



CorelDRAW[®] X3

GRAPHICS SUITE

PROGRAMMING GUIDE FOR VBA



Table of contents

Introduction	1
About CorelDRAW Graphics Suite X3	1
About VBA in CorelDRAW Graphics Suite X3	1
About this manual	2
About Corel Corporation	3
 Chapter 1	
Understanding VBA	4
What is VBA?	4
What is automation?	4
Who is VBA designed for?	5
How does VBA compare with other programming languages?	6
What are the main elements used in VBA?	7
What is an object model?	7
How is VBA code structured?	8
Declaring variables	9
Building functions and subroutines	10
Ending lines	11
Including comments	11
Using memory pointers and memory allocation	12
Defining scope	12
Using Boolean comparison and assignment	13
Using logical and bitwise operators	13
Providing message boxes and input boxes	14
 Chapter 2	
Getting started with VBA	15
Installing VBA for CorelDRAW Graphics Suite X3	15
Using the VBA toolbars	16
Using the VBA toolbar in CorelDRAW	16
Using the VB Editor toolbar in Corel PHOTO-PAINT	16
Using the VB Editor	17
Using the Project Explorer	17
Using the Properties window	18
Using the Code window	19
Using the toolbars	22
Using the Object Browser	23



Chapter 3	
Working with macros	27
Creating macros	27
Writing macros	27
Recording macros	28
Referencing objects in macros.	30
Referencing collections in macros	30
Using shortcut objects	32
Providing event handlers in macros	33
Running macros	34
Debugging macros	35
Using the debugging windows	36
Chapter 4	
Creating user-interfaces for macros	39
Creating dialog boxes for macros	39
Coding dialog boxes	41
Designing dialog boxes	42
Creating toolbars and buttons for macros	46
Creating toolbars for macros	47
Adding captions and tooltips to macros	47
Associating images or icons with macros	48
Providing user interaction for macros	48
Capturing mouse actions	48
Capturing coordinates	50
Providing help for macros	51
Chapter 5	
Organizing and deploying macros	52
Organizing macros	52
Deploying macros	52
Deploying project files	52
Deploying workspaces	53
Appendix	
Understanding the CorelDRAW object model	55
Working with documents	55
Creating documents	57
Opening documents	58
Importing files into documents	58
Switching between documents	58
Viewing documents	59
Changing content in documents	61
Setting the Undo string for documents	61
Exporting files from documents	61
Printing documents	63
Publishing documents to PDF	64
Closing documents	65



Working with pages	65
Creating pages	65
Switching between pages	66
Reordering pages	67
Resizing pages	67
Deleting pages	69
Working with layers	69
Creating layers	69
Activating layers	69
Locking and hiding layers	70
Reordering layers	70
Renaming layers	70
Importing files into layers	70
Deleting layers	71
Working with shapes	71
Creating shapes	72
Selecting shapes	77
Determining shape type	79
Changing shape properties	80
Coloring shapes	84
Duplicating shapes	87
Applying effects to shapes	87
Glossary.	89
Index	92





Introduction

Welcome to the *CorelDRAW® Graphics Suite X3 Programming Guide for VBA*, your resource for developing and distributing Microsoft® Visual Basic® for Applications (VBA) solutions in CorelDRAW Graphics Suite X3.

About CorelDRAW Graphics Suite X3

CorelDRAW Graphics Suite X3 simplifies the design process for projects of any scale, from logo creation and Web graphics to multipage marketing brochures and eye-catching signs. With powerful bitmap-to-vector tracing, helpful new learning tools, and enhanced illustration, page layout, and photo editing features, CorelDRAW Graphics Suite X3 delivers a combination of superior design capabilities, speed, ease of use, and affordability that's unmatched in the graphics software industry.

About VBA in CorelDRAW Graphics Suite X3

In 1995, Corel incorporated automation into CorelDRAW 6 by including its Corel SCRIPT™ language. This enabled solution developers to create intelligent mini-applications within CorelDRAW, such as ones that draw shapes, reposition and resize shapes, open and close documents, or set styles.

Corel SCRIPT was included with CorelDRAW versions 6 through 9. Although the Corel SCRIPT editor is not included with CorelDRAW in versions 9 and later, the run-time engine is included, so scripts written for earlier versions of CorelDRAW can easily be migrated to the latest versions.

In 1998, Corel took the strategic decision to augment the Corel SCRIPT functionality of CorelDRAW 9 by licensing the Microsoft Visual Basic for Applications engine to handle its behind-the-scenes automation. The addition of VBA made CorelDRAW immediately accessible to millions of VBA and Visual Basic developers around the world.

VBA in CorelDRAW can be used as a platform for developing powerful corporate graphical solutions, such as automated ticket generators, customized calendars, or batch-processors of files. VBA can also be used to enhance and optimize the workflow within CorelDRAW. For example, you can improve and customize some of the built-in functionality of CorelDRAW (such as object creation, alignment, or transformation) or add page layouts on-the-fly (such as to create company letterheads).

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.

For CorelDRAW Graphics Suite X3, VBA is available to both CorelDRAW and Corel PHOTO-PAINT®.



About this manual

This manual, which should be read in conjunction with the CorelDRAW and Corel PHOTO-PAINT object model diagrams that install with CorelDRAW Graphics Suite X3, was designed as a resource for the following:

- exploring the VBA IDE and many of its advanced features
- understanding the most important CorelDRAW Graphics Suite X3 functions and how to use them
- examining how to package and deploy VBA solutions developed for CorelDRAW Graphics Suite X3

This manual should be used by anyone who is interested in automating simple and complex tasks in CorelDRAW Graphics Suite X3 or who is developing commercial solutions that integrate with CorelDRAW Graphics Suite X3. It is assumed that the reader already has experience with at least one other procedural programming language, such as BASIC, Visual Basic, C, C++, Java™, Pascal, Cobol, or Fortran. This manual 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 CorelDRAW Graphics Suite X3 solutions.

This manual is organized into the following chapters, which deal with specific aspects of automating tasks and building solutions in CorelDRAW Graphics Suite X3:

- **Chapter 1: Understanding VBA** provides a brief introduction to VBA.
- **Chapter 2: Getting started with VBA** lets you explore the VBA workspace in CorelDRAW and Corel PHOTO-PAINT.
- **Chapter 3: Working with macros** shows you how to create, run, and debug macros.
- **Chapter 4: Creating user-interfaces for macros** demonstrates how to provide dialog boxes, toolbars and buttons, user interaction, and help for your macros.
- **Chapter 5: Organizing and deploying macros** helps you organize and deploy the macros you create.

This manual also provides an appendix (page 55), which provides detailed information on the CorelDRAW object model.

Finally, a glossary (page 89) defines the terms used in this manual.

Documentation conventions

The following conventions are used in this manual:

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 manual 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.



For more information

VBA for CorelDRAW Graphics Suite X3 also installs with the following documentation:

- VBA object model diagrams for CorelDRAW and Corel PHOTO-PAINT — These are available as Adobe® Acrobat® Reader® (PDF) files and are installed to `..\Program Files\Corel\CorelDRAW Graphics Suite 13\Programs`.
- VBA Help files for CorelDRAW and Corel PHOTO-PAINT — These can be accessed through the VB Editor: select any item in the Object Browser and press **F1** to access a Help topic on that item.

For comprehensive information about the features in CorelDRAW Graphics Suite X3, you can consult the user guide, which is installed by default to `..\Program Files\Corel\CorelDRAW Graphics Suite 13\Languages\<language>\Help\CorelDRAW Graphics Suite X3.pdf`. Alternatively, you can access a Help system from within a CorelDRAW Graphics Suite X3 application by clicking **Help ► Help topics**.

For the latest news, tips and tricks, and product upgrade information, you can check the CorelDRAW Graphics Suite X3 Web site. Go to www.corel.com, and follow the links for CorelDRAW Graphics Suite X3.

For prompt and accurate information about product features, specifications, pricing, availability, services, and technical support, contact Corel® Support Services™. For the most current information on available support and professional services for your Corel product, please visit www.corel.com/support.

If you have any comments or suggestions about CorelDRAW Graphics Suite X3 or about the *CorelDRAW Graphics Suite X3 Programming Guide for VBA*, you can submit them by using the contact information provided at www.corel.com/contact.

About Corel Corporation

Corel Corporation provides innovative software solutions that help millions of value-conscious businesses and consumers in more than 75 countries improve their productivity. The company is renowned for its powerful software portfolio, which combines innovative photo-editing, graphics-creation, vector-illustration, and technical-graphics applications with office and personal productivity solutions. Corel's flagship products include CorelDRAW Graphics Suite, WordPerfect® Office suite, Corel® Paint Shop Pro®, Corel® Painter™, and Corel DESIGNER® Technical Suite. For more information, please visit www.corel.com.



Chapter I

Understanding VBA



Before getting started with VBA in CorelDRAW Graphics Suite X3, it's important to understand a little bit about VBA in general.

This chapter answers the following questions:

- What is VBA?
- Who is VBA designed for?
- How does VBA compare with other programming languages?
- What are the main elements used in VBA?
- 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 CorelDRAW and Corel PHOTO-PAINT. CorelDRAW Graphics Suite X3 includes VBA version 6.3.

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 Corporation (in CorelDRAW Graphics Suite, Corel DESIGNER® Technical Suite, and 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 at <http://www.msdn.microsoft.com/vba/companies/company.asp>.

It is not necessary for an application to support VBA in order for the CorelDRAW Graphics Suite X3 VBA engine to control that application. That means you can build solutions in CorelDRAW Graphics Suite X3 that access databases, word processors, specialized content editors, XML documents, and more.

What is automation?

Most actions that you can do in CorelDRAW Graphics Suite X3 can be done programmatically through VBA. This programmability of CorelDRAW Graphics Suite X3 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 manual, a macro refers to VBA *functions* and *subroutines* (which are explained in “Building functions and subroutines” on page 10).

While it is possible to record a sequence of actions in CorelDRAW Graphics Suite X3, the real power of automation and VBA is that these recordings can be edited to provide conditional and looping execution. For example, a simple macro may set the selected shape’s fill color to red and apply a one-point outline; however, by adding a condition and a looping mechanism to the VBA code, the macro could, for example, be made to seek out each selected shape and apply only the fill to text shapes and only the outline to all other shape types.

Who is VBA designed for?

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

VBA for non-programmers

VBA is based on Microsoft’s successful 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, CorelDRAW or Corel PHOTO-PAINT).

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 manual 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 CorelDRAW Graphics Suite X3. 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 manual.

VBA for programmers

VBA is an in-process automation controller. In other words, it can be used to control the functionalities of CorelDRAW Graphics Suite X3 that can be automated. In addition, because VBA runs “in-process,” it bypasses the interprocess synchronization mechanisms so as to run much more efficiently.

All of the automation that is available to the in-process VBA is also available to external out-of-process automation controllers, or OLE clients. This includes applications developed in programming languages that can be used to develop OLE clients, such as the following:

- Microsoft Visual Basic, Visual C++, and Windows® Script Host
- Borland® Delphi™ and C++
- 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 12 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, CorelDRAW or Corel PHOTO-PAINT) so as to manipulate the host’s *object model* (which is discussed in “What is an object model?” on page 7).

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 13.)

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 10.)

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 13.)

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 CorelDRAW Graphics Suite X3. 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"). In the context of CorelDRAW, each open document is an instance of the Document class, each page in the document is an instance of the Page class, and each layer (and each shape on each layer) are more instances of more classes.

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. In the context of CorelDRAW, Document objects have a name, a resolution, and horizontal and vertical ruler units; individual shapes have outline and fill properties, as well as a position and a rotation factor; and text objects have text properties, which may include the text itself.

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." In the context of CorelDRAW, documents have methods for creating new pages, layers have methods for creating new shapes, and shapes have methods for applying transformations and effects.

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. In the context of CorelDRAW, a document object contains page objects, which contain layer objects, which contain shape objects, some of which contain other objects. 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. For example, in the context of CorelDRAW, the Shape type has many subtypes (or "inherited types"), including Rectangle, Ellipse, Curve, and Text. All of these subtypes can make use of the basic members of the Shape type, including methods for moving and transforming the shape and for setting its color. However, the subtypes also have their own specialist members; for example, a Rectangle can have corner radii, whereas Text has an associated Font property.

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. They also allow object types or classes to be used repeatedly in different ways; for example, a Shape object may be of the type "group" and may contain other Shape objects, some of which may also be of the type "group," or



“rectangle,” or “curve,” or “text.” This high level of organization and re-use makes the object model simple to use and yet powerful — and also easy to navigate by using the VBA Object Browser (which is discussed in “Using the Object Browser” on page 23).

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 CorelDRAW, the root object of all objects is the `Application` object, because all objects are children or grandchildren of the application.

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.

```
Application.Documents(1).Pages(1).Layers(1).Shapes(1).Name = "Bob"
```

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 example, the shortcut object `ActiveLayer` replaces the long-hand version

`Application.ActiveDocument.ActivePage.ActiveLayer`, while the object shortcut `ActiveSelection` replaces the long-hand version `Application.ActiveDocument.Selection`. For information on the shortcut objects that are available to CorelDRAW, see “Using shortcut objects” on page 32.

For detailed information on the CorelDRAW object model, see the Appendix.

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 20). 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 manual.

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 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.

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()`.

Multidimensional 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).

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 ANSI, 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:

Comparison	VBA operator	C-style operator
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.



Chapter 2

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:

- Installing VBA for CorelDRAW Graphics Suite X3
- Using the VBA toolbars
- Using the VB Editor

Installing VBA for CorelDRAW Graphics Suite X3

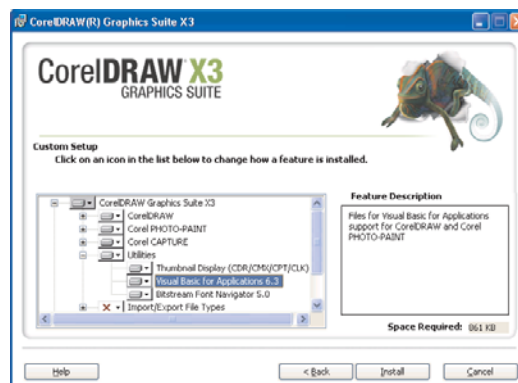
Before you can develop and run macros in CorelDRAW Graphics Suite X3, you may need to install the VBA feature.



When you perform a “typical installation” of CorelDRAW Graphics Suite X3, VBA is installed by default.

To install VBA

- 1 Insert CorelDRAW Graphics Suite X3 Disc 1 into the CD drive.
If the installation wizard does not start automatically, click **Start ▶ Run** on the Windows taskbar, and then type **X:\CGS13\Setup.exe** (where **X** is the letter that corresponds to the CD drive).
- 2 Click **Install CorelDRAW Graphics Suite X3**, and follow the on-screen instructions.
- 3 On the **Select which applications you would like to install** page, click **Advanced options**, and then in the Custom setup window that appears, set **Utilities\Visual Basic for Applications 6.3** to install.



*Set **Utilities\Visual Basic for Applications 6.3** to install*

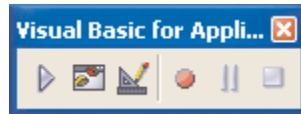


Using the VBA toolbars

CorelDRAW and Corel PHOTO-PAINT feature a toolbar that lets you access common VBA functionalities.

Using the VBA toolbar in CorelDRAW

CorelDRAW has a toolbar that provides easy access to several VBA features and to the VB Editor.



The Visual Basic for Applications toolbar in CorelDRAW

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
- recording macros
- pausing the recording of macros
- stopping the recording of macros

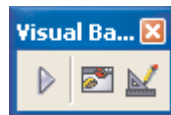
To display the Visual Basic for Applications toolbar in CorelDRAW

- Click **Window** ► **Toolbars** ► **Visual Basic for Applications**.

A check mark next to the command indicates that the toolbar is displayed.

Using the VB Editor toolbar in Corel PHOTO-PAINT

Corel PHOTO-PAINT has a toolbar that provides easy access to the VB Editor.



The Visual Basic Editor toolbar in Corel PHOTO-PAINT

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 VB Editor toolbar in Corel PHOTO-PAINT

- Click **Window** ► **Toolbars** ► **Visual Basic Editor**.

A check mark next to the command indicates that the toolbar is displayed.

