match, the VB Editor inserts the word without popping up the list. To display the pop-up list for the selected keyword at any time without auto-filling it, press **Ctrl + J**.

## Using the toolbars in the VB Editor

The VB Editor features four toolbars that you can use to carry out your VBA tasks. The **Standard** toolbar is the default toolbar.



*The Standard toolbar*

The **Debug** toolbar contains buttons for common debugging tasks (as discussed in “Debugging macros” on page 35).



*The Debug toolbar*

The **Edit** toolbar contains buttons for common editing tasks.



*The Edit toolbar*

The **UserForm** toolbar contains buttons specific to designing forms (as discussed in “Designing dialog boxes” on page 45).



*The UserForm toolbar*

You can choose to display or hide each toolbar.

### To display or hide a toolbar

- Click **View** ► **Toolbars**, and then click the command that corresponds to the toolbar you want to display or hide.

A check mark next to a command indicates that its toolbar is currently displayed.

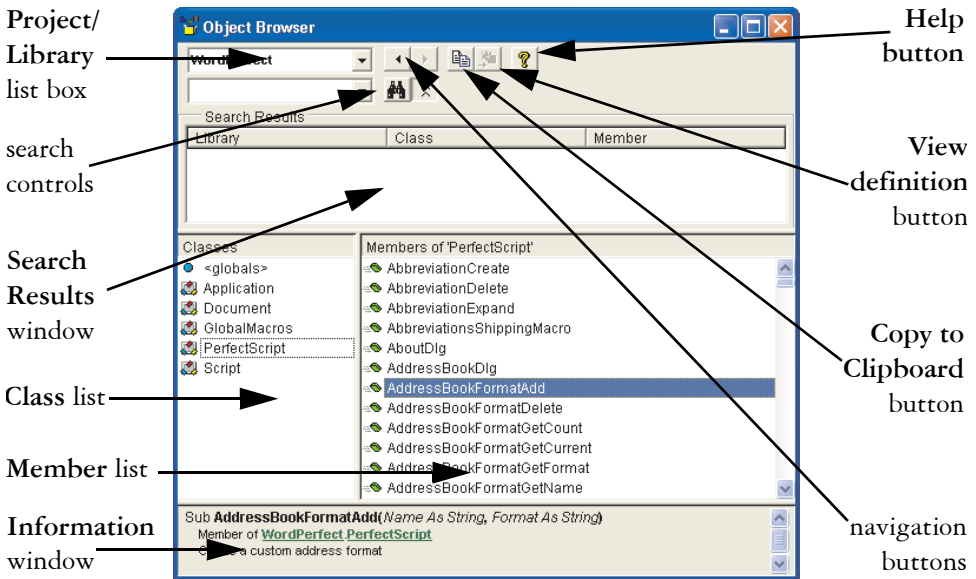


You can “float” a toolbar by dragging it from the menu bar.

You can dock a toolbar by dragging it to the menu bar.

## Using the Object Browser in the VB Editor

The Object Browser is one of the most useful tools provided by the VB Editor. The Object Browser displays the entire object model of all referenced components (that is, all ActiveX® or OLE objects that are used by the project) and, most importantly, the object model of WordPerfect, Quattro Pro, or Presentations — all in an easy-to-use, structured format.



*The Object Browser window*

To open the Object Browser, click **View** ► **Object Browser**, or press F2.



To reference the object models for other applications, click **Tools** ► **References**. Referenced components can be accessed by the VBA code.

All of the referenced objects — plus the current module — are listed in the **Project/Library** list box in the upper-left corner of the Object Browser. By default, all of the member classes for the referenced objects are provided in the **Class** list.



It is easier to use the Object Browser when only one project or library is displayed. To display only one project or library, choose it from the **Project/Library** list box.

More detailed information follows about certain Object Browser elements.






## Using the Class list

The **Class** list shows all of the classes in the current project or library.



When you select a class in the **Class** list, the members of that class are shown in the **Member** list.

Every project or library has an object model that contains a number of member classes. Next to each item in the **Class** list, an icon depicts its class type:

Class icon	Type
	global value
	module
	enumerated type
	type
	class module

*Global values* (which apply to the selected project in its entirety) include individual members from *enumerated types* (such as text paragraph alignments, shape types, and import/export filters).

Member classes of an object have their own members.








To receive detailed information about a selected item, click the **Help** button at the top of the Object Browser.



## Using the Member list

The **Member** list shows all of the properties, methods, and events that are members of the current class. Each member is given an icon according to its type:

Class icon	Type
	property
	implied or default property
	method
	event
	constant

Property members may be simple types (such as Booleans, integers, or strings), or they may be a class or enumerated type from the **Class** list. A property that is based on a class from the **Class** list inherits all the members of that class.

Many classes have a default property, indicated by a blue dot in their icon. The default property is implied if no property name is given when getting or setting the value of the parent object.

Methods are commonly known as “member functions” — functions that the class can perform on itself.

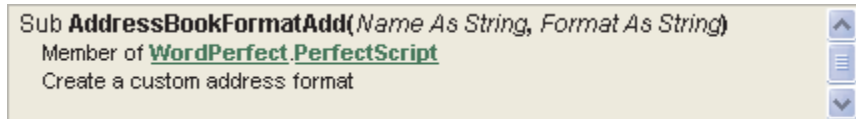
If the return value of a function is not used, the function call does not take parentheses around the argument list unless the `Call` keyword is used.

Some classes have various *events* associated with them. By setting up an *event handler* for a class’s event, when that event occurs in the application, the event’s handler is called. This functionality enables sophisticated applications to be developed that respond automatically to what is happening within the application.

The *constants* listed in the **Member** list are either members of enumerated types or defined as `Public` in a module. Enumerated types are used to group related items from a closed list.

## Using the Information window

The **Information** window gives information about the selected class or class member. This information includes a “prototype” of the member, its parent, and a short description of the member, and it also states whether the member is a read-only property.



*The Information window*

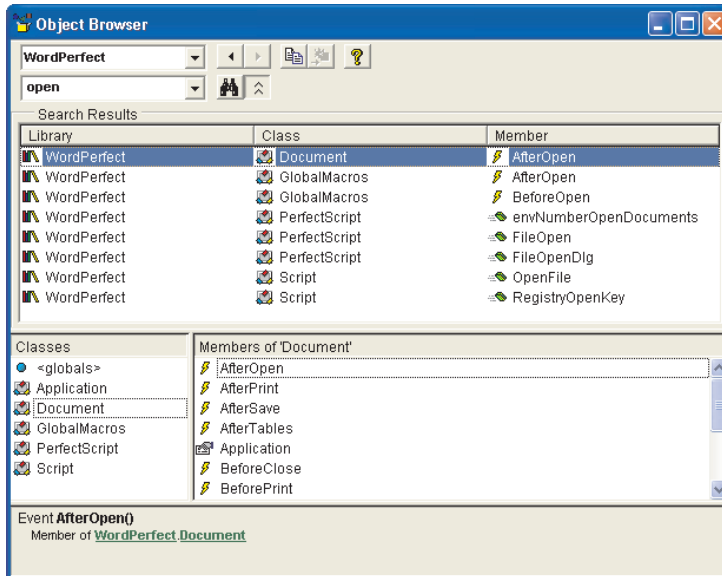
The types of any function parameters and properties are given as hyperlinks to the type definition or class itself, if the type is defined within the current object model.



To increase the height of the **Information** window, drag the top border of the window upwards to reveal its contents, or scroll down using the scrolls bar at the right of the window.

## Using the search controls

You can search the object model for a matching string. This is useful for finding a class or member whose name you can only partly remember, or for finding classes and members that have similar names (such as names based on, or containing, the word “open”).



*Searching an object model*

To search an object model's classes and members, type a string into the **Search** box, and then click the **Search** button. The **Search Results** list appears, displaying all of the found matches in alphabetical order. Clicking a found match advances the **Class** and **Member** lists to that item and displays its information in the **Information** window.



Matching class names have a blank **Member** column in the **Search Results** window.



To hide the **Search Results** window, click the **Hide search results** button







## Working with macros

---

Now that you know your way around the VBA workspace, you are ready to begin working with macros.

This chapter covers the following topics:

- Creating macros
- Running macros
- Debugging macros
- Using the debugging windows



For information on the WordPerfect Office X5 object models, see the Appendix: About the object models.

### Creating macros

WordPerfect, Quattro Pro, and Presentations let you create VBA macros. However, Corel WordPerfect Office uses the PerfectScript™ scripting language natively, and although PerfectScript is a separate editor from VBA, VBA in Corel WordPerfect Office must access the PerfectScript object to write a VBA macro.

Traditionally, there are two ways to create a macro: by writing it, or by recording it.

Detailed information on scripting VBA macros is beyond the scope of this guide, but some additional information (along with documentation on the applications' VBA commands) is provided in the VBA Help systems for WordPerfect, Quattro Pro, and Presentations.

You can record macros in WordPerfect, Quattro Pro, and Presentations. However, macros recorded from within Corel WordPerfect Office are not rendered as VBA code but instead as PerfectScript code; you cannot record VBA macros. For information on recording PerfectScript macros in WordPerfect, Quattro Pro, or Presentations, see the Help system for that application.



Recording PerfectScript macros can give you a glimpse into the logic of VBA. If you need help getting started with scripting VBA, you may want to start by recording a PerfectScript macro.

## Creating global macros

WordPerfect and Presentations let you create global macros, which can be used in all projects. When you write a global macro, you store it to the application's Global Macro Storage (GMS) file. The GMS files for WordPerfect and Presentations are **WordPerfect\*\*.gms** and **Presentations\*\*.gms**, respectively (where \*\* represents the version number). The VB Editor stores all of the modules for that project in the application's GMS file.



WordPerfect and Presentations do not support having more than one GMS file.

Quattro Pro does not support GMS files.

## Using modules

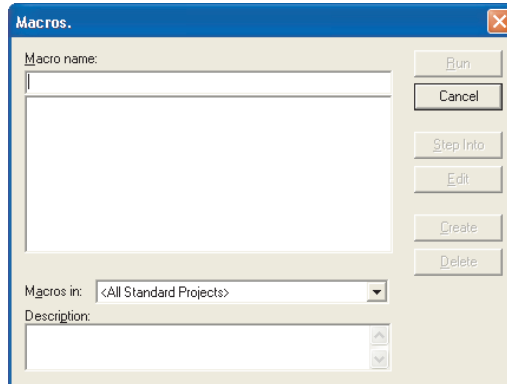
Each project that you create can contain several modules. The Project Explorer (see page 21) presents each module type in its own folder. You cannot move a module from one folder to another within the same project, but you can drag a module to another project to make a copy of it there. There are four types of modules:

- **WordPerfect Objects, QuattroPro Objects, or Presentations Objects** —used mostly for event handling, contains a single item (`ThisDocument`, `ThisNotebook`, and `ThisSlideShow`, respectively) and should not be used for normal code
- **forms** — used for custom dialog boxes and user interfaces, including the code to control them
- **modules** — used for general code and macros
- **class modules** — used for object-oriented Visual Basic classes (which are not discussed in this guide)

To write the macro, you can use the VB Editor. Macros that are developed in the VB Editor can take advantage of full programming control, including conditional execution, looping, and branching. In effect, you can write macros that are programs in their own right. (For the purposes of this guide, however, all VBA code is referred to as a macro even though, in some contexts, a macro is just those parts of that code that can be launched by WordPerfect, Quattro Pro, or Presentations.)

## Running macros


You can run macros either from directly within WordPerfect, Quattro Pro, or Presentations or from within the VB Editor.



*The WordPerfect Macros dialog box*

### To run a macro from within WordPerfect, Quattro Pro, or Presentations

- 1 Click Tools ► Visual Basic ► Play.

In WordPerfect and Quattro Pro, you can also click the **Play** button  on the **Visual Basic** toolbar.

- 2 From the **Macros in** list box, choose the VBA project in which the recorded macro is stored.
- 3 Select the macro in the **Macro name** list.
- 4 Click **Run**.

### To run a macro from within the VB Editor

- Click anywhere in the subroutine that forms the macro, and then click **Run** ► **Run macro**.

## Debugging macros

The VB Editor provides strong debugging facilities that are common to language editors. It is possible to set breakpoints and to step through code.



You can also make changes to the code while it is running and watch and change variables, but these are advanced techniques that are not discussed in this guide.