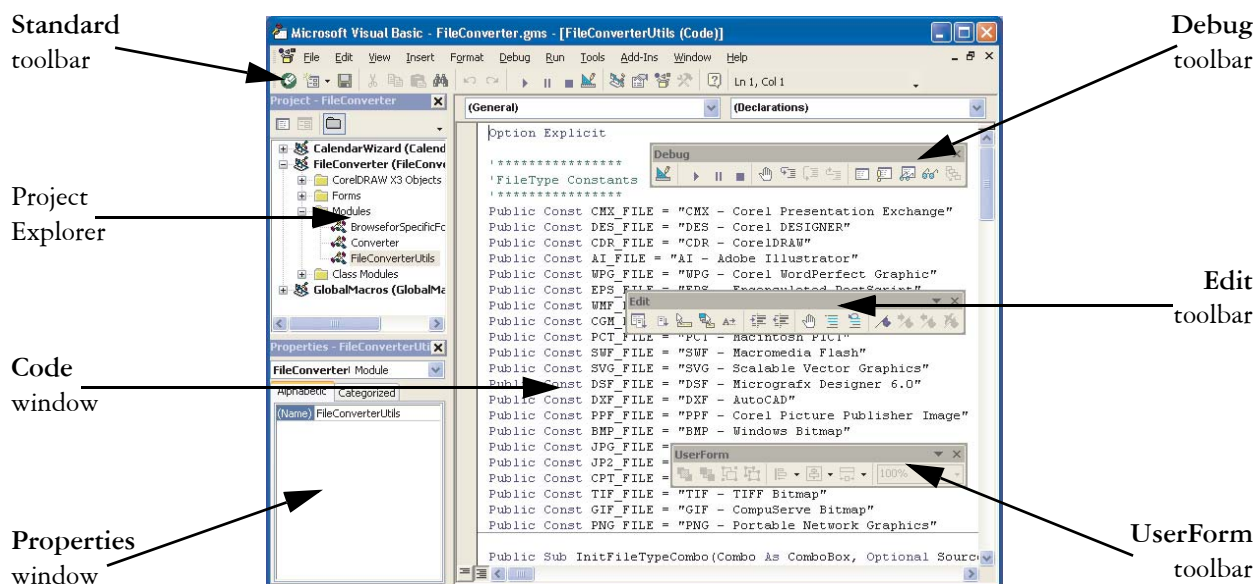


Using the VB Editor

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 17), the Properties window (see page 18), and the Code window (see page 19). The four available toolbars (see page 22), 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 23).

You can invoke the VB Editor from within CorelDRAW or Corel PHOTO-PAINT. Although this starts VBA as a new application in Windows, it runs within the CorelDRAW or Corel PHOTO-PAINT process.

To start the VB Editor

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

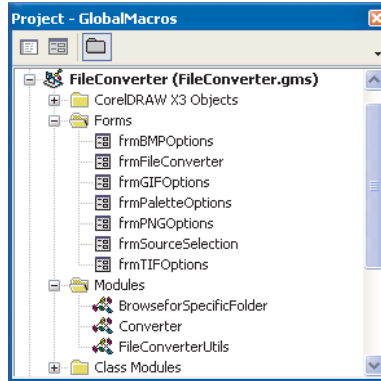


To switch between the VB Editor and CorelDRAW or Corel PHOTO-PAINT, use the Windows taskbar, or press Alt + F11 or Alt + Tab.

Using the Project Explorer

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
	form
	module
	class module

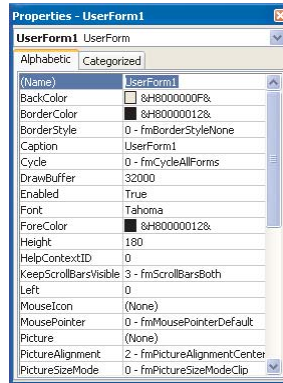
To display or hide the Project Explorer

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

Using the Properties window

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).

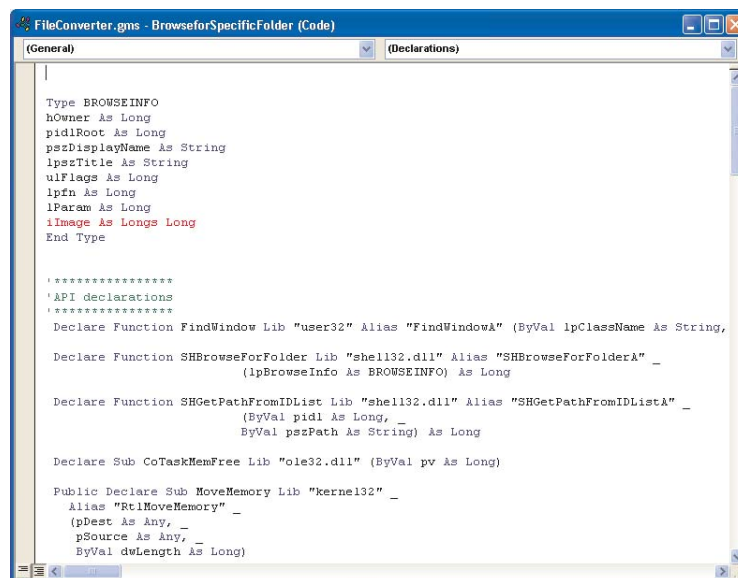
To display or hide the Properties window

- Click View ► Properties window, or press F4.

Using the Code window

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.



The VBA Code window



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)
```

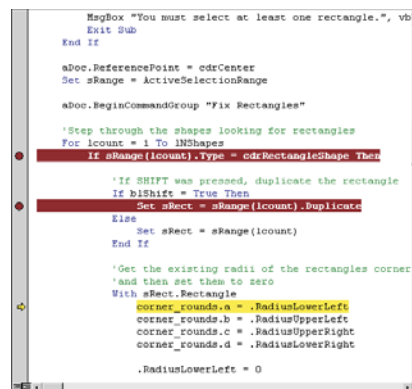
```
Call fooBarSub (f)
```

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.



Syntax highlighting and coloring

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 35.



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

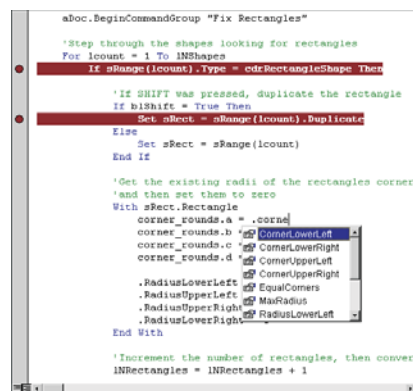
You can jump directly to the definition of a variable, procedure, or object by right-clicking the item in the Code window and then clicking **Definition**. This takes you either to the definition of the variable or function in the code or to the object’s definition in the Object Browser.



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.



Auto-completion pop-up menu



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

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 42).



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.



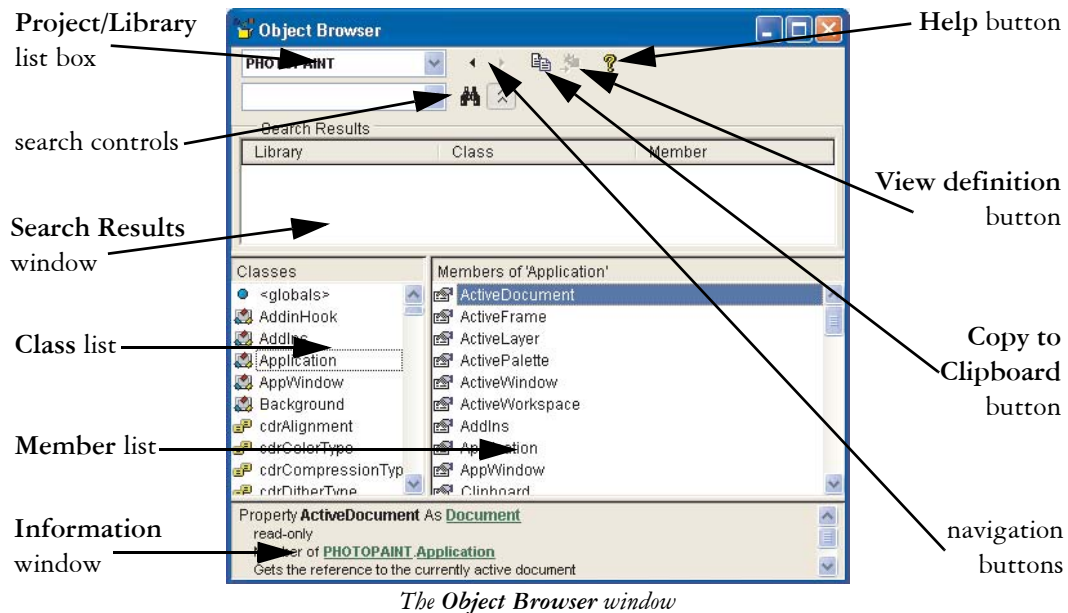


You can “float” a toolbar by dragging it from the menu bar.

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

Using the Object Browser

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 CorelDRAW or Corel PHOTO-PAINT — all in an easy-to-use, structured format.



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. For example, collection types have the default property `Item`, which can be indexed. However, because `Item` is usually the default property, in such instances, it is not necessary to specify the item property. Here,

```
ActiveSelection.Shapes.Item(1).Selected = False
```

is the same as the shorter

```
ActiveSelection.Shapes(1).Selected = False
```

because `Item` is the default or implied property of a collection of shapes.

Methods are commonly known as “member functions” — functions that the class can perform on itself. A good example is the `Move` method of the `Shape` class, which is used to move a `CorelDRAW` shape using an `[x, y]` vector. The following code moves the selected shapes 2 measurement units to the right and 3 measurement units upwards:

```
ActiveSelection.Move 2, 3
```

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.



For more information about `ActiveSelection`, see the Appendix.

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.

Commonly handled events include the `BeforeClose`, `BeforePrint`, `BeforeSave`, `ShapeMove`, `PageActivate`, and `SelectionChange` events of the `Document` class.



Only the `Document`, `GlobalMacroStorage`, and `AddinHook` classes have events in `CorelDRAW`. The `AddinHook` class is not discussed in this manual.

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, such as `CorelDRAW` shape types, import/export filters, and alignments. These constants can be used anywhere an integer value is required.

Most `CorelDRAW` constants begin with `cdr` (for example, `cdrLeftAlignment`, `cdrEPS`, `cdrOutlineBevelLineJoin`, `cdrCurveShape`, and `cdrSymmetricalNode`). Some constants begin with `prn` or `pdf`. Visual Basic also has its own constants, including ones for keystrokes (such as `vbKeyEnter`) and dialog box buttons (such as `vbOK`).

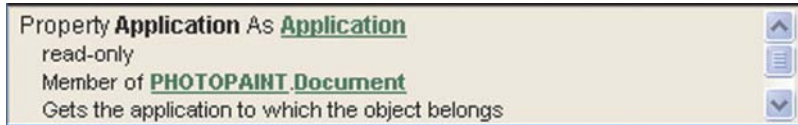


To receive detailed information about a selected item, click the **Help** button at the top of the Object Browser.

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

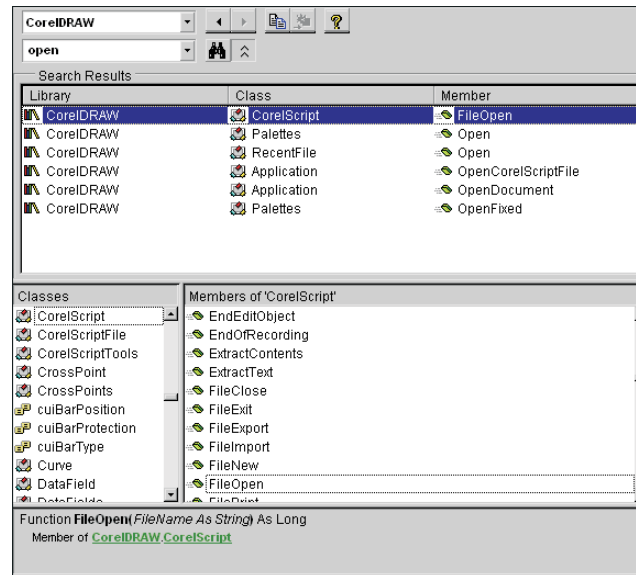
The types of any function parameters and properties are given as hyperlinks to the type definition or class itself, if the type is defined within the current object model. For example, the **Application** property in the preceding figure is a member of the **PHOTOPAINT Document** class; however, **Application** is also the property's type, and it is a **PHOTOPAINT** class, so to view the class and its members, you can click the **Application** hyperlink.



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 .



Chapter 3

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
- Referencing objects in macros
- Providing event handlers in macros
- Running macros
- Debugging macros



For detailed information on the CorelDRAW object model, see the Appendix.

Creating macros

You can create a macro either by writing it or by recording it.

Writing macros

Before you can begin writing a macro, you must create a Global Macro Storage (GMS) file — also known as a project file — for it. A GMS file is stored in its application's **GMS** folder, which is typically located at **X:\Program Files\Corel\CorelDRAW Graphics Suite 13\<application>** (where X is the letter that corresponds to the drive to which you've installed CorelDRAW Graphics Suite X3). The VB Editor stores all of the modules for that project in the project's GMS file.

Each project that you create can contain several modules. The Project Explorer (see page 17) 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:

- **CorelDRAW X3 Objects or Corel PHOTO-PAINT X3 Objects** — used mostly for event handling, contains a single item (**ThisMacroStorage** or **ThisDocument**, 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 manual)

After you create a GMS file, you can change its project name and add modules to it.



To write the macro, you must 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 manual, 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 CorelDRAW or Corel PHOTO-PAINT.)

To create a project file

- 1 Close the application (that is, CorelDRAW or Corel PHOTO-PAINT) for which you want to create the project file.
- 2 In Windows Explorer, navigate to the **GMS** folder for the application for which you want to create the project file.
- 3 Click **File ► New ► Text document**.
A new, empty text document is created in this folder.
- 4 Rename the file as **<filename>.gms**, where **<filename>** is any valid Windows filename.
- 5 Restart the application for which you've created the project file.
When you launch the VB Editor, the project file you've created is displayed in the Project Explorer as **Global Macros (<filename>.gms)**.

To rename a project

- 1 In the Project Explorer, select the project you want to rename.
- 2 In the **Properties** window, edit the **(Name)** value.
Names must follow normal variable-naming conventions, so they must begin with an alphabetic character, and they must not contain spaces nor special characters other than underscores (**_**).
- 3 Press **Enter** to commit your changes.

To add a module to a project

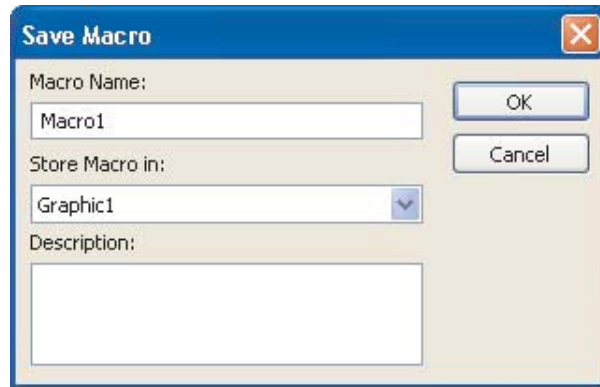
- 1 In the Project Explorer, right-click the project to which you want to add a module.
- 2 Click one of the following
 - **Insert ► Module** — inserts a normal code module
 - **Insert ► UserForm** — inserts a new form (that is, dialog box)
 - **Insert ► Class Module** — inserts a new class code module

The new module is stored in the project's folder for that type of module.

Recording macros

You can record macros from directly within CorelDRAW (that is, without using the VB Editor). Recording a macro creates (in the chosen project) a new VBA macro, which can then be edited and customized by using the VB Editor.





The Save Macro dialog box

You may prefer to create macros by recording if you are not familiar with the application's object model or are not sure which objects and methods to use. For many simple and repetitive tasks, recorded macros are a quick, efficient solution; they store the sequence of keys you press and mouse actions you perform within CorelDRAW, so that you can reuse that sequence in the future.



You cannot record macros in Corel PHOTO-PAINT.

To record a macro

- 1 Click **Tools ▸ Visual Basic ▸ Record**, or click the **Record** button  on the **Visual Basic for Applications** toolbar.
- 2 From the **Store macro in** list box, choose the VBA project (GMS) file or CorelDRAW (CDR) file in which you want to store the recorded macro.
- 3 In the **Macro name** box, type a name that is unique to the file in which you have chosen to store the macro. Macro names can contain numerals, but they must begin with a letter. Macro names cannot contain spaces or non-alphanumeric characters other than underscores (_).
- 4 If you want, type a description of the macro in the **Description** box.
- 5 Click **OK**.

CorelDRAW begins recording your actions. To pause or resume recording, click **Tools ▸ Visual Basic ▸ Pause**, or click the **Pause** button  on the **Visual Basic for Applications** toolbar.

- 6 To stop recording, click **Tools ▸ Visual Basic ▸ Stop**, or click the **Stop** button  on the **Visual Basic for Applications** toolbar.

The macro is saved using the settings you've specified.



Not all actions can be recorded, some because of their complexity (although many such actions can be manually coded in the VB Editor). When an action cannot be recorded, the following comment is placed in the code: The recording of this command is not supported.



Referencing objects in macros

If you want to create a reference to an object so that you can treat that reference like a variable (sh, in the example that follows), you can use the Set keyword:

```
Dim sh As Shape
Set sh = ActiveSelection.Shapes.Item(1)
```

After you create this reference, you can treat it as if it were the object itself:

```
sh.Outline.Color.GrayAssign 35
```

If the selection is changed while sh is still in scope, sh references the original shape from the old selection and is unaffected by the new selection. You cannot simply assign the object to the variable as in the following example:

```
Dim sh As Shape
sh = ActiveSelection.Shapes.Item(1)
```

To release an object, you must set its reference value to Nothing:

```
Set sh = Nothing
```

You can also test whether a variable references a valid object by using the Nothing keyword:

```
If sh Is Nothing Then MsgBox "sh is de-referenced."
```

Objects do not need to be explicitly released. In most cases, Visual Basic releases the object when it disposes of the variable when you exit the function or sub.



For detailed information on the CorelDRAW object model, see the Appendix.

Referencing collections in macros

Many objects are members of *collections* of objects. A collection is similar to an array, except that it contains objects rather than values. Despite this, members of collections can be accessed in the same way as arrays. For example, a collection that is used frequently in CorelDRAW is the collection of shapes on a layer: the object **ActiveLayer** references either the current layer or the layer that is selected in the **CorelDRAW Object Manager** docker.

CorelDRAW contains many collections: a document contains pages, a page contains layers, a layer contains shapes, a curve contains subpaths, a subpath contains segments and nodes, a text range contains lines and words, a group contains shapes, and the application contains windows. All of these collections are handled in the same way by VBA.



For detailed information on the CorelDRAW object model, see the Appendix.

Referencing the items in a collection

To reference the shapes on a layer, that layer's collection of shapes is used: **ActiveLayer.Shapes**. To reference the individual shapes in the collection, the **Item()** property is used:

```
Dim sh As Shape
Set sh = ActiveLayer.Shapes.Item(1)
```



Most elements of a collection start at 1 and increase. For the collection `ActiveLayer.Shapes`, `Item(1)` is the item at the “top” or “front” of the layer — in other words, the one that is in front of all the other shapes.

Because each item in the `ActiveLayer` collection is an object of type `Shape`, you can reference the item’s members merely by appending the appropriate dot-notated member:

```
ActiveLayer.Shapes.Item(1).Outline.ConvertToObject
```

Sometimes, individual items have names. If the item you are looking for has an associated name (and you know what the name is and which collection the item is in), you can reference the item directly by using its name:

```
Dim sh1 As Shape, sh2 As Shape
Set sh1 = ActiveLayer.CreateRectangle(0, 5, 7, 0)
sh1.Name = "myShape"
Set sh2 = ActiveLayer.Shapes.Item("myShape")
```

Also, because an item is usually the implied or default member of a collection, it is not strictly required, so the last line of the preceding code can be rewritten as follows:

```
Set sh2 = ActiveLayer.Shapes("myShape")
```

Counting the items in a collection

All collections have a property called `Count`. This read-only property gives the number of members in the collection:

```
Dim count As Long
count = ActiveLayer.Shapes.Count
```

The returned value is not only the number of items in the collection, but because the collection starts from 1, it is also the index of the last item.

Parsing the items in a collection

It is often necessary to parse through the members of a collection to check or change the items’ properties.

By using the `Item()` and `Count` members, it is straightforward to step through a collection of items. With each iteration, it is possible to test the current item’s properties, or to call its methods. The following code restricts all shapes on the layer to no wider than ten units:

```
Dim I As Long, count As Long
count = ActiveLayer.Shapes.Count
For I = 1 to count
    If ActiveLayer.Shapes.Item(i).SizeWidth > 10 Then
        ActiveLayer.Shapes.Item(i).SizeWidth = 10
    End If
Next I
```



There is, however, a more convenient way of parsing a collection in VBA. Instead of using the `Count` property and a `For -Next` loop, this technique uses a `For -Each -In` loop:

```
Dim sh As Shape
For Each sh In ActiveLayer.Shapes
    If sh.SizeWidth > 10 Then
        sh.SizeWidth = 10
    End If
Next sh
```

If you want to copy the selection and then parse it later when it is no longer selected, copy the selection into a `ShapeRange` object:

```
Dim sr As ShapeRange
Dim sh As Shape
Set sr = ActiveSelectionRange
For Each sh In sr
    ' Do something with each shape
Next sh
```

Using shortcut objects

CorelDRAW provides several shortcuts to frequently accessed objects. These shortcuts are easier to use than their longhand versions because they require less typing. (Also, using shortcuts can improve run-time performance because the compiler does not have to determine every object in a long dot-separated reference.)

The following table provides the shortcuts and their long forms, as well as a description of each:

Shortcut	Long form	Description
ActiveLayer	ActivePage.ActiveLayer	Gets the current layer being edited in the current page of the document
ActivePage	ActiveDocument.ActivePage	Gets the current page of the active document
ActiveSelection	ActiveDocument.Selection	Gets the selection in the active document
ActiveSelectionRange	ActiveDocument.SelectionRange	Gets the selection range in the active document
ActiveShape	ActiveDocument.Selection.Shapes(1)	Gets the last-selected shape in the active document
ActiveView	ActiveWindow.ActiveView	Gets the active view of the document (that is, the representation of the document within the <code>ActiveWindow</code>)



Shortcut	Long form	Description
ActiveWindow	ActiveDocument.ActiveWindow	Gets the active window (that is, the window in which the active document is displayed)

A shortcut can be used on its own as a property of the CorelDRAW Application object.

The following shortcuts can also be used as members of a given Document object: **ActiveLayer**, **ActivePage**, **ActiveShape**, and **ActiveWindow**. The Document object also has the properties **Selection** and **SelectionRange**, which let you get the selection or selection range (respectively) from a specified document regardless of whether that document is active.

Providing event handlers in macros

While running, CorelDRAW raises various events to which VBA can respond through the use of *event handlers* — subroutines with specific, defined names within a **ThisMacroStorage** module in CorelDRAW. Every CorelDRAW VBA project (that is, **GMS**) file has one **ThisMacroStorage** module within its **CorelDRAW X3 Objects** subfolder for the project. (Similarly, every Corel PHOTO-PAINT VBA project has one **ThisDocument** module within its **Corel PHOTO-PAINT X3 Objects** subfolder for the project.)

The **GlobalMacroStorage** object is a virtual object that represents each and all open CorelDRAW documents. The **GlobalMacroStorage** object has several events that are raised at the time of any event, such as opening, printing, saving, or closing a CorelDRAW document (although the range of events is actually greater than this because each one has a “before” and “after” event).

To respond to an event, you must provide an event handler — a subroutine in any **ThisMacroStorage** module with a specific name for which CorelDRAW is pre-programmed to search. However, CorelDRAW does check all **ThisMacroStorage** modules in all installed projects, so you can create an event-driven solution and distribute it as a single project file just as you would provide any other form of solution. Each project can have only one **ThisMacroStorage** module, and it is automatically created when the project is first created.

For example, a solution may need to respond to a document closing by logging the closure in a file as part of a workflow-management and recording system. To respond to a document opening, the solution must respond to the **GlobalMacroStorage** class’s **OpenDocument** event. To do this, open a **ThisMacroStorage** module for editing in the VB Editor; next, choose **GlobalMacroStorage** from the **Object** list box at the top of the **Code** window, and then choose **DocumentOpen** from the **Procedure** list box. The VB Editor creates a new, empty sub called **GlobalMacroStorage_DocumentOpen()** — or, if it exists already, places the cursor into it. You then need only write some code that adds the name of the opened file to the log. The recommended way to do this is to create a public sub in a different module that actually does the work, such that the event handler simply calls that sub when required; this keeps the size of the **ThisMacroStorage** module as small as possible, so that the run-time interpreter can more easily parse all the **ThisMacroStorage** modules each time an event is raised. The following code illustrates this example:

```
Private Sub GlobalMacroStorage_OpenDocument( ByVal Doc As Document, _
                                             ByVal FileName As String)
    Call LogFileOpen(FileName)
End Sub
```



Here is a small sample of the events available in CorelDRAW:

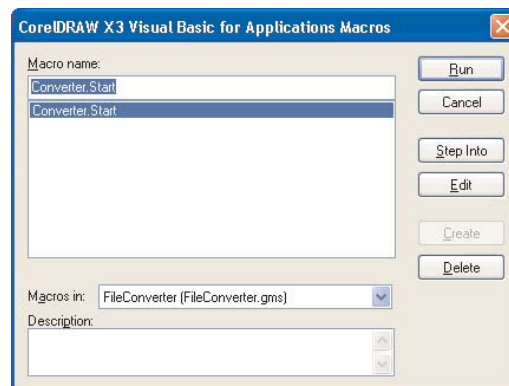
Event	Description
Start	Raised when the user starts CorelDRAW
DocumentNew	Raised when a document is created; passes a reference to the document
DocumentOpen	Raised when a document is opened; passes a reference to the document
DocumentBeforeSave	Raised before a document is saved; passes the file name of the document as a parameter
DocumentAfterSave	Raised after a document is saved; passes the file name of the document as a parameter
DocumentBeforePrint	Raised before the Print dialog box is displayed
DocumentAfterPrint	Raised after a document is printed
SelectionChange	Raised when a selection changes
DocumentClose	Raised before a document is closed
Quit	Raised when the user quits CorelDRAW



Event handlers for frequent events — such as events related to the Shape class — should be as efficient as possible, to keep the application running as quickly as possible.

Running macros

You can run macros either from directly within CorelDRAW or Corel PHOTO-PAINT or from within the VB Editor.



The CorelDRAW X3 Visual Basic for Applications Macros dialog box



To run a macro from within CorelDRAW

- 1 Click Tools ► Visual Basic ► Play, or click the Play button  on the Visual Basic for Applications toolbar.
- 2 From the Macros in list box, choose the VBA project (GMS) file or CorelDRAW (CDR) file 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 Corel PHOTO-PAINT

- 1 Click Tools ► Visual Basic ► Play, or click the Play button  on the Visual Basic Editor toolbar.
- 2 From the Macros in list box, choose the VBA project (GMS) file 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 manual.

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.

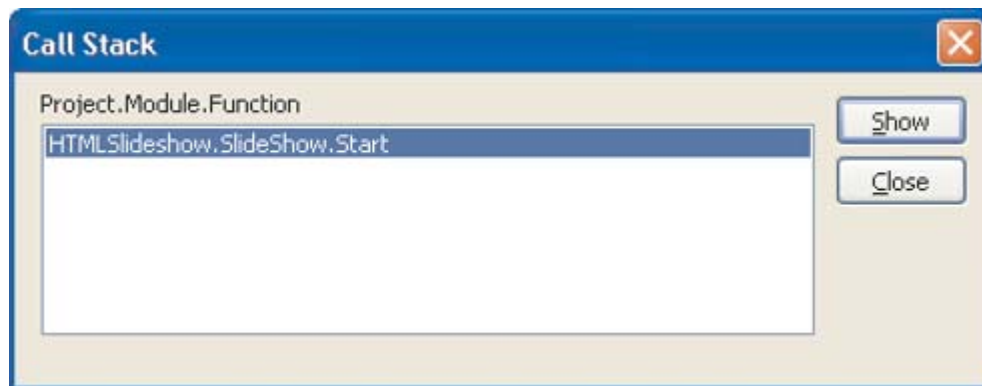
Using the debugging windows

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**.



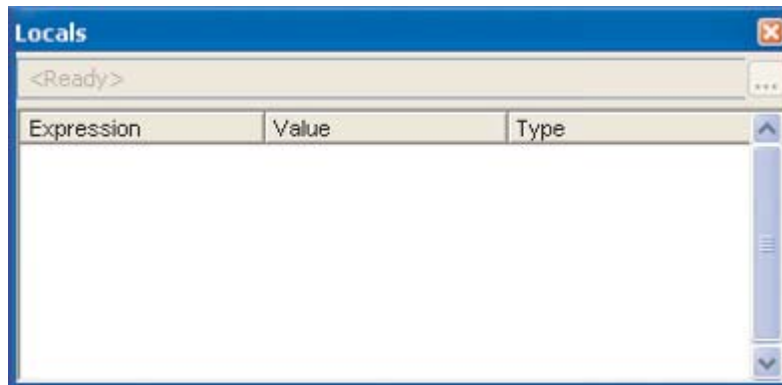


The Immediate window

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**.

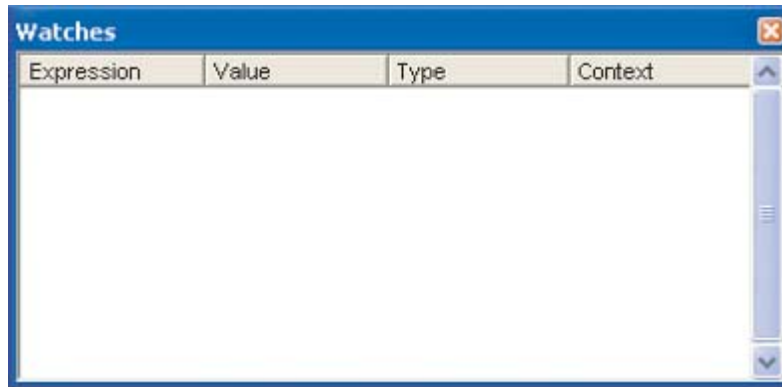


The Locals window

Using the Watches window

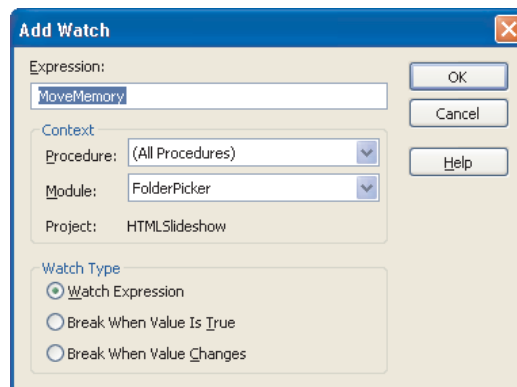
The **Watches** window is used to watch specific variables or object properties. This is very useful for selecting just one or two values to watch as opposed to having to find the value you want in all the values in the **Locals** window.





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.





Chapter 4

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). Still other user-interfaces require the user to click or drag with the mouse.

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
- Creating toolbars and buttons for macros
- Providing user interaction for macros
- 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.

For example, if you are creating a solution that allows an effect or action to be applied to one or more shapes, and if you think that the user will want to apply the same effect or action subsequently to a different selection of shapes, then you should provide a modeless dialog box so that the user can set up the effect in the dialog box once and then apply it many times.

If, on the other hand, if you are creating a solution that is a “one-shot” end-to-end solution (such as a replacement **Print** or **Save** dialog box), then a modal dialog box would be more appropriate; in a case such as this, it is unlikely that a user would want to apply the same settings repeatedly, so having to re-opening the dialog box would be less convenient for them than having to manually close it.

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:

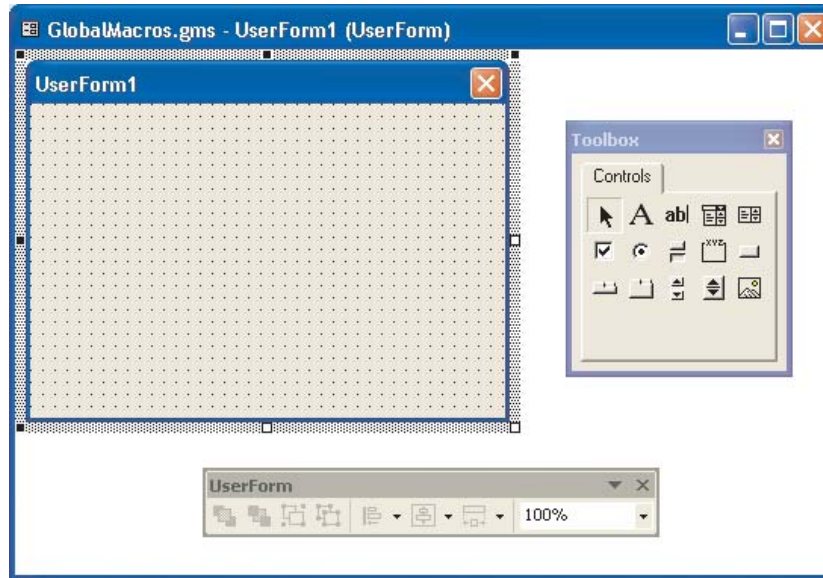
- an **Apply** or **Create** button — performs the dialog box’s action and can, in fact, be specially labelled to described the dialog box’s action. This button should be the default.
- a **Close** button — closes the dialog box. This button is used after the user has applied the desired action.

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” on page 22.

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. For more information on event handlers, see “Providing event handlers in macros” on page 33.

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 44.
	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 44.
	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 44.
	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)
	RadioButton	Lets you provide a “radio button” that is linked to other radio buttons with the same <code>GroupName</code> 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 44.
	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 <code>TextBox</code>) 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 46.

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
```