## Setting breakpoints

A breakpoint is a marker in a line of code that causes execution to pause. To continue, you must either restart the execution or step through the subsequent lines of code.

To set or clear a breakpoint, click the line and then click **Debug ▶ Toggle breakpoint**. By default, the line is highlighted in dark red and a red dot is placed in the margin. To clear all breakpoints, click **Debug ▶ Clear all breakpoints**.

To restart the code after it pauses at a breakpoint, click **Run ▶ Continue**. To pause the execution of the code (immediately exiting from all functions and discarding all return values), click **Run ▶ Reset**.

You can also “run to cursor” — that is, execute the code until it reaches the line that the cursor is on, and then pause at that line. To do this, click the line where you want execution to pause, and then click **Debug ▶ Run to cursor**.



If the line with the breakpoint (or the cursor when “running to cursor”) is not executed because it is in a conditional (if-then-else) block, the code does not stop at that line.

Breakpoints are not saved. They are lost when you close the VB Editor.

## Stepping through the code

When execution pauses at a breakpoint, you can continue through the code one line at a time. This lets you examine the values of individual variables after each line and determine how the code affects the values (and how the values affect the code). This is called “stepping through the code.”

To step through the code (one line at a time), click **Debug ▶ Step into**. The execution advances to every line in all called functions and subs.

To step through each line of the current function or sub but not through the lines of each called function or sub, click **Debug ▶ Step over**. The called functions and subs are executed, but not line-by-line.

To execute the rest of the current function or sub but pause when the function or sub returns to the point where it was called, click **Debug ▶ Step out**. This is a quick way



of returning to the point of entry of a function, so as to continue stepping through the calling function's code.

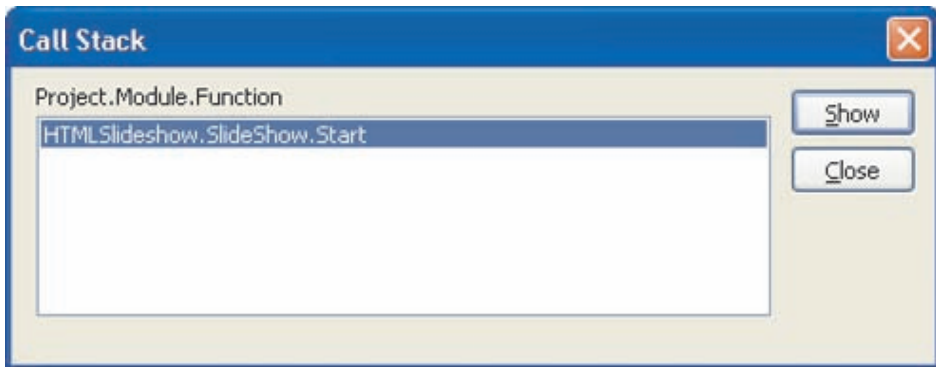
## Using the debugging windows

There are four windows that are used when debugging code: the **Call Stack** window, the **Immediate** window, the **Locals** window, and the **Watches** window. All of these windows provide important information about the state of functions and variables while an application is running.

### Using the Call Stack window

The **Call Stack** window is a modal dialog box that lists which function calls which function. In long, complicated applications, this is useful for tracing the steps to a particular function being called. To visit a function listed in the window, select the function name and then click **Show**, or else close the window.

To display the **Call Stack** window, click **View ▶ Call Stack**.



*The Call Stack window*

### Using the Immediate window

The **Immediate** window allows you to type in and run arbitrary lines of code while a macro is paused. This is useful for getting or setting the property of an object in the document, or for setting the value of a variable in the code. To run a piece of code, type it in the **Immediate** window, and then press **Enter**. The code is executed immediately.

To display the **Immediate** window, click **View ▶ Immediate window**.

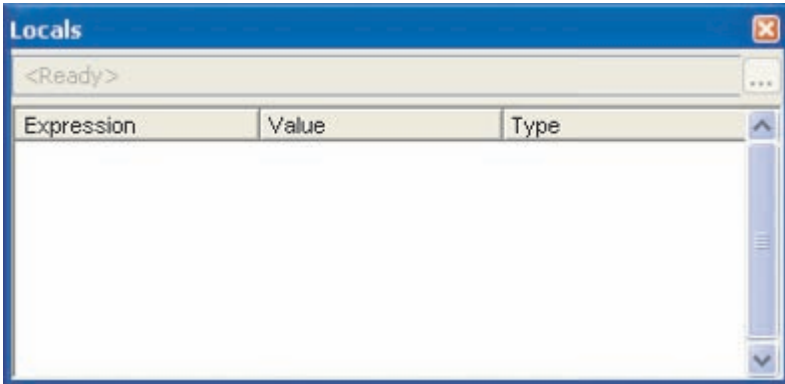


*The Immediate window*

### Using the Locals window

The **Locals** window displays all of the variables and objects that exist within the current scope. Each variable's type and value are listed in the columns next to the variable's name. Some variables and objects may have several children, which can be displayed by clicking the expand tree button next to the parent. Many variables let you edit their value by clicking it.

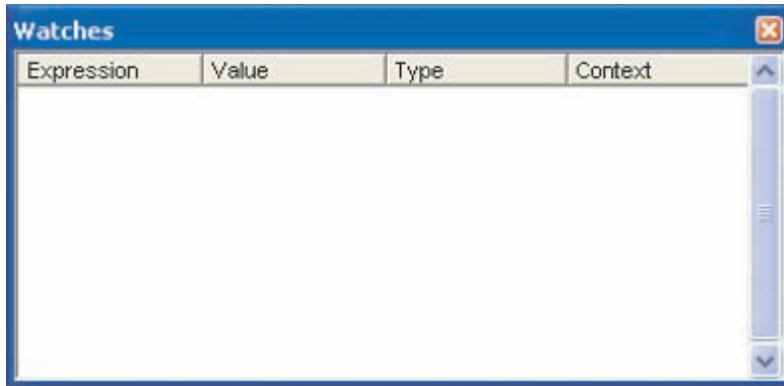
To display the **Locals** window, click **View** ► **Locals window**.