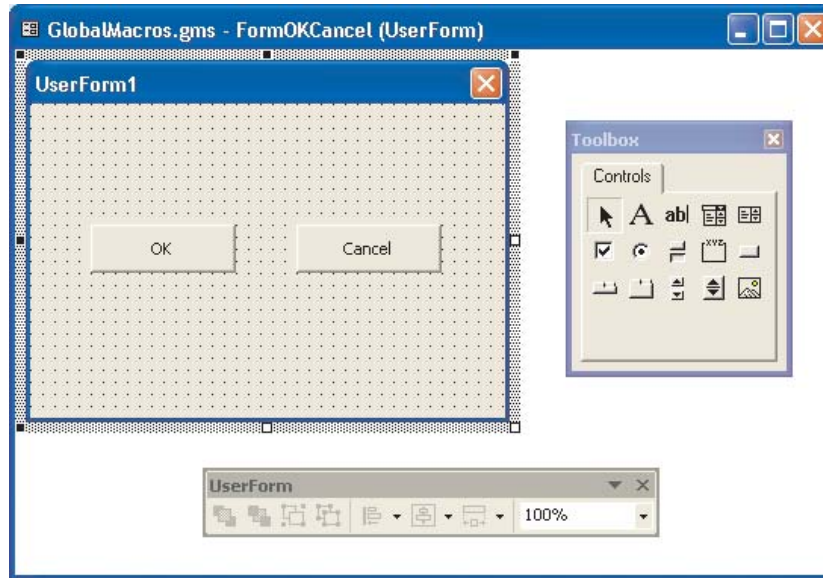
To test which item is selected when the **OK** button is clicked, test the list's `ListIndex` property. To get the value of a selected item's caption, test the `Text` property for the `ComboBox` or `Listbox`:

```
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`.

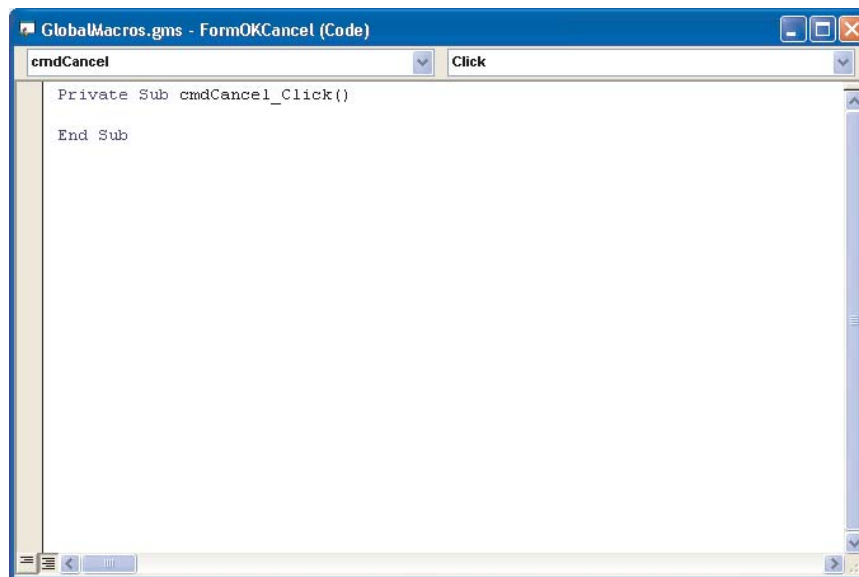




Designing buttons in the Form Designer

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
```

Similarly, the size-setting subroutine could resemble the following:

```
Private Sub SetSize(width As String, height As String)  
    ActiveDocument.Unit = cdrInch  
    ActiveSelection.SetSize Cdbl(width), Cdbl(height)  
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")
```

Creating toolbars and buttons for macros

Toolbars for any VBA solution are almost always visible. Toolbars are useful because their graphical icons are memorable although they occupy only a small area, and because their buttons can have meaningful titles that are displayed as part of the face and tooltips that display when the pointer hovers over them.



Creating toolbars for macros

When creating toolbars, you should plan carefully. It is better to have lots of small toolbars containing a few related buttons than one big toolbar containing all of the buttons for all of your macros. By breaking your buttons into small groups, it is much easier to deploy them with the projects to which they belong.

To create a toolbar

- 1 Click **Tools ▶ Options**.
- 2 Click **Workspace ▶ Customization ▶ Command bars**.
- 3 Click **New**.
- 4 Type a new name for the toolbar.
- 5 Enable the check box next to the toolbar's name.

To add macro buttons to a toolbar

- 1 Click **Workspace ▶ Customization ▶ Commands**.
- 2 Choose **Macros** (in CorelDRAW) or **VBA Macros** (in Corel PHOTO-PAINT) from the **Commands** list box.
The list displays the fully qualified names of all of the public, parameter-free subs from all of the installed project (GMS) files.
- 3 Drag a macro from the list to the toolbar.
The macro appears on the toolbar with the default macro icon.

Adding captions and tooltips to macros

Each macro can have both a caption and a tooltip. The caption is displayed whenever the menu command is used and can be displayed as part of a button, while the tooltip appears when the pointer hovers over the menu item or button.

To set a caption for a macro

- 1 Click **Workspace ▶ Customization ▶ Commands**.
- 2 Choose **Macros** from the **Commands** list box.
The list displays the fully qualified names of all of the public, parameter-free subs from all of the installed project (GMS) files.
- 3 Select a macro in the **Command** list.
- 4 Click the **Appearance** tab.
- 5 Type the caption in the **Caption** box.
To associate with a caption an accelerator key that can be activated in combination with the **Alt** key, type an ampersand (&) in front of the character in the caption that you want to set as the accelerator. This accelerator key applies only to menu commands, which display accelerator characters with an underscore (_).



To set a tooltip for a macro

1 Click **Workspace ► Customization ► Commands**.

2 Choose **Macros** from the **Commands** list box.

The list displays the fully qualified names of all of the public, parameter-free subs from all of the installed project (GMS) files.

3 Select a macro in the **Command** list.

4 Click the **General** tab.

5 Type the tooltip in the **Tooltip help** box.

Associating images or icons with macros

Commands can have a small image or icon associated with them. This icon can be displayed or hidden on toolbars and menus, and it can be sized as small (16 × 16 pixels), medium (32 × 32 pixels), or large (48 × 48 pixels).

To associate an image or icon with a macro

1 Click **Workspace ► Customization ► Commands**.

2 Choose **Macros** from the **Commands** list box.

The list displays the fully qualified names of all of the public, parameter-free subs from all of the installed project (GMS) files.

3 Select a macro in the **Command** list.

4 Click the **Appearance** tab.

5 Do one of the following:

- To apply a Windows bitmap (BMP) image to the macro, click **Import**, navigate to where the image is stored, and select it. The image's colors are mapped to their closest available matches.
- To create a customized icon for the macro, edit the pixels displayed in the **Image** area.

Providing user interaction for macros

The CorelDRAW object model provides two main ways to receive input from users as they work with documents: capturing mouse actions and capturing coordinates.

Capturing mouse actions

You can capture the position of a mouse click by using the `Document.GetUserClick` function.

You can capture the position of a mouse drag by using the `Document.GetUserArea` function.



Capturing mouse clicks

To get from the user the position of a single mouse click, use the `GetUserClick` member function of the `Document` object. This function pauses the macro for a specified period of time, or until the user clicks somewhere in the document or presses **Escape**. Here is an example that uses the `GetUserClick` function:

```
Dim doc As Document, retval As Long
Dim x As Double, y As Double, shift As Long
Set doc = ActiveDocument
doc.Unit = cdrCentimeter
retval = doc.GetUserClick(x, y, shift, 10, True, cdrCursorPick)
```

The following parameters for the `GetUserClick` function are coded into this example:

- In the variables `x` and `y`, the position of the mouse click is returned.
- In the parameter `shift`, the combination of the **Shift**, **Ctrl**, and **Alt** keys that the user holds down when clicking the mouse is returned. The **Shift**, **Ctrl**, and **Alt** keys are assigned values of 1, 2, and 4 (respectively); these values are added together before being returned.
- 10 seconds is given to the user to click somewhere in the document.
- The `SnapToObjects` parameter is enabled (that is, set to `True`).
- The **Pick** tool's icon is used for the cursor icon. (You cannot use a custom icon.)

The returned value is 0, 1, or 2, depending on whether the user successfully completes the click, the user cancels by pressing **Escape**, or the operation times out (respectively).

The returned coordinates of the click are relative to the origin of the page and are in the document's VBA units, so it is often necessary to explicitly set the units in which you want the value to be returned.

To get the shapes that are underneath the returned click point, use the function `SelectShapesAtPoint`, which is a member of `Page`:

```
doc.ActivePage.SelectShapesAtPoint x, y, True
```

The value indicates whether to select unfilled objects (`True`), or not (`False`).

Capturing mouse drags

To get from the user the position of a mouse drag, or an area or rectangle, use the `GetUserArea` member function of the `Document` object. This function pauses the macro for a specified period of time or until the user clicks, drags, and releases somewhere in the document, or until the user presses **Escape**. Here is an example that uses the `GetUserArea` function:

```
Dim doc As Document, retval As Long, shift As Long
Dim x1 As Double, y1 As Double, x2 As Double, y2 As Double
Set doc = ActiveDocument
doc.Unit = cdrCentimeter
retval = doc.GetUserArea(x1, y1, x2, y2, shift, 10, True, cdrCursorExtPick)
ActivePage.SelectShapesFromRectangle x1, y1, x2, y2, False
```



The following parameters for the `GetUserArea` function are coded into this example:

- In the variables `x1`, `y1`, `x2`, and `y2`, the position of the area is returned as two opposite corners of a rectangle.
- In the parameter `shift`, the combination of the **Shift**, **Ctrl**, and **Alt** keys that the user holds down when clicking the mouse is returned. The **Shift**, **Ctrl**, and **Alt** keys are assigned values of 1, 2, and 4 (respectively); these values are added together before being returned.
- 10 seconds is given to the user to click somewhere in the document.
- The `SnapToObjects` parameter is enabled (that is, set to `True`).
- The cursor icon to be used is defined by the last parameter.

In this example, the code ends by selecting the shapes that lie completely within the area by using the `Page` object's `SelectShapesFromRectangle` method.

The returned value is 0, 1, or 2, depending on whether the user successfully completes the area selection, the user cancels by pressing **Escape**, or the operation times out (respectively).

The returned coordinates of the area are relative to the origin of the page and are in the document's VBA units, so it is often necessary to explicitly set the units in which you want the value to be returned.

This method returns two points that are interpreted as the corners of a rectangle. However, the two points can also be used as the start and end points of a mouse drag.

Capturing coordinates

You can convert from screen to document coordinates or from document to screen coordinates by using the `Window.ScreenToDocument` or `Window.DocumentToScreen` method (respectively).

You can determine if a set of coordinates is on a shape by using the `Shape.IsOnShape` method.

Converting coordinates

When capturing mouse actions, or when developing an complex VBA solution, it can be useful to convert between screen coordinates and document coordinates. This is done with the member functions `ScreenToDocument` and `DocumentToScreen` of the `Window` object.

The following code takes a coordinate from the screen and converts it into a point in the document that is visible in the active window:

```
Dim docX As Double, docY As Double
ActiveDocument.Unit = cdrMillimeter
ActiveWindow.ScreenToDocument 440, 500, docX, docY
```

The following code returns the screen coordinate of a point in the document as it displays on the screen:

```
Dim screenX As Long, screenY As Long
ActiveDocument.Unit = cdrMillimeter
ActiveWindow.DocumentToScreen 40, 60, screenX, screenY
```

In both cases, the converted coordinates are returned in the last two parameters.



Screen coordinates start from the upper-left corner of the screen, so positive Y values are down the screen, whereas negative Y values in the document are up the screen.



Testing coordinate placement

You can test whether a set of coordinates (that is, a point) is inside, outside, or on the outline of a curve by using the shape's `IsOnShape` member function. This function takes a document coordinate and returns `cdrInsideShape` if the coordinate is inside the shape, `cdrOutsideShape` if the coordinate is outside the shape, or `cdrOnMarginOfShape` if the coordinate is on or near the outline of the shape.

For example, the following code tests where the point (4, 6) is in relation to the `ActiveShape`:

```
Dim onShape As Long
ActiveDocument.Unit = cdrInch
onShape = ActiveShape.IsOnShape(4, 6)
```

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`.



Chapter 5

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 (GMS) 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 CorelDRAW or Corel PHOTO-PAINT.

Deploying macros

You can deploy macros to users for installation. You can deploy project files or workspaces, or both.

Deploying project files

Because every CorelDRAW or Corel PHOTO-PAINT document has an intrinsic project (GMS) file, 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 (GMS) 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 GMS file

- 1 Make the GMS file available to the users.
- 2 Instruct the users to install the GMS file by saving it to their **GMS** folder.



Deploying workspaces

When you have developed a customized workspace that contains customized toolbars, menus, and shortcut keys, these customizations can form an intrinsic part of your macro solution.

You can deploy the entire workspace to users for installation.

Alternatively, you can export some features of your workspace so that users can import only those features into their workspace. You can distribute individual toolbars and menus, as well as complete sets of shortcut keys.

Users can import a workspace or workspace features by using the `Application.ImportWorkspace` method or by manually importing the Corel workspace (XSLT) files.

To export workspace features

- 1 Right-click the toolbar, and click **Customize ► Workspace ► Export workspaces....**
- 2 In the list, enable the check boxes next to the workspace features you want to export.
- 3 Click **Save**.
- 4 In the **File name** box, type a filename.

The workspace features you've specified are saved to a single Corel Workspace (XSLT) file with the filename you've provided.



The status bar and the sizes and positions of dockers can also be exported.



If you want, you can export each workspace feature to a separate file. Simply export one item at a time to create a series of XSLT files.

When you export shortcut keys, you export all shortcut keys. If you want to distribute only a few keys, create a new workspace and remove all the shortcut keys from it, and then create only the keys that you want to export.

To import workspace features manually

- 1 Right-click the toolbar, and click **Customize ► Workspace ► Import workspaces....**
- 2 Click **Browse**.
- 3 Navigate to and select the Corel workspace (XSLT) file from which you want to import features.
- 4 Click **Next**.
- 5 In the list, enable the check boxes next to the workspace features you want to export.
- 6 Click **Next**.
- 7 Choose a destination for the workspace features by doing one of the following:
 - Enable the **Current workspace** option to import the workspace features you've specified into the current workspace, and then click **Next**.
 - Enable the **New workspace** option to import the workspace features into a new workspace. Click **Next**, and provide details about the workspace. Click **Next**.
- 8 Confirm the details of the import, and then click **Finish**.



The workspace features you've specified are imported into the workspace you've specified.



If the name of an incoming toolbar clashes with the name of an existing toolbar, the incoming toolbar is renamed.

If an imported command calls a VBA macro that is not installed, it does not function.



Appendix

Understanding the CorelDRAW object model



In CorelDRAW, the `Application` object is the root object for all other objects. To reference the CorelDRAW object model from an out-of-process controller, you use its `Application` object. For example, in VB you would use the following code:

```
Dim cdr As CorelDRAW.Application
Set cdr = CreateObject("CorelDRAW.Application.13")
```

Although you can use the above code in VBA, it's not required for CorelDRAW because the `Application` object is used by default if no other root object is specified.

The `Application` object contains all of the `Document` objects that are created when documents are opened in its property `Documents`. It also contains all `Window` objects. See “Working with documents” on page 55 for more information.

`Document` objects contain `Pages` collections of all their `Page` objects. See “Working with pages” on page 65 for more information.

Individual `Page` objects contain `Layers` collections of all their `Layer` objects. See “Working with layers” on page 69 for more information.

Finally, `Layer` objects contain `Shapes` collections of all their `Shape` objects. See “Working with shapes” on page 71 for more information.



To view a diagram of the CorelDRAW object model, consult the file **CorelDRAW VBA Object Model.pdf** at `Corel\CorelDRAW Graphics Suite 13\Programs\` (which typically installs to the Windows **Program Files** folder).

If you want information about the Corel PHOTO-PAINT object model, you can view a diagram of it at `Corel\CorelDRAW Graphics Suite 13\Programs\PP VBA Object Model.pdf` (which typically installs to the Windows **Program Files** folder).

Working with documents

Whenever a CorelDRAW file is opened, a new `Document` object is created in the `Application` object for that document. The `Application` object contains a `Documents` collection, which provides access to all of the open documents. The order of the documents in the collection is set to the order in which the documents were created and opened.

You can browse the object model for the complete list of `Document` objects, but here is a list of the most useful ones (some of which are explained in greater detail in this chapter):



Document Member	Description
Activate	Activates the given document, bringing it to the front of the pile in CorelDRAW. ActiveDocument is set to reference it.
ActiveLayer	Represents the layer that is set as active in the Object Manager
ActivePage	Represents the active page in the document — that is, the one that is being edited in CorelDRAW
AddPages AddPagesEx	Adds pages to the end of a document
BeginCommandGroup EndCommandGroup	Creates a “command group” — that is, a series of actions that appear as a single item on the Undo list
ClearSelection	Clears the document’s selection so as to deselect all shapes in the document
Close	Closes the document
Export	Performs a simple export from the document
ExportEx	Performs a highly configurable export from the document
ExportBitmap	Performs an export to a bitmap with full control
FileName	Gets the filename
FilePath	Gets the path to the file
FullFileName	Gets the full path and filename of the document
GetUserArea	Lets you add interactivity to a macro by allowing the user to drag an area
GetUserClick	Lets you add interactivity to a macro by allowing the user to click
InsertPages InsertPagesEx	Inserts pages into the document at a specified location
Pages	Provides access to the Pages collection
PrintOut PrintSettings	Prints the document using the document’s print settings
PublishToPDF PDFSettings	Publishes the document to Adobe Acrobat Reader (PDF) format
ReferencePoint	Gets/sets the reference point used by many Shape functions (such as the ones for transforming a shape or getting a shape’s position)
Save	Saves the document using the current filename
SaveAs	Saves the document to a new filename or with new settings
Selection	Gets the selection as a Shape
SelectionRange	Gets the selection as a ShapeRange



Document Member	Description
Unit WorldScale	Sets the document units used by functions that take a measurement value, such as functions related to size and position. This is independent of the units that the rulers are set to use. Also gets/sets the drawing scale. This changes the value in the document; however, it must be explicitly calculated into functions that take a measurement value, which use 1:1 by default.