*The Locals window*

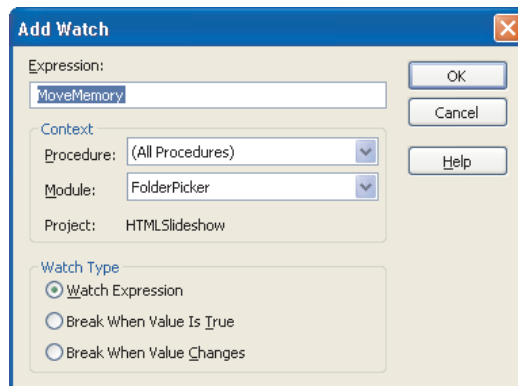
## Using the Watches window

The **Watches** window is used to watch specific variables or object properties. This is very useful for selecting just one or two values to watch as opposed to having to find the value you want in all the values in the **Locals** window.



*The Watches window*

To add a value to the **Watches** window, select the variable or object and its property, and then drag the selection onto the **Watches** window; alternatively, click the item, and then click **Debug** ► **Quick Watch** to add the item directly to the **Watches** window.



*The Add Watch dialog box*

Select the item you want to watch, select any conditions for this watch, and then click **OK**. If the condition becomes true, the application pauses to let you examine the code.





## Creating user-interfaces for macros

---

An important part of many VBA solutions is the user-interface. A well-designed user-interface makes the VBA solution so easy to use or so powerful that the user doesn't hesitate to use it.

Most user-interfaces for complex VBA solutions are based on a dialog box or form, but simpler user-interfaces can be created by using toolbars and buttons (which can, in turn, be enhanced with captions and tooltips, and images or icons).

However, regardless of the nature of a user-interface, its VBA solution can be made easier to deploy and support by providing the user with some form of help.

This chapter covers the following topics:

- Creating dialog boxes for macros
- Coding dialog boxes
- Designing dialog boxes
- Providing help for macros

### Creating dialog boxes for macros

All dialog boxes should abide by the following guidelines:

- They should have a meaningful title.
- They should provide an obvious functionality for cancelling or closing them.
- Their layout should make them easy to use, but they should also provide a **Help** button from which users can access how-to documentation.
- Their every control should feature a `ControlTipText` string, so that users can receive information about each control by passing the pointer over it.

However, there are two kinds dialog boxes: *modal* and *modeless*.

Modal dialog boxes must be acted upon before the user can resume the macro. The application is locked until the dialog box is dismissed (either by submitting it or by cancelling it). Built-in dialog boxes that you can control with VBA are almost always modal.

Modeless dialog boxes do not lock the application, so they can be left open while the user continues working in the application. In this way, they behave like dockers.

Before you can code and design dialog box, you must decide whether to make it modal or modeless.

### **Choosing between modal and modeless dialog boxes**

The kind of dialog box you should provide depends on what you want to achieve.

Modal dialog boxes usually have the following features:

- an **OK** button — performs the dialog box's ultimate action and then hides the dialog box. It is the default button.
- a **Cancel** button — dismisses the dialog box without performing the dialog box's action. The **Close** button in the upper-right corner of the dialog box provides the same functionality.

Some modal dialog boxes require an **Apply** button that performs the dialog box's action without making it permanent, such that cancelling the dialog box undoes the action.

If the dialog box is in the style of a wizard, it should have a **Previous** button and a **Next** button, as well as a **Cancel** button. On the first page of the sequence, the **Previous** button should be disabled (that is, have its `Enabled` property set to `False`), while on the last page, the **Next** button should become the **Finish** button to indicate that the final page has been reached.

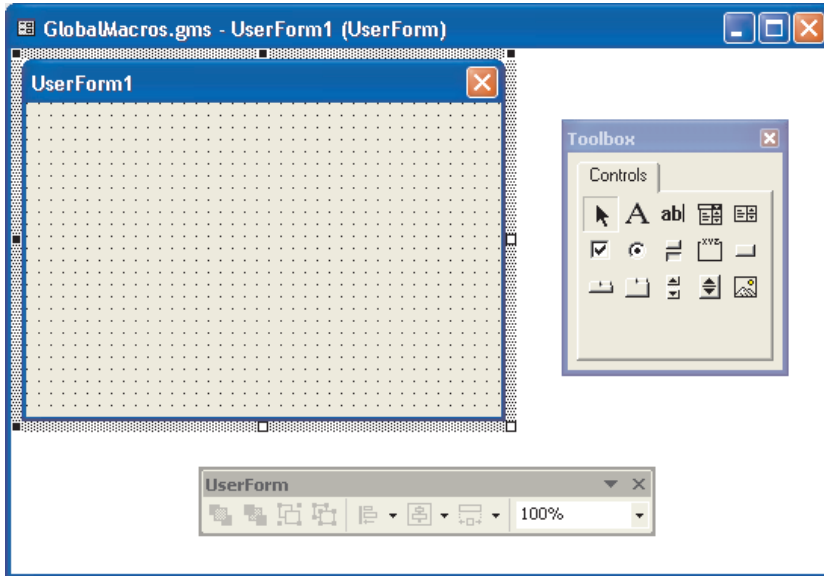
Modeless dialog boxes usually have the following features:

- an **Apply** or **Create** button — performs the dialog box's action and can, in fact, be specially labelled to describe the dialog box's action. This button should be the default.
- a **Close** button — closes the dialog box. This button is used after the user has applied the desired action.

After you have chosen whether your dialog box should be modal or modeless, you are ready to start setting it up.

### **Setting up dialog boxes**

To set up a customized dialog box for use in your VBA solution, you use the Form Designer in the VB Editor. The Form Designer provides easy access to the tools for coding and designing a form.



*A blank form in the Form Designer*

You can access the Form Designer by creating a new, blank form.



For information on accessing the **User Form** toolbar, which you can use when designing a form, see “Using the toolbars in the VB Editor” on page 26.

You can test a form at any time by running it.

### To create a new, blank form

- In the Project Explorer, right-click the project to which you want to add a dialog box, and then click **Insert ► UserForm**.



To change the title of the form, click the form to select it, and then in the **Properties** window, change the **Caption** property.

It is highly recommended that you give each form a unique, descriptive name. You can do this from the **Properties** window, but remember to follow the rules for naming variables in VBA.

## To test a form by running it

- Press F5.

## Coding dialog boxes

Custom dialog boxes can be set as either modal or modeless by using the `Modal` parameter of the form's `Show` method.