The `Application` object also contains all `Window` objects.

`Document` and `Window` objects have many member properties and methods for performing actions such as the following:

- Creating documents
- Opening documents
- Importing files into documents
- Switching between documents
- Viewing documents
- Changing content in documents
- Setting the Undo string for documents
- Exporting files from documents
- Printing documents
- Publishing documents to PDF
- Closing documents

Creating documents

The `Application` object has two methods for creating new documents.

The first function creates a new, empty document based on the default page size, orientation, and styles:

```
Application.CreateDocument() As Document
```

The second function creates a new, untitled document from a specified `CorelDRAW (.CDT)` template:

```
Application.CreateDocumentFromTemplate(Template As String, _  
[IncludeGraphics As Boolean = True]) As Document
```

The new document becomes active immediately, so `ActiveDocument` references the new document (rather than the old one).

Both of these functions return a reference to the new document, and so they are typically used in the following manner:

```
Dim newDoc as Document  
Set newDoc = CreateDocument
```



The `Document` class does not have a method for creating an instance (object) of itself; only the `Application` object can create documents.



Opening documents

To open a document, you can use the `OpenDocument` member function of the global `Application` object:

```
Dim doc As Document
Set doc = OpenDocument("C:\graphic1.cdr")
```

Importing files into documents

Files of all supported formats can be imported into CorelDRAW. However, files are imported onto layers, so for information on importing files, see “Importing files into layers” on page 70.

Switching between documents

The `ActiveDocument` property provides direct access to the active document — that is, the document that’s in front of all the other documents in the CorelDRAW window. `ActiveDocument` is an object of type `Document` and, therefore, has all of the same members — properties, objects, and methods — as the `Document` class.

If there are no open documents, `ActiveDocument` returns `Nothing`. You should test for this with the following code:

```
If Documents.Count = 0 Then
    MsgBox "There aren't any open documents.", vbOK, "No Docs"
Exit Sub
End If
```

The `Document` class also has a method, `Activate`, which activates the document and brings it to the front of all the open documents in CorelDRAW such that `ActiveDocument` references the document that was activated. The following code sample activates the third open document (providing that three or more documents are open in CorelDRAW):

```
Documents(3).Activate
```



Using the `Activate` method on the `ActiveDocument` property has no effect.

If you want, you can test the `FilePath`, `FileName`, and `FullFileName` properties of the document to activate the correct one:

- `FilePath` — for testing only the path of the file (for example, `C:\My Documents\`)
- `FileName` — for testing only the name of the file (for example, `Graphic1.des`)
- `FullFileName` — for testing both the full path and name (for example, `C:\My Documents\Graphic1.des`)



You can test the name by using the following code:

```
Public Function findDocument(filename As String) As Document
    Dim doc As Document
    For Each doc In Documents
        If doc.FileName = filename Then Exit For
        Set doc = Nothing
    Next doc
    Set findDocument = doc
End Function
```

You can then call the returned document's `Activate` method.



The documents in the `Documents` collection are sequenced in the order that they were created and opened. To reflect the current stacking order of the documents in CorelDRAW, you must use the `Windows` collection.

Viewing documents

In CorelDRAW, the user can simultaneously display several windows for viewing a single document. For a large document, one window might be zoomed in to the upper-right corner and another zoomed in to the lower-right corner. Although the individual windows can be zoomed and panned independently, turning the page in one window turns the page in them all.

By using the View Manager, the user can also save individual zoom and display settings. Selecting a view then displays that location on the page.

In VBA, the main differences between the `Window` object and the `View` object is that the `Window` object provides access to the windows that contain each `View` of the document. The `Window` is the frame, while the `View` is display of the document that's inside it.

Working with windows

Each `Document` object has a `Windows` collection for viewing the document. To switch between windows, use a window's `Activate` method:

```
ActiveDocument.Windows(2).Activate
```

The next and previous windows for the current document are referenced in a window's `Next` and `Previous` properties:

```
ActiveWindow.Next.Activate
```

To create a new window, call the `NewWindow` member function of a `Window` object:

```
ActiveWindow.NewWindow
```

To close a window, call its `Close` member function. If it is the document's last window, the document is also closed:

```
ActiveWindow.Close
```



Working with views

The Views and ActiveView members are, literally, views of the document. The difference between them is that each Window object has just one ActiveView member (which is the current view onto the document), whereas a View object is just the recorded memory of one particular ActiveView member (such that re-activating that View object zooms and pans the Window object's ActiveView member to the location and zoom setting that is stored in the properties of the View object).

Each Window object has an ActiveView member. Each Document object has a collection of View objects in its Views member.



The only way to access an ActiveView member is from a Window object's ActiveView property.

You can create a new View object and add it to the Document object's Views collection from within the collection itself. The following code adds the current ActiveView settings to the Views collection:

```
ActiveDocument.Views.AddActiveView "New View"
```

You can also create a new view with specific settings by using the CreateView member function of the Document object. The following code creates a new View object at the position (3, 4) in inches, using a zoom factor of 95%, and displaying page 6:

```
ActiveDocument.Unit = cdrInch  
ActiveDocument.CreateView "New View 2", 3, 4, 95, 6
```

To restore a view, call that view's Activate member function. The document's active window is accordingly set to that view:

```
ActiveDocument.Views("New View").Activate
```

Zooming

To zoom in to a set amount, set the Zoom property of the ActiveView member. The zoom is set as a double value in percent. For example, the following code sets the zoom factor to 200%:

```
ActiveWindow.ActiveView.Zoom = 200.0
```

You can also zoom the ActiveView member with various member functions: ToFitAllObjects, ToFitArea, ToFitPage, ToFitPageHeight, ToFitPageWidth, ToFitSelection, ToFitShape, ToFitShapeRange, and SetActualSize.

Panning

To pan the ActiveView member, move its origin. This can easily be done by modifying the OriginX and OriginY properties of the ActiveView member. The following code pans the document 5 inches to the left and 3 inches up:

```
Dim av As ActiveView  
ActiveDocument.Unit = cdrInch  
Set av = ActiveWindow.ActiveView  
av.OriginX = av.OriginX - 5  
av.OriginY = av.OriginY + 3
```



You can also use the member function `SetViewPoint`:

```
Dim av As ActiveView
ActiveDocument.Unit = cdrInch
Set av = ActiveWindow.ActiveView
av.SetViewPoint av.OriginX - 5, av.OriginY + 3
```

Changing content in documents

You can modify content documents regardless of whether they are active. For example, if you have a reference to a document, you can add a new layer called `fooLayer` by using the following code:

```
Dim doc As Document
Set doc = Documents(3)
doc.ActivePage.CreateLayer "fooLayer"
```

If you want to create the new layer in an inactive document whose name you know (in the following example, it is `barDoc.cdr`), you might use the following code to call the function `findDocument()`:

```
Dim doc As Document
Set doc = findDocument("barDoc.cdr")
If Not doc Is Nothing Then doc.ActivePage.CreateLayer "fooLayer"
```



Modifying content in an inactive document does not activate that document. To activate a document, call its `Activate` method.

Setting the Undo string for documents

Two very useful member functions of the `Document` object allow any number of programmed CorelDRAW actions to appear as a single action on the undo list. These methods are `BeginCommandGroup()` and `EndCommandGroup()`, as in the following code sample:

```
Dim sh As Shape
ActiveDocument.BeginCommandGroup "CreateCurveEllipse"
Set sh = ActiveLayer.CreateEllipse(0, 1, 1, 0)
sh.ConvertToCurves
ActiveDocument.EndCommandGroup
```

After running this code, the **Undo** string on the **Edit** menu displays **Undo CreateCurveEllipse**, and clicking **Undo** undoes not only the `ConvertToCurves` operation but also the `CreateEllipse` operation.

A command group can contain many hundreds of commands, if required. This helps to make your macros appear to be fully integrated into CorelDRAW.

Exporting files from documents

Files of all supported formats can be exported from CorelDRAW.



Files are exported from the Document object — not from Layer objects — because the range of shapes exported often extends over multiple layers (or even over multiple pages). The Document object has three export functions — Export, ExportEx, and ExportBitmap — all of which can be used to export to bitmap or vector format.

To export a page, you require only a filename and a filter type. The following code exports the current page to a TIFF bitmap file:

```
ActiveDocument.Export "C:\ThisPage.eps", cdrTIFF
```

However, this coding gives little control over the output of the image. More control is obtained by including a StructExportOptions object, as in the following code:

```
Dim expOpts As New StructExportOptions
expOpts.ImageType = cdrCMYKColorImage
expOpts.AntiAliasingType = cdrNormalAntiAliasing
expOpts.ResolutionX = 72
expOpts.ResolutionY = 72
expOpts.SizeX = 210
expOpts.SizeY = 297
ActiveDocument.Export "C:\ThisPage.eps", cdrTIFF, cdrCurrentPage, expOpts
```

A StructPaletteOptions object can also be included in the function call for palette-based image formats, so as to provide the settings for auto-generating the palette.

The ExportEx function is the same as the Export function, except that it can access the filter's dialog box to retrieve its settings, and then export the file:

```
Dim eFilt As ExportFilter
Set eFilt = ActiveDocument.ExportEx("C:\ThisPage.eps", cdrEPS)
If eFilt.HasDialog = True Then
    If eFilt.ShowDialog = True Then
        eFilt.Finish
    End If
Else
    eFilt.Finish
End If
```

The third function, ExportBitmap, is similar to ExportEx in that it returns an ExportFilter object that can be used to display the Export dialog box. However, this function takes the individual members of the StructExportOptions object as parameters, thereby simplifying using the function:

```
Dim eFilt As ExportFilter
Set eFilt = ActiveDocument.ExportBitmap("C:\Selection.eps", _
    cdrTIFF, cdrSelection, cdrCMYKColorImage, _
    210, 297, 72, 72, cdrNormalAntiAliasing, _
    False, True, False, cdrCompressionLZW)
eFilt.Finish
```



Printing documents

Printing documents with VBA is straightforward: almost all settings that are available in the CorelDRAW **Print** dialog box are available as properties of the Document object's **PrintSettings** member. When the properties are set, printing the document is simply a matter of calling the document's **PrintOut** member function.

For example, the following code prints 3 copies of pages 1, 3, and 4, to a level 3 PostScript® printer:

```
With ActiveDocument.PrintSettings
    .Copies = 3
    .PrintRange = prnPageRange
    .PageRange = "1, 3-4"
    .Options.PrintJobInfo = True
With .PostScript
    .DownloadType1 = True
    .Level = prnPSLevel3
End With
End With
ActiveDocument.PrintOut
```

For each page in the CorelDRAW **Print** dialog box, there is a corresponding object in the object model. The following table lists the objects that correspond to each page in the **Print** dialog box:

Options page in dialog box	Member of PrintSettings object
General	Properties of PrintSettings
Layout	Properties of PrintSettings
Prepress	Prepress
PostScript	PostScript
Misc	Options

Each object contains all of the properties from the corresponding page of the **Print** dialog box. The only print options that cannot be set in VBA are the layout options — however, you can, if necessary, launch the **Print** dialog box by using the **PrintSettings** object's **ShowDialog** member function.

To reset the print settings, call the **PrintSettings** object's **Reset** member function:

```
ActiveDocument.PrintSettings.Reset
```

Your code can also access any printing profiles that have been saved by the user from the **Print** dialog box by using the **PrintSettings** object's **Load** member function:

```
ActiveDocument.PrintSettings.Load "C:\Corel DRAW Defaults.prs"
```

Note that this function requires a full path to the printing profile.

You can save printing profiles by using the **Save** member function.



To print only the selected shapes, set `Document.PrintSettings.PrintRange` to `prnSelection`. To select a specific printer, set `Document.PrintSettings.Printer` to refer to the appropriate printer in the collection `Application.Printers`.

Publishing documents to PDF

Publishing to Adobe Acrobat Reader (PDF) format is a two-stage process. The first step is to specify the PDF settings, but this step can be skipped if the user specifies the document's settings from within CorelDRAW or chooses to use the default settings. The second step is to export the file.

To set the PDF settings, modify the properties of the `Document` object's `PDFSettings` member. This member is an object of type `PDFVBASettings` and has properties for all of the PDF settings that can be set in the CorelDRAW **PublishToPDF** dialog box.

The following code exports pages 2, 3, and 5 to a PDF file called `MyPDF.pdf`:

```
Dim doc As Document
Set doc = ActiveDocument
With doc.PDFSettings
    .Author = "Corel Corporation"
    .Bookmarks = True
    .ColorMode = pdfRGB
    .ComplexFillsAsBitmaps = False
    .CompressText = True
    .DownsampleGray = True
    .EmbedBaseFonts = True
    .EmbedFonts = True
    .Hyperlinks = True
    .Keywords = "Test, Example, Corel, CorelDRAW, PublishToPDF"
    .Linearize = True
    .PageRange = "2-3, 5"
    .pdfVersion = pdfVersion13
    .PublishRange = pdfPageRange
    .TrueTypeToType1 = True
End With
doc.PublishToPDF "C:\MyPDF.pdf"
```

You can give more control to the user by using the following code to display the **PDF Settings** dialog box:

```
Dim doc As Document
Set doc = ActiveDocument
If doc.PDFSettings.ShowDialog = True Then
    doc.PublishToPDF "C:\MyPDF.pdf"
End If
```



Profiles for PDF settings can be saved and loaded by using the **Save** and **Load** member functions of the **PDFSettings** object.

Closing documents

To close a document, call its **Close** method:

```
ActiveDocument.Close
```

In the above code, the active document is closed and the document that was behind it in **CorelDRAW** becomes the new active document. If the code closes a document that is not the active document, the document referenced by **ActiveDocument** does not change.



You must explicitly test the document's **Dirty** property and take appropriate action if the document has changed.

You can also close a document by using the **Close** member function of the **Document** object itself:

```
doc.Close
```

Working with pages

Each page in a document is a member of the **Document** object's **Pages** collection.

The pages are sequenced in the **Pages** collection in the same order that they appear in the document; for example, the fifth page in the document is the same page as **ActiveDocument.Pages.Item(5)**. If the pages are reordered in the **Page Sorter** view, or if pages are added or deleted, the **Pages** collection is immediately updated to reflect the new order of pages in the document.

This section shows you how to carry out the following actions:

- Creating pages
- Switching between pages
- Reordering pages
- Resizing pages
- Deleting pages

Creating pages

The member function for creating pages is actually a member of the **Document** class, not of the **Page** class.

Document Member	Description
Document.AddPages()	Adds a specified number of pages at the default size to the end of the document
Document.AddPagesEx()	Adds a specified number of pages at the specified size to the end of the document
Document.InsertPages()	Inserts pages at the default size at a specified position in the document
Document.InsertPagesEx()	Inserts pages at the given size at a specified position in the document



As an example, the following function adds 3 default-sized pages to the end of the document:

```
Public Function AddSomeSimplePages() as Page
    Set AddSomeSimplePages = ActiveDocument.AddPages(3)
End Function
```

The following sample function adds 3 pages that are 8.5 by 11 inches in size to the end of the document

```
Public Function AddSomeSpecifiedPages() as Page
    Dim doc as Document
    Set doc = ActiveDocument
    doc.Unit = cdrInch
    Set AddSomeSpecifiedPages = doc.AddPagesEx(3, 8.5, 11)
End Function
```

Both of these member functions return the first page that was added. Therefore, if you know the number of pages created, you can access any of them because they are the ones after the page returned in the document's Pages collection.

You can use the returned page's Index property to find the right place in the Pages collection, and then you can increment that to access the subsequent pages that were added:

```
Dim firstNewPage As Page, secondNewPage As Page
Set firstNewPage = AddSomeSimplePages
Set secondNewPage = ActiveDocument.Pages(firstNewPage.Index + 1)
```

Switching between pages

To access the active page of the active document, use `Application.ActivePage`, `ActiveDocument.ActivePage`, or simply `ActivePage`. These return a reference to the active page in the active document, of type `Page`:

```
Dim pg As Page
Set pg = ActivePage
```

To access the active page of any document (regardless of whether it is active), use the property `Document.ActivePage` of the given document:

```
Public Function getDocsActivePage(doc As Document) As Page
    Set getDocsActivePage = doc.ActivePage
End Function
```

To switch between pages, find the page that you want to access, and then invoke its `Activate` member function. The following code activates page 3 in a CorelDRAW document:

```
ActiveDocument.Pages(3).Activate
```



It is not necessary to activate a page to make changes to it. By explicitly referencing the page you want to edit, you can make those changes without activating the page. The following code deletes all of the shapes on page 3 of the active document without activating that page:

```
Public Sub DeleteShapesFromPage3()  
    Dim doc As Document  
    Set doc = ActiveDocument  
    doc.Pages(3).Shapes.All.Delete  
End Sub
```



Activating a page in an inactive document does not activate that document. Use the `Document.Activate` method to activate the document.

Reordering pages

Individual pages can be moved around within the document by using the `MoveTo` member function of each of the pages. The following code moves page 2 to the position of page 4:

```
ActiveDocument.Pages(2).MoveTo 4
```



Activating a page in an inactive document does not activate that document. Use the `Document.Activate` method to activate the document.