For example, the form `frmFooForm` can be displayed with the following code:

```
frmFooForm.Show
```

Because the optional parameter `Modal` has, by default, the value `vbModal` (or 1), this code creates a modal dialog box.

If the `Modal` parameter is set to `vbModeless` (or 0), as in the following example, then a modeless dialog box is created:

```
frmFooForm.Show vbModeless
```

To open a dialog box from a macro that is available from within the application itself, you must create a public sub within a VBA module; if a sub exists within a form's code or within a class module, it cannot be made available from within the application. The sub you create cannot take any parameters.

The following example sub would launch `frmFooForm` as a modeless form:

```
Public Sub showFooForm()  
    frmFooForm.Show vbModeless  
End Sub
```



When a form loads, it triggers its own `UserForm_Initialize` event. From this event handler, you should initialize all the controls on the form that must be initialized.

It is possible to launch one or more other forms from within the current form by using its **Show member** function:

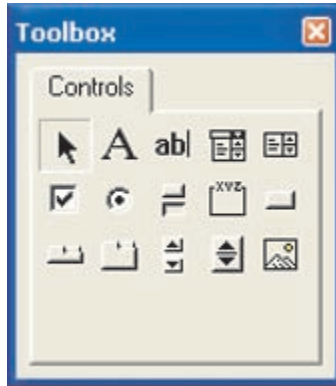
```
UserForm2.Show vbModal
```

However, VBA does not return control until all open forms have been unloaded.








## Designing dialog boxes










The Form Designer toolbox is the main utility you'll use when designing dialog boxes. It lets you add controls to a form by dragging the appropriate control from the toolbox to the form.



*Form Designer toolbox*


The Form Designer toolbox lets you add the following controls to a form:

Icon	Control	Function
	Label	Lets you provide the user with static text (for example, instructions or captions)
	TextBox	Lets you provide an area into which the user can type text. For more information, see “Providing text boxes” on page 46.
	ComboBox	Lets you provide a list from which the user can select an item and (optionally) into which the user can also type text. For more information, see “Providing combination boxes and list boxes” on page 47.
	ListBox	Lets you provide a list from which the user can select multiple items. For more information, see “Providing combination boxes and list boxes” on page 47.
	CheckBox	Lets you provide a check box that the user can enable (by clicking to insert a check mark into it) or disable (by clicking to remove the check mark from it), or that can be grayed (so as to make it unavailable to the user)

Icon	Control	Function
	OptionButton	Lets you provide a “radio button” that is linked to other radio buttons with the same GroupName property, such that the user can enable only one of them at a time
	ToggleButton	Lets you provide a button that the user can click to toggle (such that it does or does not appear pressed)
	Frame	Lets you group items together: items drawn on the frame move with the frame
	CommandButton	Lets you provide a button that the user can click to commit an assigned action. For more information, see “Providing buttons” on page 47.
	TabStrip	Lets you provide the user with separate views of related controls
	MultiPage	Lets you provide the user with multiple pages of controls
	ScrollBar	Lets you provide the user with immediate access to a range of values by scrolling
	SpinButton	Lets you enhance another control (such as a TextBox) so that the user can more quickly set that control’s value
	Image	Lets you provide an image. For more information, see “Providing images” on page 50.

More detailed information follows for some of these controls.



The Form Designer toolbox also features a Pick tool , which lets you select and move the controls on a form.



For more information about any one of these controls (or about the other controls supported by VBA), draw the control in a form and then press **F1** to display a Help topic for it.

## Providing text boxes

Text boxes (that is, `TextBox` controls) are the mainstay of user input. They are simple to use, quick to program, and very flexible.

To set the text in a text box when initializing it, set the `TextBox` control's `Text` (default or implicit) property:

```
txtWidth.Text = "3"  
txtHeight = "1"
```

To get the value of a `TextBox` control, get its `Text` property:

```
Call SetSize(txtWidth.Text, txtHeight.Text)
```

## Providing combination boxes and list boxes

In a combination box (that is, a `ComboBox` control), the user can either choose an item from the list or type into the text box. You can prevent users from being able to type into a `ComboBox` control by setting its `Style` property to `fmStyleDropDownList`.

In a list box (that is, a `ListBox` control), the user can choose one or more items (from, typically, between three and ten items).

To populate a list of any type, you must call the list's member function `AddItem`. This function takes two parameters: the string or numerical value, and the position in the list. The position parameter is optional, so omitting it inserts the item at the last position in the list. For example, the following code populates the list `ComboBox1` with four items:

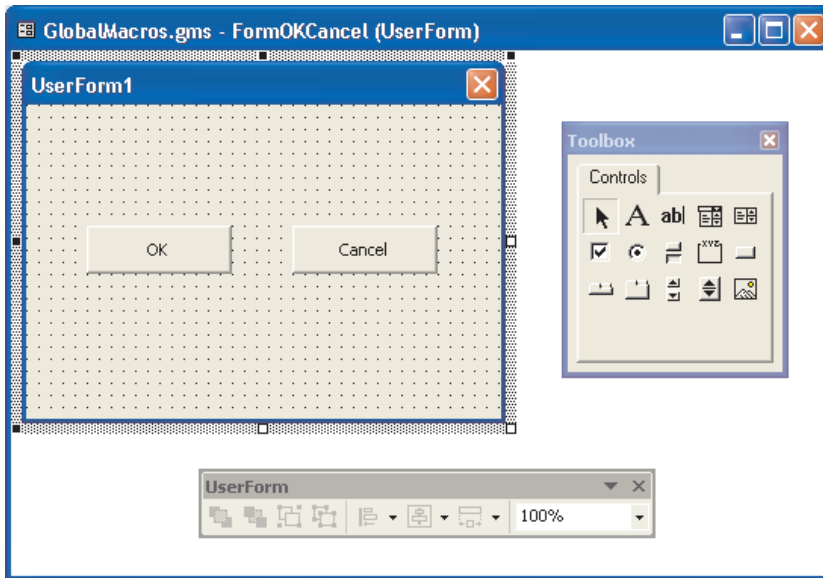
```
ComboBox1.AddItem 1  
ComboBox1.AddItem 2  
ComboBox1.AddItem 3  
ComboBox1.AddItem 0, 0
```

To test which item is selected when the **OK** button is clicked, test the list's `ListIndex` property. To get the value of a selected item's caption, test the `Text` property for the `ComboBox` or `ListBox`:

```
Dim retList As String  
retList = ComboBox1.Text
```

## Providing buttons

As previously discussed, you can add a button to a form by using the `CommandButton` control. Click the form to add a default-sized button, or drag to create one to your own specifications. Click the caption to edit it, or select the button and edit its `Caption` property in the **Properties** window. You might also want to change the name of the button to something more descriptive, such as `buttonOK` or `buttonCancel`.