Resizing pages

You can resize pages and set their orientation, set the default page size, and use defined page sizes.

Resizing pages and setting their orientation

Pages can be individually resized by using the `SetSize` member function of the `Page` class. This function takes two size values (width and height) and applies them to the page. The following code changes the size of the active page in the active document to A4:

```
ActiveDocument.Unit = cdrMillimeter  
ActivePage.SetSize 210, 297  
ActivePage.Orientation = cdrLandscape
```



For the `SetSize` method, the first number is always the page width and the second number is always the page height. Reversing the two numbers causes CorelDRAW to switch the page's orientation.



Setting the default page size

To set the default page size for the document, set the value of the item in the document's Pages collection with index 0:

```
Dim doc As Document
Set doc = ActiveDocument
doc.Unit = cdrMillimeter
doc.Pages(0).SetSize 297, 210
```

Alternatively, you can use a shortcut property MasterPage of the Document object:

```
Dim doc As Document
Set doc = ActiveDocument
doc.Unit = cdrMillimeter
doc.MasterPage.SetSize 297, 210
```

Using the defined page sizes

All of the page sizes that are defined (either by CorelDRAW or by the user) are stored in the PageSizes collection of the Application class. You can get all of the names of the page sizes by parsing this collection and getting each PageSize object's Name property:

```
Dim pageSizeName As String
pageSizeName = Application.PageSizes(3).Name
```

Page sizes can be specified by using their name. For example, the following code gets the PageSize called Business Card:

```
Dim thisSize As PageSize
Set thisSize = Application.PageSizes("Business Card")
```

You can get the actual dimensions of each PageSize object using the Width and Height properties. The following code retrieves the Width and Height (in millimeters) of PageSize object 3:

```
Dim pageWidth As Double, pageHeight As Double
Application.Unit = cdrMillimeter
pageWidth = Application.PageSizes(3).Width
pageHeight = Application.PageSizes(3).Height
```

PageSize objects have a Delete member function, which can be used only on user-defined page sizes. To determine whether a PageSize is user-defined, test its BuiltIn Boolean property:

```
Public Sub deletePageSize(thisSize As PageSize)
    If Not thisSize.BuiltIn Then thisSize.Delete
End Sub
```



If you need the page size in a particular unit of measurement, always set the document's units before getting the width and height.



Deleting pages

Pages can be deleted by naming each page's `Delete` member function as follows:

```
ActivePage.Delete
```

This deletes all of the shapes that exist on that page and deletes the page from the document's `Pages` collection. The collection is immediately updated to reflect the change. `Delete` must be called individually for each page that you want to delete.

You cannot delete all of the pages in a document. By using the following code, you can avoid trying to delete the last remaining page in a document:

```
If ActiveDocument.Pages.Count > 1 Then ActivePage.Delete
```

Working with layers

All of the layers in the document are contained by the document's `Page` objects.

In CorelDRAW, all visible shapes are contained on layers within each page. Although one layer can have different properties on each page, all pages have the same layers and all those layers have the same names on each page.

If you want to create new shapes in CorelDRAW by using VBA, you must use the shape-creation member functions of the `Layer` class. For information, see "Working with shapes" on page 71.

This section describes how to perform the following tasks:

- Creating layers
- Activating layers
- Locking and hiding layers
- Reordering layers
- Renaming layers
- Importing files into layers
- Deleting layers

Creating layers

To create a layer, use the `Page` class's `CreateLayer` member function. The following code creates a new layer called `My New Layer`:

```
ActivePage.CreateLayer "My New Layer"
```

A new layer is always positioned at the top of the list of non-master layers.

Activating layers

To activate a layer, call that layer's `Activate` member function:

```
ActivePage.Layers("Layer 1").Activate
```

This makes the layer active but does not enable the layer or make it visible if it is already locked or hidden.



Locking and hiding layers

Layer objects have the properties `Enabled` and `Visible` that control (respectively) whether you can edit the layer and whether its contents are visible in CorelDRAW. Both properties are Boolean. By setting both the properties to `True`, you unlock and display the layer for editing. By setting either property to `False`, however, you lock the layer such that it cannot be edited.

The following sample code locks, but displays, the layer on the active page:

```
ActivePage.Layers("Layer 1").Visible = True
ActivePage.Layers("Layer 1").Editable = False
```

The result of any changes to these properties is immediately displayed in the CorelDRAW Object Manager.

The preceding sample code affects only the active page. You can set the layer settings for a given page by specifying a page from the `Pages` collection, or by referencing the `ActivePage`. To make the changes to all of the pages in the document, use the property `MasterPage` of the `Document`:

```
ActiveDocument.MasterPage.Layers("Layer 1").Visible = True
```

Reordering layers

You can use VBA to reorder layers. The `Layer` class has two member functions: `MoveAbove` and `MoveBelow`. Both methods take a `Layer` object as the only parameter, so the layer is moved above or below this layer.

The following code moves the layer called `Layer 1` to immediately below the layer `Guides`:

```
Dim pageLayers As Layers
Set pageLayers = ActivePage.Layers
pageLayers("Layer 1").MoveBelow pageLayers("Guides")
```

The change is immediately reflected in the CorelDRAW Object Manager (although sometimes the effects are apparent only in the Layer Manager).

Renaming layers

Layers can be renamed by editing the `Name` property of the layer.

The following code renames `Layer 1`:

```
ActivePage.Layers("Layer 1").Name = "Layer with a New Name"
```

Importing files into layers

Files of all supported formats can be imported into CorelDRAW.

Files are imported onto layers; therefore, the `Import` and `ImportEx` functions are members of the `Layer` object.

The following code imports the file `C:\logotype.gif` onto the active layer at the center of the page:

```
ActiveLayer.Import "C:\logotype.gif"
```

When importing, any shapes that were previously selected are deselected, and the contents of the imported file are selected.



To reposition or resize the imported shapes, get the document's selection:

```
ActiveDocument.Unit = cdrInch  
ActiveSelection.SetSize 3, 2
```

Some file formats — notably Encapsulated PostScript (EPS) and PDF — can be imported by using one of two filters. Using the EPS filter, you can import the EPS file as a placeable object that can be printed but not modified; you can also interpret the PostScript portion of the EPS file, importing the actual artwork from within the EPS rather than just the low-resolution TIFF or WMF placeable header (which cannot be as easily edited). To specify which filter to use, include the optional parameter `Filter`:

```
ActiveLayer.Import "C:\map.eps", cdrPSInterpreted
```

The `ImportEx` function provides much better control over the import filter through its optional use of a `StructImportOptions` object. The following code imports the file as a linked file:

```
Dim iFilt As ImportFilter  
Dim importProps As New StructImportOptions  
importProps.LinkBitmapExternally = True  
Set iFilt = ActiveLayer.ImportEx("C:\world-map.epsf", cdrAutoSense, importProps)  
iFilt.Finish
```

Deleting layers

Layers can be deleted by calling the `Layer` class's `Delete` function. The `Layers` collection can be accessed only from the `Page` object. Calling the `Delete` function removes the layer completely from the document, deleting all of the shapes on that layer on all of the pages in the document.

The following code deletes the layer called `Layer 1`:

```
ActivePage.Layers("Layer 1").Delete
```

Working with shapes

All of the shapes in a document are contained by the document's `Layer` objects.

Objects of type `Shape` are the actual shapes that exist in the CorelDRAW document. If you change a shape's properties in the document (such as by moving the shape, changing its size, or giving it a new fill), these changes are immediately visible to the VBA object model.

`Shape` objects exist as members of `Layer` objects. Each `Layer` object contains a collection, `Shapes`, which contains all of the shapes on the layer. The first item of this collection is the shape at the top of the layer (in other words, the one that is above all the others), and the last item is the shape at the bottom; if you reorder the shapes on the page, the collection is updated to reflect the change.

Each `Page` object in the document also has a collection, `Shapes`, which contains all of the `Shapes` collections on all the layers (including any master layers) on the page. The first shape in the collection is the shape at the very top of the page, and the last shape is the shape at the bottom.



This section describes how to perform the following tasks:

- Creating shapes
- Selecting shapes
- Determining shape type
- Changing shape properties
- Coloring shapes
- Duplicating shapes
- Applying effects to shapes

Creating shapes

Shape objects represent the shapes that you create in a CorelDRAW document by using the drawing tools. Among the shapes you can create are rectangles, ellipses, curves, and text objects.

Because each Shape object is a member of the Shapes collection, which is a member of one of the Layer objects on the Page, the methods for creating new shapes belong to the Layer class, and they all begin with the word `Create`.

Creating rectangles

There are two functions for creating new rectangle shapes — `CreateRectangle` and `CreateRectangle2` — both of which return a reference to the new Shape object. The two functions differ only in the parameters that they take.

For example, the following code, which uses `CreateRectangle`, creates a simple two-by-one-inch rectangle positioned six inches up from the bottom and three inches in from the left of the page:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
Set sh = ActiveLayer.CreateRectangle(3, 7, 6, 5)
```

The parameters are given as left, top, right, bottom and they are measured in the document's units (which can be explicitly set before creating the shape).

The alternative method, `CreateRectangle2`, creates the rectangle by specifying the coordinates of its lower-left corner and its width and height. The following code creates the same rectangle as above:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
Set sh = ActiveLayer.CreateRectangle2(3, 6, 2, 1)
```

The alternative methods are provided to make it simpler to develop solutions; they provide identical functionality.

Round-cornered rectangles can also be created by using the `CreateRectangle` and `CreateRectangle2` methods. Both functions have four optional parameters that set the roundness of the corners when the rectangle is created, but these values have slightly different meanings for the two functions.



The four optional parameters of the method `CreateRectangle` take integer values in the range 0 to 100 (with zero being the default). These values define the radius of the four corners as a whole-number percentage of half the shortest side length. The following code re-creates the two-by-one-inch rectangle from earlier, but the four corner radii are set to 100%, 75%, 50%, and 0% of half the shortest side; in other words, the radii will be 0.5 inches, 0.375 inches, 0.25 inches, and a cusp:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
Set sh = ActiveLayer.CreateRectangle(3, 7, 6, 5, 100, 75, 50, 0)
```

The four parameters define the radii of the corners in the order upper-left, upper-right, lower-left, lower-right.

The method `CreateRectangle2` defines the corner radii in the same order, except that it takes double (floating-point) values that are the radius measurements in the document's units. The following code creates the same round-cornered rectangle:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
ActiveDocument.ReferencePoint = cdrBottomLeft
Set sh = ActiveLayer.CreateRectangle2(3, 6, 2, 1, 0.5, 0.375, 0.25, 0)
```



You must limit the radii passed to the method `CreateRectangle2` to less than half the shorter dimension of the rectangle.

Creating ellipses

There are two methods for creating ellipses: `CreateEllipse` and `CreateEllipse2`. They differ in the parameters that they take, so you can create an ellipse based on its bounding box or instead on its center point and radius. Both functions also create arcs or partial ellipses, or segments or pie slices.

The `CreateEllipse` method takes four parameters that define its bounding box in the same way as for `CreateRectangle` — in other words, left, top, right, bottom, in the document's units. The following code creates a 50-millimeter circle:

```
Dim sh As Shape
ActiveDocument.Unit = cdrMillimeter
Set sh = ActiveLayer.CreateEllipse(75, 150, 125, 100)
```

To create an arc or a segment, three additional parameters are required: start angle, end angle, and a Boolean value that defines whether it is an arc (`False`) or a segment (`True`). Angles are measured with zero being horizontally right on the page and positive values being degrees from zero moving counterclockwise. The arc or pie is drawn from the start angle to the end angle. The following code creates a letter “C” shape:

```
Dim sh As Shape
ActiveDocument.Unit = cdrMillimeter
Set sh = ActiveLayer.CreateEllipse(75, 150, 125, 100, 60, 290, False)
```



The `CreateEllipse2` method creates ellipses based on their center points, and on horizontal and vertical radii. (If only one radius is given, a circle is created.) The following code creates the same 50-millimeter circle as the preceding code:

```
Dim sh As Shape
ActiveDocument.Unit = cdrMillimeter
Set sh = ActiveLayer.CreateEllipse2(100, 125, 25)
```

To create an ellipse, provide a second radius. (The first radius is the horizontal radius, the second is the vertical radius.)

```
Dim sh As Shape
ActiveDocument.Unit = cdrMillimeter
Set sh = ActiveLayer.CreateEllipse2(100, 125, 50, 25)
```

To create an arc or a segment, use the same additional parameters as for the `CreateEllipse` method.

Creating curves

Curves are made from several other objects: each `Curve` has one or more `SubPath` member objects, each `SubPath` has one or more `Segment` objects, and each `Segment` has two `Node` objects as well as two control-point position/angle properties.

You can create a curve object in CorelDRAW by using the `CreateCurve` method of the `Application` object.

Create a new `SubPath` inside the `Curve` object by using the `CreateSubPath` member function. This creates the first `Node` of the curve.

Next, append a new line- or curve-type `Segment` to the `SubPath` by using the `AppendLineSegment` and `AppendCurveSegment` member functions. This adds another `Node` and sets the positions of the control handles for that `Segment`. Repeat this as often as required to build up the `Curve`.

Create the curve shape on the `Layer` by using the `CreateCurve` member function.

You can add additional `SubPaths` to the `Curve` and build those up with their own `Nodes`. You can also close a `Curve` by setting its `Closed` property to `True`.

The following code creates a D-shaped closed curve:

```
Dim sh As Shape, spath As SubPath, crv As Curve
ActiveDocument.Unit = cdrCentimeter
Set crv = Application.CreateCurve(ActiveDocument) 'Create Curve object
Set spath = crv.CreateSubPath(6, 6) ' Create a SubPath
spath.AppendLineSegment 6, 3 ' Add the short vertical segment
spath.AppendCurveSegment 3, 0, 2, 270, 2, 0 ' Lower curve
spath.AppendLineSegment 0, 0 ' Bottom straight edge
spath.AppendLineSegment 0, 9 ' Left straight edge
spath.AppendLineSegment 3, 9 ' Top straight edge
spath.AppendCurveSegment 6, 6, 2, 0, 2, 90 ' Upper curve
spath.Closed = True ' Close the curve
Set sh = ActiveLayer.CreateCurve(crv) ' Create curve shape
```



The `AppendLineSegment` method requires only a single node position for the end of the segment. However, the `AppendCurveSegment` method requires one Cartesian coordinate and two polar coordinates; the Cartesian coordinate is for the end node, while the polar coordinates are for the two control handles where the first parameter is the length of the handle and the second parameter is the control handle's angle. Alternatively, you can use the `AppendCurveSegment2` method to specify the coordinates of both control handles.

The `Layer` class has three additional member functions for creating curve objects: `CreateLineSegment`, `CreateCurveSegment`, and `CreateCurveSegment2`. These functions are shortcuts for creating a `Curve` shape along with its first `Segment` on the first `SubPath`, all in a single function.



The new `Segments` are appended to the last `Node` of the `SubPath`. There is an optional parameter that causes the `Segment` to be appended to the first `Node` of the `SubPath`.

Creating text objects

Text objects are another type of `Shape` object. However, handling text is more complicated than handling other shapes.

In `CorelDRAW`, text is manipulated as a `TextRange` object. Text ranges can be accessed by using any of the following properties of `Shape.Text`:

- **Frames** collection — In this scenario, each `TextFrame` object's default member is a `TextRange` object, which is the text within the frame.
- **Story** — In this scenario, the `TextRange` object contains all of the text in all of the frames linked to the current `Shape` object on all of the pages in the document.

The text in a `TextRange` object can be manipulated as a single block of text, and its properties (such as font, size, and style) can be set simultaneously. Alternatively, the `TextRange` object has several properties that are collections of smaller text ranges: `Columns`, `Paragraphs`, `Lines`, `Words`, and `Characters`.

You can create two kinds of text: artistic text and paragraph text.

To create a new artistic text shape, use the member function `CreateArtisticText` of the `Layer` object. The following code creates an artistic text shape of the words `Hello World`, with the left end of the line at 1 inch and the baseline at 4 inches from the origin:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
Set sh = ActiveLayer.CreateArtisticText(1, 4, "Hello World")
```

There are many optional parameters for this function, such that you can setting attributes such as italic, bold, size, and alignment.

To create a new paragraph text shape, use the member function `CreateParagraphText` of the `Layer` object. Paragraph text differs from artistic text in that it flows within a rectangular container, rather than being as wide as necessary before reaching a line feed. The first four parameters of the function are left, top, right, bottom:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
Set sh = ActiveLayer.CreateParagraphText(1, 4, 5, 2, "Hi There", _
    Alignment := cdrCenterAlignment)
```



You can format text. To do this, first get a reference to a `TextRange` object, and then apply the formatting to it. Frames, columns, paragraphs, lines, and the story can all be used to get a reference to a `TextRange` object.

The following code formats the first paragraph of the story into a heading style and the second and third paragraphs into a body-text style:

```
Dim txt As TextRange
' Format the first paragraph
Set txt = ActiveShape.Text.Story.Paragraphs(1)
txt.ChangeCase cdrTextUpperCase
txt.Font = "Verdana"
txt.Size = 18
txt.Bold = True
' Format the second and third paragraphs
Set txt = ActiveShape.Text.Story.Paragraphs(2, 2)
txt.Font = "Times New Roman"
txt.Size = 12
txt.Style = cdrNormalFontStyle
```

All of the formatting options from the **Format Text** dialog box in CorelDRAW can be applied programmatically by using VBA. See the `Text` object in the VBA Object Browser for more information.

You can fit text to a path by using the `Text` object's member function `FitToPath`, which simply attaches a text object to the outline of a shape such that the text flows along the path.

The following code creates a new text object and attaches it to the selected shape:

```
Dim sh As Shape, sPath As Shape
ActiveDocument.Unit = cdrInch
Set sPath = ActiveShape
Set sh = ActiveLayer.CreateArtisticText(1, 4, "Hello World")
sh.Text.FitToPath sPath
```

Paragraph text shapes can be flowed inside closed shapes to form non-rectangular frames. This is done by placing the `Text` object inside the `Shape` object by using the member function `PlaceTextInside` of the `Shape` object. Given that a text shape — artistic or paragraph — is selected in CorelDRAW, the following code creates a 5-inch by 2-inch ellipse and places the selected text inside it:

```
Dim txt As Shape, sh As Shape
ActiveDocument.Unit = cdrInch
Set txt = ActiveShape
Set sh = ActiveLayer.CreateEllipse(0, 2, 5, 0)
sh.PlaceTextInside txt
```



Selecting shapes

To determine whether a Shape is selected, you can test its `Selected` Boolean property:

```
Dim sh As Shape
Set sh = ActivePage.Shapes(1)
If sh.Selected = False Then sh.CreateSelection
```

You can add a Shape to the selection simply by setting its `Selected` property to `True`; this selects the shape without deselecting all the other shapes. To select just one shape without any other shapes, use the `CreateSelection` method, as in the preceding code.

To deselect all the shapes, call the document's `ClearSelection` method:

```
ActiveDocument.ClearSelection
```

To select all the shapes on the page or layer, use the following code:

```
ActivePage.Shapes.All.CreateSelection
```

This calls the `CreateSelection` of the `ShapeRange` object, which is returned by the `All` member function of the `Shapes` collection on the active page. Only those shapes on locked or hidden layers will not be selected.

To get the selection — in other words, to access the shapes that are selected in the document — you have a choice: get a reference to a document's `Selection` object, or instead get a copy of the document's `SelectionRange` property. The difference between these two selection-type objects is that `Selection` is updated whenever the selection changes in the document while `SelectionRange` is a copy of the selection state at the time and is not updated.

Referencing the ActiveSelection object

The shortcut for `ActiveDocument.Selection` is `ActiveSelection`. This returns a Shape object, which is a reference to the document's `Selection` object. Because this is a reference, whenever the selection in the document is changed (either by the user or programmatically), the shapes that this `Selection` object contains reflect the change.

```
Dim sel As Shape
Set sel = ActiveDocument.Selection
```

`ActiveSelection` returns a Shape object of subtype `cdrSelectionShape`. This Shape subtype has a member collection called `Shapes`, which is a collection of all of the selected shapes in the document. The items in the `ActiveSelection.Shapes` collection can be accessed in the normal manner:

```
Dim sh As Shape, shs As Shapes
Set shs = ActiveSelection.Shapes
For Each sh In shs
    sh.Rotate 15 'Rotate each shape thru 15° counterclockwise
Next sh
```

You cannot directly remember and recall the objects that are selected by using `ActiveSelection`. To copy the references to the selected objects, you must populate an array of Shape or a collection of type Shapes, or you must use a `ShapeRange`.



Referencing the ActiveSelectionRange object

ActiveSelectionRange is the shortcut for ActiveDocument.SelectionRange. This is a property of the document of type ShapeRange:

```
Dim selRange As ShapeRange
Set selRange = ActiveDocument.SelectionRange
```

The ShapeRange object returned by ActiveSelectionRange contains a collection of references to the shapes that were selected at the moment when the property was read. Because these references are to the shapes themselves (and not to the selection), if you change the selection, the ShapeRange is not updated.

The shapes referenced by ActiveSelectionRange are held in a ShapeRange object. This collection can be parsed in the usual way:

```
Dim sh As Shape, shRange As ShapeRange
Set shRange = ActiveSelectionRange
For Each sh In shRange
    sh.Skew 15 ' Skew each shape thru 15° counterclockwise
Next sh
```

Because the SelectionRange is a static copy of references to the objects selected at the time, it is an ideal way of remembering and recalling the selection. Also, shapes can be added to and removed from a ShapeRange object, so it is an ideal way of manipulating the selection.

Comparing the ActiveSelection and ActiveSelectionRange objects

To get the selection, you have a choice: you can get a reference to a document's Selection object (of type Shape), which is updated if the selection changes, or you can get a reference to a new ShapeRange object, which is like a collection of references to the objects in the selection at that moment in time and is not updated.

Although ActiveSelection is generally easier to use, the big advantage of ActiveSelectionRange over it is that, even when the selection changes, the ShapeRange content does not change (whereas the ActiveSelection Shapes collection does change). Also, ShapeRange can be duplicated, and it even allows for shapes to be added and removed and for transformations to be applied, all without the ShapeRange being active at the time.

The individual shapes in a ShapeRange can be selected by calling the range's CreateSelection member function:

```
Dim shRange As ShapeRange
Set shRange = ActiveSelectionRange
shRange.Remove 1
shRange.Remove 2
shRange.CreateSelection
```

The above code creates a ShapeRange from the selection and then removes the first and second — actually the third shape of the original range, because it becomes the second shape when you remove the first — shapes from the range; it then creates a new selection from those shapes left in the range (that is, all the shapes of the original selection, minus the two shapes that were removed from the collection).





If you want to add a `ShapeRange` to the current selection rather than replace the selection, use the `AddToSelection` member function of `ShapeRange`.

Parsing selected shapes

Parsing through the selected shapes is done in the same way for both selections and selection ranges.

Here is a code sample for selections:

```
Dim shs As Shapes, sh As Shape
Set shs = ActiveSelection.Shapes
For Each sh In shs
    ' Do something with the shape, sh
Next sh
```

And here is a code sample for selection ranges:

```
Dim sRange As ShapeRange, sh As Shape
Set sRange = ActiveSelectionRange
For Each sh In sRange
    ' Do something with the shape, sh
Next sh
```

However, selection ranges have the advantage that even if the selection subsequently changes, the range is not updated, and so the “memory” of the selection is not lost. Getting a reference to the `ActiveSelection`, though, does not create a copy of the references to the shapes in the selection, but creates a reference to the intrinsic `ActiveSelection` object. This means that if the selection changes, all references to the `ActiveSelection` are also updated, and so the “memory” of the selection is lost.

Ordering selected shapes in a collection

The order of the `Shape` objects in both the `ActiveSelection.Shapes` and `ActiveSelectionRange` collections is the reverse of the order in which the shapes were selected by the user. Consequently, the first shape in both collections is the last shape that the user selected, and the last shape in both collections is the first shape that the user selected. This property of these collections is useful for macros that need to distinguish which shape was selected first or last.

Determining shape type

Each `Shape` object has a property `Type` that returns the shape’s subtype — that is, what type of shape it is (rectangle, ellipse, curve, text, group, and so on). This is a read-only property. Because some types of shapes have member functions and properties that others do not, it is often necessary to test the shape’s type before calling a method that doesn’t necessarily apply to it.



The following sample code tests a shape to determine whether it is text. It then tests to determine whether it is artistic or paragraph text; if it determines it to be artistic text, it rotates it through 10 degrees:

```
Dim sh As Shape
Set sh = ActiveShape
If sh.Type = cdrTextShape Then
    If sh.Text.IsArtisticText = True Then
        sh.Rotate 10
    End If
End If
```

Changing shape properties

You can modify the properties of the shapes you create by using VBA.

Sizing shapes

To get the width or height of a Shape, test its `SizeWidth` or `SizeHeight` properties:

```
Dim width As Double, height As Double
ActiveDocument.Unit = cdrMillimeter
width = ActiveShape.SizeWidth
height = ActiveShape.SizeHeight
```

The returned `Double` value depends on the setting of `ActiveDocument.Unit`, so if you need the size in a specific unit, begin by setting it. The same applies for all of the size properties and methods in the object model.

To get the width and height in a single command, use the member function `GetSize`:

```
Dim width As Double, height As Double
ActiveDocument.Unit = cdrMillimeter
ActiveShape.GetSize width, height
```

To set the size of a Shape, set its `SizeWidth` or `SizeHeight` properties to the new values. The following code sets the size of the active shape to 50 by 70 millimeters:

```
ActiveDocument.Unit = cdrMillimeter
ActiveShape.SizeWidth = 50
ActiveShape.SizeHeight = 70
```

Alternatively, you can use the `SetSize` method, which sets both in one call:

```
ActiveDocument.Unit = cdrMillimeter
ActiveShape.SetSize 50, 70
```



There is an additional method for setting the shape size: `SetSizeEx`. This method takes a center point in addition to the width and height; the resize is performed relative to this point rather than to the shape's center point. For example, the following code resizes the selection to 10 by 8 inches about the point (6, 5) in the document:

```
ActiveDocument.Unit = cdrInch
ActiveSelection.SetSizeEx 6, 5, 10, 8
```



The object `ActiveSelection` refers to a `Shape` object that contains all of the selected shapes in the document.

All of the preceding code refers to the object's size in terms of its vector curves and ignores the fact that any shape with an outline may actually extend beyond the rectangle containing the curves. There is one method that can be used to get the size of the bounding box of the shape, with the option of including the widths of any outlines. This method is `GetBoundingBox`:

```
Dim width As Double, height As Double
Dim posX As Double, posY As Double
ActiveDocument.Unit = cdrInch
ActiveDocument.ReferencePoint = cdrBottomLeft
ActiveShape.GetBoundingBox posX, posY, width, height, True
```

The method's main use is to get the shape's bounding box, including the position of its lower-left corner. The final parameter is a Boolean value that indicates whether to return the bounding box of the shape including its outline (`True`) or excluding it (`False`). The method for setting the shape's bounding box does not include the necessary parameter for doing so in relation to its bounding box, although by using `GetBoundingBox` twice (once including the outline and a second time excluding the outline) it is possible to calculate the size and position of the bounding box of the vector, not including the outline:

```
Public Sub SetBoundingBoxEx(X As Double, Y As Double, _
                           Width As Double, Height As Double)

    Dim sh As Shape
    Dim nowX As Double, nowY As Double
    Dim nowWidth As Double, nowHeight As Double
    Dim nowXol As Double, nowYol As Double
    Dim nowWidthol As Double, nowHeightol As Double
    Dim newX As Double, newY As Double
    Dim newWidth As Double, newHeight As Double
    Dim ratioWidth As Double, ratioHeight As Double
    Set sh = ActiveSelection
    sh.GetBoundingBox nowX, nowY, nowWidth, nowHeight, False
    sh.GetBoundingBox nowXol, nowYol, nowWidthol, nowHeightol, True
    ratioWidth = Width / nowWidthol
    ratioHeight = Height / nowHeightol
    newWidth = nowWidth * ratioWidth
```



```

newHeight = nowHeight * ratioHeight
newX = X + (nowX - nowXol)
newY = Y + (nowY - nowYol)
sh.SetBoundingBox newX, newY, newWidth, newHeight, False, cdrBottomLeft
End Sub

```

Stretching and scaling shapes

Shapes can be stretched and scaled by a proportional amount. The **Shape** object has two member functions — **Stretch** and **StretchEx** — that perform this operation. Both functions take a decimal value for horizontal and vertical stretching, where 1 is 100% (or no change); you cannot use zero, so you must use a very small value instead.

The following code stretches the selection to half its current height and twice its width, about the midpoint of the bottom edge of its bounding box:

```

ActiveDocument.ReferencePoint = cdrBottomMiddle
ActiveSelection.Stretch 2, 0.5

```

The stretch can also be performed about any point on the page by using the **StretchEx** member function. The following code performs the same stretch as above, but about the point (4, 5) on the page in inches:

```

ActiveDocument.Unit = cdrInch
ActiveSelection.StretchEx 4, 5, 2, 0.5

```

Both of the above functions have an optional Boolean parameter that, when **True**, stretches paragraph text characters by the given amount; when it is **False**, only the bounding box of the text is stretched, and the text is re-flowed within the box.

Positioning shapes

The position of a **Shape** object can be determined by using the properties **PositionX** and **PositionY**, and by using the methods **GetPosition** and **GetBoundingBox**. The position of a shape can be specified by setting the properties **PositionX** and **PositionY**, or by using the methods **SetPosition**, **SetSizeEx**, and **SetBoundingBox**.

The following code gets the position of the **ActiveSelection** shape relative to the current **ReferencePoint** property of **ActiveDocument**, which the code explicitly sets to the lower-left corner:

```

Dim posX As Double, posY As Double
ActiveDocument.ReferencePoint = cdrBottomLeft
ActiveSelection.GetPosition posX, posY

```



The above code returns the position of the reference point of the selection without accounting for the widths of any of the selected shapes' outlines. To account for the widths of the outlines, use the function **GetBoundingBox**.



The following code sets the position of the lower-right corner of each selected shape in the active document to (3, 2) in inches:

```
Dim sh As Shape
ActiveDocument.Unit = cdrInch
ActiveDocument.ReferencePoint = cdrBottomRight
For Each sh In ActiveSelection.Shapes
    sh.SetPosition 3, 2
Next sh
```

Rotating shapes

Shapes can be rotated by using the member functions `Rotate` and `RotateEx` of the `Shape` object.

The `Rotate` member function simply rotates the shape by the given angle (in degrees) about the shape's rotation center. The following code rotates the selection by 30 degrees about its center of rotation:

```
ActiveSelection.Rotate 30
```



Positive rotation angles are always degrees counterclockwise.

To find the center of rotation, get the values of the shape's properties `RotationCenterX` and `RotationCenterY`. Changing the values of these properties moves the center of rotation for the next call to this function.

The function `RotateEx` takes additional parameters that specify the center of rotation. This is a quicker, simpler method than setting each of `RotationCenterX` and `RotationCenterY` and then calling `Rotate`. The following code rotates each of the selected shapes by 15 degrees clockwise about each shape's lower-right corner:

```
Dim sh As Shape
ActiveDocument.ReferencePoint = cdrBottomRight
For Each sh In ActiveSelection.Shapes
    sh.RotateEx -15, sh.PositionX, sh.PositionY
Next sh
```

Skewing shapes

Shapes can be skewed by using the member functions `Skew` and `SkewEx` of the `Shape` object. These two functions are similar to the `Rotate` and `RotateEx` member functions, except that they take two angle parameters: the first is horizontal skew (positive values move the top edge to the left and the bottom edge to the right), and the second is vertical skew (positive values move the right edge upwards and the left edge downwards). The horizontal skew is applied before the vertical skew.

The following code skews the selection by 30 degrees horizontally and by 15 degrees vertically:

```
ActiveSelection.Skew 30, 15
```



Skews of angles close to or greater than 90° are not allowed.



Coloring shapes

Color can be applied to both the fill and the outline of an object, as well as to page backgrounds, lenses, and gradient fills.

You can copy one color to another using the method `CopyAssign`:

```
Dim sh As Shape
Set sh = ActiveShape
sh.Outline.Color.CopyAssign sh.Fill.UniformColor
```

You can get the color model of a color object by getting its `Type` property:

```
Dim colType As cdrColorType
colType = ActiveShape.Outline.Color.Type
```

When you have the type of the color (usually one of `cdrColorRGB`, `cdrColorCMYK`, or `cdrColorGray`, but not limited to those), you can get the components of the color, which are as follows:

- for CMYK color — `CMYKCyan`, `CMYKYellow`, `CMYKMagenta`, and `CMYKBlack`
- for RGB color — `RGBRed`, `RGBGreen`, and `RGBBlue`
- for grayscale — `Gray`

To convert a `Color` object's color to a different color model, use the member functions `ConvertToCMYK`, `ConvertToRGB`, `ConvertToGray`, and so on. The color is converted by using the CorelDRAW color management settings. For example, the following code converts the fill to RGB:

```
ActiveShape.Fill.UniformColor.ConvertToRGB
```

To set a new color, use methods such as `CMYKAssign`, `RGBAssign`, and `GrayAssign`. The number of parameters these methods take depend on how many color components are required for that color model. For example, the following code assigns a deep-blue RGB color to the active shape's outline:

```
ActiveShape.Outline.Color.RGBAssign 0, 0, 102
```

Each component's value range depends on the color model.

Some functions in the CorelDRAW object model take a `Color` object as a parameter. To create a new `Color` object, use the VBA keyword `New`, as in the following example:

```
Dim col As New Color
col.RGBAssign 0, 255, 102
ActiveShape.Outline.Color.CopyAssign col
```



The color “none” does not exist. To set the color “none” to an outline or fill, you must actually set the outline- or fill-type to “none”.

Outlining shapes

Every `Shape` object has a property `Outline`, which refers to an `Outline` object. `Outline` objects have such properties as `Type`, `Width`, `Color`, and `Style`.