*Designing buttons in the Form Designer*

Most forms have an **OK** button and a **Cancel** button. However, no button functions until its form has code for handling the button's click event. (This is because VBA forms are event-driven.)

The **Cancel** button is the simplest control: it must dismiss the form without doing anything else. To add a cancel action to a **Cancel** button, double-click the button from within the Form Designer to display its code in the **Code** window. This creates a new subroutine called `cmdCancel_Click`:



*The Code window with code for a Cancel button*

The following code, if applied to a **Cancel** button, instructs the form to be dismissed when the button is clicked:

```
Private Sub cmdCancel_Click()  
    Unload Me  
End Sub
```

If you continue by setting the form's **Cancel** property to **True**, you'll find that when the user presses **Escape**, the `cmdCancel_Click` event is triggered and the code you've provided unloads the form.

Similarly, you can select the **OK** button and set its **Default** property to **True**, so that when the user presses **Enter** to activate the form, the **OK** button's event handler is called; the **OK** button's click-event handler performs the form's functionality and then unloads the form.

If the form is used to set the size of the selected shapes by setting their width and height, then the **OK** button's click-event handler could resemble the following code sample (which assumes you have already created two text boxes called `txtWidth` and `txtHeight`):

```
Private Sub buttonOK_Click()
```

```
Me.Hide
Call SetSize(txtWidth.Text, txtHeight.Text)
Unload Me
End Sub
```

From inside the form's own code module, the form object is implicit, and so all the controls can be simply accessed by name. From other modules, the controls must be accessed via their full name, which would be `UserForm1.buttonOK`.

## Providing images

The `Image` control is used to place graphics on the form. The image (a bitmap) is contained in the `Picture` property, so you can either load an RGB image from a file (such as a GIF, JPEG, or Windows Bitmap BMP file) or paste one into the property.

At run time, new images can be loaded into the `Image` control by changing the `Picture` property — by using the VBA function `LoadPicture` and providing a path to the new image file — as in the following example:

```
Image1.Picture = LoadPicture("C:\Images\NewImage.gif")
```

## Providing help for macros

It is highly recommended that you provide users with documentation for your macros.

One solution is to provide them with a `Readme` file or a printed manual. Another option is to build instructions directly into the dialog box, but this uses up valuable on-screen “real estate”. Yet another alternative is to create an online help system, but this requires tools and a fair amount of additional work.

For these reasons, you may instead want to provide help in the form of plain-text file. If you create a registry value when the project is installed that points to the location of the file, you can use the following function to open it:

```
Public Sub launchNotepad(file As String)
    Shell "Notepad.exe" & " " & file, vbNormalFocus
End Sub
```

Pass the full path to the text file, such as `C:\ReadMe.txt`, in the parameter `file`.

A much more powerful — but still easily created — solution is to use `HTML`. By using `HTML`, you can include graphics in your documentation, and you can also direct the user to a specific location on the page by using “hash” references (such as

index.html#middle). If you know where the HTML file is installed, you can use the following function to open it:

```
' Put this Declare statement before all Subs and Functions!
Declare Function ShellExecute Lib "shell32.dll" _
    Alias "ShellExecuteA" (ByVal hwnd As Long, _
    ByVal lpOperation As String, ByVal lpFile As String, _
    ByVal lpParameters As String, ByVal lpDirectory As String, _
    ByVal nShowCmd As Long) As Long
Public Sub launchBrowser(url As String)
    ShellExecute 0, vbNullString, url, vbNullString, vbNullString, 5
End Sub
```

Simply pass the file name (for example, C:\Program Files\ReadMe.htm) or URL in the parameter url.







## Organizing and deploying macros

---

When you've finished designing your macro, you can make it available to other users.

This chapter covers the following topics:

- Organizing macros
- Deploying macros

### Organizing macros

To make your macros easy to deploy, it's a good idea to organize them.

The best way to organize and maintain your macros is to use a separate module for each macro, and then group related macros into a single project file.

To help the user find the entry point to your macros, it's a good idea to place all of the public subs into a single module and then instruct the user how to find them; that way, the macros can be called from within WordPerfect, Quattro Pro, or Presentations.

### Deploying macros

You can deploy macros to users for installation by deploying project files.



For more information on deploying macros across an organization, network administrators can refer to the *Corel WordPerfect Office Deployment Guide* that is included with their purchase of a license SKU.

### Deploying project files

You can explicitly distribute macros as part of a document so that when that document is opened, the user has immediate access to the macros even if they have not installed them. In this way, you can, for example, set up a macro to track how much time has been spent editing the document.

You must devise a mechanism for distributing your project files to users for installation. While it is possible to distribute modules on their own, it is much simpler for your users

if you distribute project files instead (so that the users do not have to manually integrate the module into their existing project file).

### **To deploy a project file**

- 1 Make the project file available to the users.
- 2 Instruct the users to install the project file.



## Appendix: About the object models

---

This appendix contains information on the object models of WordPerfect, Quattro Pro, and Presentations.

### About the PerfectScript and Script classes

The object models for WordPerfect, Quattro Pro, and Presentations contain a class called **PerfectScript**, the methods for which correspond to the PerfectScript product commands for WordPerfect, Quattro Pro, and Presentations (respectively).

The object models for WordPerfect and Presentations also contain a class called **Script**, the methods for which correspond to the PerfectScript programming commands.

For guidance on using the methods of the **PerfectScript** and **Script** classes, please refer to the macro-command documentation in the PerfectScript Help (**psh.chm**).