The property `Type` sets whether the shape has an outline: the shape has an outline if it is set to `cdrOutline`, but it does not have an outline if it is set to `cdrNoOutline`. Setting this property to `cdrOutline` for a shape that does not have an outline gives the shape the document's default outline, while setting this property to `cdrNoOutline` removes the outline from the shape. This is the same as setting `Width` to zero.

The property `Width` sets the width of the outline in the units of the document; to set the width in points, for example, first set the document's `Unit` property to `cdrPoint`. In the following code sample, the outline of the selected shapes is set to 1 millimeter:

```
ActiveDocument.Unit = cdrMillimeter
ActiveSelection.Outline.Width = 1
```

Any shapes whose `Type` property is `cdrNoOutline` has that property changed to `cdrOutline` when the outline color or width is set.

The property `Color` is a color object that defines the color of the outline. Setting the color of the outline automatically sets the `Type` property of the outline to `cdrOutline` and gives the outline the default width:

```
ActiveSelection.Outline.Color.GrayAssign 0 ' Set to black
```

The `Style` property of `Outline` sets the dash properties of the outline. It has four properties, of which only three can be set:

- `DashCount` — sets the number of dashes in the style
- `DashLength` — an array of values that gives the length of each dash, with the dash size determined by the value of `DashCount`
- `GapLength` — an array of values that gives the length of each gap following each dash, with the dash size determined by the value of `DashCount`
- `Index` — a read-only property that gives the index of the outline style in the document's `OutlineStyles` collection, which is presented to the user in the **Outline** dialog box

The values in the arrays `DashLength` and `DashGap` are drawn as multiples of the width of the line; therefore, if `DashLength(1)` is 5 and the line is 0.2 inches in width, the dash's length is 1 inch; if the line's width is changed to 0.1 inches, the dash's length becomes 0.5 inches.

To use one of the application's outline styles, you must reference the style from the `OutlineStyles` collection by giving an index of the style you want to use. The problem with this is that each user's installation of CorelDRAW can have a different collection of outline styles (most notably if the user has modified them), so it is not guaranteed that a given indexed style is the same for all users. To use one of the styles, assign it to the outline:

```
ActiveShape.Outline.Style = Application.OutlineStyles(3)
```



`OutlineStyles.Item(0)` is always a solid line; to set an outline to solid, set it equal to this style.

Outline objects have many other properties, including the following:

- `StartArrow`, `EndArrow` — sets the arrowhead to use on each end of an open curve
- `LineCaps`, `LineJoin` — sets the type of line caps (butt, round, or square) and joins (bevel, miter, or round)
- `NibAngle`, `NibStretch` — sets the shape of the nib used to draw the line
- `BehindFill`, `ScaleWithShape` — draws the outline behind the fill, and scales the outline with the shape

Outline objects also have two methods:

- `ConvertToObject` — converts the outline to an object



-
- **SetProperties** — a single method that can be used to set most of the outline’s properties in a single call, a technique that is much more efficient and much quicker than specifying each property individually when setting the properties of hundreds or thousands of outlines at the same time

Filling shapes

There are many types of fills in CorelDRAW, including uniform fills, fountain fills, PostScript fills, pattern fills, and texture fills, and each type needs to be handled differently.

Only uniform and fountain fills are discussed here.

The read-only **Type** property of a **Fill** object gives the type of the fill, or whether there is no fill. The following code gets the fill type:

```
Dim fillType As cdrFillType
fillType = ActiveShape.Fill.Type
```

The fill type cannot be set with this property; it is set when the fill is created and applied.



To remove any type of fill, use the member function **ApplyNoFill** of the **Fill** object.

Uniform fills consist of a single, solid color. This color is represented by the fill’s property **UniformColor**, which is a **Color** object.

To apply a uniform fill to a shape, use the **ApplyUniformFill** member function of the shape’s **Fill** property:

```
ActiveShape.Fill.ApplyUniformFill CreateRGBColor(255, 0, 0)
```

Unless you are copying the color from an existing **Color** object, you have to create a new **Color** object first and then apply the color to the shape’s **Fill** property.

To change the color of a uniform fill, change the **UniformColor** property of the **Fill** object:

```
ActiveShape.Fill.UniformColor.RGBAssign 0, 0, 102
```

Uniform fills have a **Type** property of **cdrUniformFill**.

Fountain fills are defined by the property **Fountain**, which is a **FountainFill** object. **FountainFill** objects have many properties, including type, angle, and blend type. The most important property, though, is the collection of colors that comprise the fountain fill.

To create a new fountain fill, use the **ApplyFountainFill** member function of the **Fill** object. This creates a simple two-color fountain with basic properties. The following code creates a simple linear fountain fill, from red to yellow, at 30 degrees to the horizontal:

```
Dim startCol As New Color, endCol As New Color
startCol.RGBAssign 255, 0, 0
endCol.RGBAssign 255, 255, 0
ActiveShape.Fill.ApplyFountainFill startCol, endCol, cdrLinearFountainFill, 30
```

All of the parameters for the **ApplyFountainFill** member function are optional, and not all have been used in the preceding code sample. It is possible to set the midpoint, the number of steps, and the color-blend type in the same function call.



After the fountain fill has been created, you can add more colors to the blend by adding new `FountainColor` items to the `Colors` collection, which is a property of `Fill.Fountain`. For example, the following code adds a green color to the `Colors` collection at a position about one-third (33%) of the way from the red:

```
Dim fFill As FountainFill
Set fFill = ActiveShape.Fill.Fountain
fFill.Colors.Add CreateRGBColor(0, 102, 0), 33
```

Individual colors in the fountain can be moved using the `FountainColor.Move` member function. The following code moves the green color from the previous code to a position that is 60% from the red towards the yellow:

```
ActiveShape.Fill.Fountain.Colors(1).Move 60
```



Color positions are integer values in percent, where 0% is the start-color position and 100% is the end-color position.

The number of colors reported by the `Count` property of the `Colors` collection is the number of colors between the start and end colors. So for the fountain fill created above, the value of the `Count` property is 1. The first color in the `Colors` collection is item 0 and is the start color; it cannot be moved, but its color can be changed. The last color in the `Colors` collection is item (`Count + 1`) and is the end color; it also cannot be moved, but its color can be changed. The following code changes the color of the end color from yellow to blue:

```
Dim cols As FountainColors
Set cols = ActiveShape.Fill.Fountain.Colors
cols(cols.Count + 1).Color.RGBAssign 0, 0, 102
```

There are additional properties (not described here) that define edge padding, center position (for conical, radial, and rectangular fountains), and the number of steps used to draw the fill.



Fountain fills have a `Type` property value of `cdrFountainFill`.

Duplicating shapes

To duplicate shapes, you can use the `Duplicate` member function of the `Shape` object:

```
ActiveSelection.Duplicate
```

This function takes two option values that set the offset for the duplicate from the original. The following code positions the duplicate two inches to the right and one inch above the original:

```
ActiveDocument.Unit = cdrInch
ActiveSelection.Duplicate 2, 1
```

Applying effects to shapes

Effects can be applied directly to shapes by using the appropriate member function of the `Shape` object.



Creating blends

The `CreateBlend` member function of the `Shape` object creates a blend between the current `Shape` and the `Shape` in the parameter list. The following code creates a 10-step blend:

```
Dim sh As Shapes, eff As Effect
Set sh = ActiveSelection.Shapes
Set eff = sh(1).CreateBlend(sh(2), 10)
```



The number of shapes in the blend is 12 — the start and end shapes, plus the ten steps created.

There are several optional parameters (not shown) for the `CreateBlend` method that control the acceleration of the blend, as well as set the path along which the blend is created.

The `Effect` object returned by the preceding code is a reference to the blend. By setting the properties of the `Blend` property of the `Effect` object, the blend can be fine-tuned. The `Effect` object itself has a member function `Separate` for separating the `Blend` effect, as well as `Clear` for removing it.

Creating contours

Contours can be created with the `CreateContour` member function of the `Shape` object. The following code creates a 3-step contour at a 5 millimeter spacing:

```
Dim eff As Effect
ActiveDocument.Unit = cdrMillimeter
Set eff = ActiveShape.CreateContour(cdrContourOutside, 5, 3)
```

There are several optional parameters (not shown) for the `CreateContour` method that control the colors and acceleration of the contour shapes.

Creating other effects

The `Shape` object has several other functions for creating effects: `CreateDropShadow`, `CreateEnvelope`, `CreateExtrude`, `CreateLens`, `CreatePerspective`, `CreatePushPullDistortion`, `CreateTwisterDistortion`, and `CreateZipperDistortion`. For more information on these, access the VBA Help system from the Object Browser.





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 CorelDRAW and Corel PHOTO-PAINT





Index

A

activating layers	69
adding	
captions to macros	47
modules to projects	28
tooltips to macros	47
applying effects to shapes	87
arrays	10
definition	89
associating images or icons with macros	48
automatic completion	21
automation	
definition	4, 89

B

bitwise operators	13
Boolean comparison and assignment	13
breakpoints	35
building	
functions	10
subroutines	10
buttons	44
creating for macros	46

C

C and C++	6
Call Stack window	36
captions, adding to macros	47
capturing	
coordinates for macros	50
mouse actions for macros	48
changing	
content in documents	61
shape properties	80

checking syntax automatically	21
class modules	
definition	89
classes	
definition	7, 89
closing documents	65
code	
formatting automatically	20
stepping through	36
Code window	19
coding dialog boxes	41
collections	
counting items in	31
definition	89
parsing items in	31
referencing in macros	30
referencing items in	30
coloring	
shapes	84
syntax automatically	20
combination boxes	44
comments	11
constants	
definition	89
content, changing in documents	61
contextual pop-up lists	21
converting coordinates	50
coordinates	
capturing for macros	50
converting	50
testing	51
Corel Corporation	3



CorelDRAW Graphics Suite X3

about	1
about this manual	2
about VBA in	1
for more information	3
installing VBA	15
CorelDRAW object model	55
documents	55
layers	69
pages	65
shapes	71
counting items in a collection	31
creating	
buttons for macros	46
curves	74
dialog boxes for macros	39
documents	57
ellipses	73
forms	41
GMS files	28
layers	69
macros	27
pages	65
project files	28
projects	28
rectangles	72
shapes	72
text objects	75
toolbars for macros	46, 47
curves	74
D	
debugging macros	35
setting breakpoints	35
stepping through code	36
using the windows	36
declaring	
arrays	10
enumerated types	10
strings	9
variables	9
defining scope	12
definitions, jumping to	21

deleting	
layers	71
pages	69
deploying	
GMS files	52
macros	52
project files	52
workspaces	53
designing dialog boxes	42
determining shape type	79
dialog boxes	
choosing between modal and modeless	40
coding	41
creating for macros	39
designing	42
setting up	40
docking toolbars	23
documentation conventions	2
documents	55
changing content in	61
closing	65
creating	57
exporting files from	61
importing files into	58
opening	58
printing	63
publishing to PDF	64
setting Undo string for	61
switching between	58
viewing	59
duplicating shapes	87
E	
effects, applying to shapes	87
ellipses	73
ending lines	11
enumerated types	10
definition	89
enumerations. See enumerated types.	
event handlers	
definitions	89
providing in macros	33
events	
definition	89



exporting	
files from documents	61
workspace features	53

F

files	
exporting from documents	61
importing into documents	58
importing into layers	70
filling shapes	86
floating toolbars	23
Form Designer	42
formatting code automatically	20
forms	
buttons	44
combination boxes	44
creating	41
images	46
list boxes	44
testing	41
text boxes	44
functions	
building	10
definition	89
G	
global values	
definition	89
GMS files	
creating	28
deploying	52

H

help, providing for macros	51
hiding layers	70

I

icons	
associating images with	48
associating with macros	48
images	
associating with macros	48
providing in forms	46
Immediate window	36

importing	
files into documents	58
files into layers	70
workspace features	53
including comments	11
input boxes	14
installing VBA	15

J

Java and JavaScript	6
jumping to definitions	21

L

layers	69
activating	69
creating	69
deleting	71
importing files into	70
locking and hiding	70
renaming	70
reordering	70
lines, ending	11
list boxes	44
Locals window	37
locking layers	70
logical operators	13



M

macros

adding captions to	47
adding tooltips to	47
associating images with	48
capturing coordinates for	50
capturing mouse actions for	48
creating	27
creating buttons for	46
creating dialog boxes for	39
creating toolbars for	46, 47
debugging	35
definition	89
deploying	52
organizing	52
providing event handlers in	33
providing help for	51
providing user interaction for	48
recording	28
referencing collections in	30
referencing objects in	30
running	34
writing	27

memory allocation	12
-------------------	----

memory pointers	12
-----------------	----

message boxes	14
---------------	----

methods

definition	7, 90
------------	-------

modal dialog boxes

creating	39
definition	90

modeless dialog boxes

creating	39
definition	90

modules	27
---------	----

adding to projects	28
definition	90

mouse actions, capturing for macros	48
-------------------------------------	----

N

non-programmers	5
-----------------	---

O

Object Browser	23
----------------	----

Class list	23
Information window	25
Member list	24
search controls	26

object model

CoreIDRAW	55
definition	7, 90

objects

definition	7, 90
hierarchy	8
parent/child relationship	7
referencing in macros	30

opening documents	58
-------------------	----

operators	13
-----------	----

organizing macros	52
-------------------	----

outlining shapes	84
------------------	----

P

pages

creating	65
deleting	69
reordering	67
resizing	67
switching	66

panning	60
---------	----

parsing items in a collection	31
-------------------------------	----

passing by reference	12
definition	90

passing by value	12
definition	90

PDF, publishing documents to	64
------------------------------	----

pop-up lists, contextual	21
--------------------------	----

positioning shapes	82
--------------------	----

printing documents	63
--------------------	----

programmers	5
-------------	---

programming languages	6
-----------------------	---

Project Explorer	17
------------------	----

project files

creating	28
deploying	52



projects		shapes	71
adding modules to	28	applying effects to	87
creating	28	changing properties	80
renaming	28	coloring	84
properties		creating	72
definition	7, 90	determining type	79
Properties window	18	duplicating	87
providing		filling	86
event handlers in macros	33	outlining	84
help for macros	51	positioning	82
input boxes	14	rotating	83
message boxes	14	scaling	82
user interaction for macros	48	selecting	77
publishing documents to PDF	64	sizing	80
		skewing	83
		stretching	82
R		shortcut objects	32
recording macros	28	definition	8, 90
rectangles	72	sizing shapes	80
referencing		skewing shapes	83
collections in macros	30	starting the VB Editor	17
items in a collection	30	stepping through code	36
objects in macros	30	stretching shapes	82
renaming		strings	9
layers	70	subroutines	
projects	28	building	10
reordering		definition	90
layers	70	subs. See subroutines.	
pages	67	switching	
resizing pages	67	documents	58
rotating shapes	83	pages	66
running macros	34	syntax	
		checking automatically	21
S		coloring automatically	20
scaling shapes	82	T	
scope		testing	
defining	12	coordinates	51
definition	90	forms	41
selecting shapes	77	text boxes	44
setting		text objects	75
breakpoints	35		
Undo string	61		
setting up dialog boxes	40		



toolbars	
creating for macros	46, 47
docking	23
floating	23
VB Editor	22
VBA	16
tooltips, adding to macros	47
 U	
Undo string	61
user interaction (for macros)	48
capturing coordinates	50
mouse actions	48
 V	
variables	
declaring	9
definition	91
VB Editor	17
Code window	19
debugging windows	36
Form Designer	42
Object Browser	23
Project Explorer	17
Properties window	18
starting	17
toolbars	22
VBA	
code structure	8
compared with C and C++	6
compared with Java and JavaScript	6
compared with Windows Script Host	6
definition	4, 91
for non-programmers	5
for programmers	5
installing	15
main elements of	7
operators	13
toolbars	16
viewing documents	59
views	60
Visual Basic for Applications. See VBA.	
 W	
Watches window	37
 windows	59
Windows Script Host	6
workspaces	
deploying	53
exporting features	53
importing features	53
writing macros	27
 Z	
zooming	60



Copyright 2002–2006 Corel Corporation. All rights reserved.

CorelDRAW® Graphics Suite X3 Programming Guide for VBA

Protected by U.S. Patents 5652880; 5347620; 5767860; 6195100; 6385336; 6552725; 6657739; 6731309; 6825859; 6633305; Patents Pending.

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, Corel PHOTO-PAINT, Corel SCRIPT, Corel Support Services, Painter, Paint Shop Pro, and WordPerfect are trademarks or registered trademarks of Corel Corporation and/or its subsidiaries in Canada, the U.S., and/or other countries.

Adobe, Acrobat, PostScript, and Reader are registered trademarks of Adobe Systems Incorporated in the United States and/or other countries. AutoCAD is a registered trademark of Autodesk, Inc. Borland and Delphi are trademarks or registered trademarks of Borland Software Corporation. IntelliCAD is a registered trademark of IntelliCAD Technology Consortium. Java is a trademark of Sun Microsystems, Inc. JavaScript is a registered trademark of Sun Microsystems, Inc. in the U.S. and other countries. Microsoft, ActiveX, Visual Basic, Visual Studio, and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries. Other product, font, and company names and logos may be trademarks or registered trademarks of their respective companies.

023016