### About the WordPerfect object model

The WordPerfect object model breaks down into five main classes:

- **Application** — contains properties that govern the application
- **Document** — contains properties, methods, and events that govern documents
- **GlobalMacros** — contains properties, methods, and events that are used for macro storage
- **PerfectScript** — contains methods that correspond to the PerfectScript product commands for WordPerfect, which are documented in the “WordPerfect Command Reference” section of the PerfectScript Help (**psh.chm**).
- **Script** — contains methods that correspond to the PerfectScript programming commands, which are documented in the “PerfectScript Command Reference” section of the PerfectScript Help (**psh.chm**).

For information specific to using VBA in WordPerfect, please see the main WordPerfect Help (**wpwp.chm**). The topic “Using the PerfectScript class to change WordPerfect documents” is particularly useful for creating a VBA macro that uses the **PerfectScript** class.

## About the Quattro Pro object model

The Quattro Pro object model breaks down into three main classes:

- **Application** — contains properties that govern the application
- **Document** — contains properties and events that govern documents
- **PerfectScript** — contains methods that correspond to the PerfectScript product commands for Quattro Pro, which are documented in the “Quattro Pro Command Reference” section of the PerfectScript Help ([psh.chm](#)).

For information specific to using VBA in Quattro Pro, please see the main Quattro Pro Help ([qp.chm](#)). The topic “Using the PerfectScript class to change a Quattro Pro document” is particularly useful, as are the VBA examples in the “Reference: Using macros” section.

## About the Presentations object model

The Presentations object model breaks down into five main classes:

- **Application** — contains properties that govern the application
- **Document** — contains properties, methods, and events that govern documents
- **GlobalMacros** — contains properties, methods, and events that are used for macro storage
- **PerfectScript** — contains methods that correspond to the PerfectScript product commands for Presentations, which are documented in the “Presentations Command Reference” section of the PerfectScript Help ([psh.chm](#)).
- **Script** — contains methods that correspond to the PerfectScript programming commands, which are documented in the “PerfectScript Command Reference” section of the PerfectScript Help ([psh.chm](#)).

For information specific to using VBA in Presentations, please see the topic “Working with VBA macros” in the main Presentations Help ([wppr.chm](#)). The subtopic “Using the PerfectScript class to change slide shows” is particularly useful.



# Glossary

---

## **array**

A set of sequentially indexed elements of the same data type. By default, array indexes are zero-based.

## **automation**

The process of recording or scripting a macro

## **class**

The definition (that is, description) of an object

## **class module**

A module that contains the definition of a class, including its property and method definitions

## **collection**

A set of objects

## **constant**

A named item that keeps a constant value while a macro is being executed

## **enumerated type (enumeration)**

A data type that lists all the possible values for the variables that use it

## **event**

An action that is recognized by a form or control

## **event handler**

A subroutine that is programmed to cause the application to respond to a specific event

## **function**

A procedure that performs a given task in a macro and that can be used to return a value. A function procedure begins with a `Function` statement and ends with an `End`

Function statement. In VBA, functions do not need to be declared before they are used, nor before they are defined.

### **global value**

A value that applies to a given project in its entirety

### **macro**

A scripted or recorded set of actions that can be repeatedly invoked within an application

### **method**

An operation that an object can have performed on itself

### **modal dialog box**

A dialog box that must be acted upon before the user can resume the macro. The application is locked until the dialog box is dismissed, either by submitting it or by cancelling it. Built-in dialog boxes that you can control with VBA are almost always modal.

### **modeless dialog box**

A dialog box that does not lock the application, such that it can be left open while the user continues working in the application. In this way, modeless dialog boxes behave like dockers.

### **module**

A set of declarations followed by procedures

### **object**

An instance of a class. An object can be a parent to child objects.

### **object model**

A high-level structure of the relationship between parent and child objects. Without an object model, VBA cannot gain access to the objects in the document, nor query or change an application's documents.

### **passing by reference**

The act of passing an argument to a function or subroutine by using a reference to the original. By default, function and subroutine parameters are passed by reference, but if

you want to explicitly annotate the code to indicate that an argument is being passed by reference, you can prefix the argument with `ByRef`.

### **passing by value**

The act of passing an argument to a function or subroutine by using a copy of the original. To indicate that you want to pass an argument by value, prefix the argument with `ByVal`.

### **property**

A characteristic of a class. Properties that are fixed by the design of the class are called “read-only.”

### **scope**

The visibility of a data type, procedure, or object

### **shortcut object**

A syntactical replacement for the long-hand version of an object

### **subroutine (sub)**

A procedure that performs a given task in a macro but that cannot be used to return a value. A subroutine procedure begins with a `Sub` statement and ends with an `End Sub` statement. In VBA, subroutines do not need to be declared before they are used, nor before they are defined.

### **variable**

An item that can be created (or “declared”) for the purposes of storing data. The built-in data types are `Byte`, `Boolean`, `Integer`, `Long`, `Single`, `Double`, `String`, `Variant`, and several other less-used types including `Date`, `Decimal`, and `Object`. If a variable is not declared before it is used, the compiler interprets it as a `Variant`.

## **Visual Basic for Applications (VBA)**

A built-in programming language that can automate repetitive functions and create intelligent solutions in `WordPerfect`, `Quattro Pro`, and `Presentations`





# Index

---

## A

- arrays, declaring . . . . . 12
- automatic completion . . . . . 25
- automation . . . . . 6

## B

- bitwise operators . . . . . 16
- Boolean comparison and assignment . . . 15
- breakpoints . . . . . 36
- buttons . . . . . 47

## C

- C and C++ . . . . . 8
- Call Stack window . . . . . 37
- checking syntax automatically . . . . . 24
- classes . . . . . 8
- code
  - formatting automatically . . . . . 23
  - stepping through . . . . . 36
- Code window . . . . . 23
- coding dialog boxes . . . . . 44
- coloring syntax automatically . . . . . 24
- combination boxes . . . . . 47
- comments . . . . . 14
- contextual pop-up lists . . . . . 25

## D

- debugging macros . . . . . 35
  - setting breakpoints . . . . . 36
  - stepping through code . . . . . 36
  - using the windows . . . . . 37

## declaring

- arrays . . . . . 12
- enumerated types . . . . . 13
- strings . . . . . 12
- variables . . . . . 11

## defining scope . . . . . 15

## definitions, jumping to . . . . . 25

## deploying

- GMS files . . . . . 54
- macros . . . . . 53
- project files . . . . . 53

## designing dialog boxes . . . . . 45

## dialog boxes

- coding . . . . . 44
- creating for macros . . . . . 41
- designing . . . . . 45
- modal vs. modeless . . . . . 42
- setting up . . . . . 42

## docking toolbars . . . . . 26

## documentation conventions . . . . . 2

## E

- ending lines . . . . . 13
- enumerated types, declaring . . . . . 13
- enumerations
  - See* enumerated types

## F

- floating toolbars . . . . . 26
- Form Designer . . . . . 45
- formatting code automatically . . . . . 23
- forms
  - buttons . . . . . 47
  - combination boxes . . . . . 47

---

creating . . . . .	43	debugging . . . . .	35
images . . . . .	50	deploying . . . . .	53
list boxes . . . . .	47	global . . . . .	34
testing . . . . .	44	organizing . . . . .	53
text boxes . . . . .	46	providing help for . . . . .	50
functions, building . . . . .	12	running . . . . .	35
<b>G</b>		memory allocation . . . . .	14
global macros . . . . .	34	memory pointers . . . . .	14
GMS files, deploying . . . . .	54	message boxes . . . . .	17
<b>H</b>		methods . . . . .	9
help, providing for macros . . . . .	50	modal dialog boxes . . . . .	41
<b>I</b>		modeless dialog boxes . . . . .	41
images, providing in forms . . . . .	50	modules . . . . .	34
Immediate window . . . . .	37	<b>N</b>	
including comments . . . . .	14	non-programmers . . . . .	6
input boxes . . . . .	17	<b>O</b>	
<b>J</b>		Object Browser . . . . .	27
Java and JavaScript . . . . .	7	Class list . . . . .	28
jumping to definitions . . . . .	25	Information window . . . . .	30
<b>L</b>		Member list . . . . .	29
lines, ending . . . . .	13	search controls . . . . .	30
list boxes . . . . .	47	object model . . . . .	9
Locals window . . . . .	38	Presentations . . . . .	56
logical operators . . . . .	16	Quattro Pro . . . . .	56
<b>M</b>		WordPerfect . . . . .	55
macros		objects . . . . .	9
creating . . . . .	33	hierarchy . . . . .	10
creating dialog boxes for . . . . .	41	parent/child relationship . . . . .	9

---

---

operators . . . . . 15, 16  
organizing macros . . . . . 53

## **P**

passing by reference . . . . . 14  
passing by value . . . . . 14  
pop-up lists, contextual . . . . . 25  
Presentations object model . . . . . 56  
programmers . . . . . 7  
programming languages . . . . . 7  
Project Explorer . . . . . 21  
project files, deploying . . . . . 53  
properties . . . . . 9  
Properties window . . . . . 22

## **Q**

Quattro Pro object model . . . . . 56

## **R**

running macros . . . . . 35

## **S**

scope, defining . . . . . 15  
setting breakpoints . . . . . 36  
setting up dialog boxes . . . . . 42  
shortcut objects . . . . . 10  
starting the VB Editor . . . . . 21  
stepping through code . . . . . 36  
strings . . . . . 12  
subroutines, building . . . . . 12  
subs  
    *See* subroutines

## **syntax**

checking automatically . . . . . 24  
coloring automatically . . . . . 24

## **T**

testing forms . . . . . 44  
text boxes . . . . . 46

## **toolbars**

docking . . . . . 26  
floating . . . . . 26  
VB Editor . . . . . 26  
VBA . . . . . 19

## **V**

variables, declaring . . . . . 11  
VB Editor . . . . . 20  
    Code window . . . . . 23  
    debugging windows . . . . . 37  
    Form Designer . . . . . 45  
    Object Browser . . . . . 27  
    Project Explorer . . . . . 21  
    Properties window . . . . . 22  
    starting . . . . . 21  
    toolbars . . . . . 26

VBA . . . . . 5  
    code structure . . . . . 10  
    compared with C and C++ . . . . . 8  
    compared with Java and JavaScript . . . . . 7  
    compared with Windows Script Host . . . . . 8  
    for non-programmers . . . . . 6  
    for programmers . . . . . 7  
    main elements of . . . . . 8  
    operators . . . . . 15, 16  
    toolbars . . . . . 19

## **Visual Basic for Applications**

*See* VBA

---

**W**

Watches window .....	39
Windows Script Host .....	8
WordPerfect object model .....	55

Copyright 2010 Corel Corporation. All rights reserved.

Corel® WordPerfect® Office X5 User Guide for VBA

Product specifications, pricing, packaging, technical support and information (“specifications”) refer to the retail English version only. The specifications for all other versions (including other language versions) may vary.

INFORMATION IS PROVIDED BY COREL ON AN “AS IS” BASIS, WITHOUT ANY OTHER WARRANTIES OR CONDITIONS, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THOSE ARISING BY LAW, STATUTE, USAGE OF TRADE, COURSE OF DEALING OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS OF THE INFORMATION PROVIDED OR ITS USE IS ASSUMED BY YOU. COREL SHALL HAVE NO LIABILITY TO YOU OR ANY OTHER PERSON OR ENTITY FOR ANY INDIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING, BUT NOT LIMITED TO, LOSS OF REVENUE OR PROFIT, LOST OR DAMAGED DATA OR OTHER COMMERCIAL OR ECONOMIC LOSS, EVEN IF COREL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR THEY ARE FORESEEABLE. COREL IS ALSO NOT LIABLE FOR ANY CLAIMS MADE BY ANY THIRD PARTY. COREL’S MAXIMUM AGGREGATE LIABILITY TO YOU SHALL NOT EXCEED THE COSTS PAID BY YOU TO PURCHASE THE MATERIALS. SOME STATES/COUNTRIES DO NOT ALLOW EXCLUSIONS OR LIMITATIONS OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

Corel, the Corel logo, Corel DESIGNER, CorelDRAW, Digital Studio, Painter, PaintShop Photo, PerfectScript, Presentations, Quattro Pro, VideoStudio, WinDVD, WinZip, and WordPerfect are trademarks or registered trademarks of Corel Corporation and/or its subsidiaries in Canada, the U.S., and/or other countries. All other product names and any registered and unregistered trademarks mentioned are used for identification purposes only and remain the exclusive property of their respective owners.

103030

