

Bonus Chapter 2

Automating Tasks and Visual Basic for Applications

Both seasoned macro programmers and designers who don't know a macro from a macaroon will benefit from this chapter on CorelDRAW's macro and Visual Basic for Applications (VBA) programming features. Novices can use the macro recording feature in CorelDRAW to create simple macros to automate repetitive tasks in CorelDRAW; if you're ambitious, you can also build macros to design very complex artwork—later in this chapter, you'll see an example of a background design generated through a simple macro you can program. If you don't fancy yourself the programming type, you'll also learn how to install and use macros and mini-programs that have been written by third-party programmers.

It is easy to assign VBA macros that you've written yourself or downloaded from the Web to toolbar buttons, menus, and shortcut keys in CorelDRAW. You can insert macros right into CorelDRAW's interface in the places where they're handy. You put macros to work and you can turbo-charge your workflow, making your hours much more efficient and streamlined.

With VBA, any user can tweak or supplement CorelDRAW's features to meet your exact (and perhaps unique) needs. This news is very exciting when you realize that Visual Basic is not limited to working solely within CorelDRAW. VBA can be used to integrate work that takes place, and data that is exchanged, between CorelDRAW and PHOTO-PAINT... *and* most other Corel products, including the WordPerfect Office Suite. Your VBA automation possibilities expand beyond the Corel universe of products to include many other applications such as Microsoft Office and Autodesk's AutoCAD to name but two.

As an application in its own right, learning all the ins-and-outs of VBA can fill its *own* book (and it does), and mastering CorelDRAW's Object Model is not a small topic, either. Everything you would need to know to master VBA control of CorelDRAW cannot be completely documented in this one chapter, but a novice *can* get started using VBA macros by reading the information here. You'll learn how to build your own simple macros, and if you're so inclined, there is plenty of VBA documentation on the Web and in other reading. If you're a programmer, use this chapter as a quick brush-up with pointers to CorelDRAW-specific VBA information and implementations.



Note Check out the Macro Programming Guide.pdf and CorelDRAW Object Model Diagram .pdf files in the Data folder in the CorelDRAW Graphics Suite X6 folder on your hard drive for comprehensive and well-organized details on macro structure.



Download and extract the example files in BonusChapter2.zip

What's Possible with VBA?

You can build simple macros from the ground up that perform everyday tasks, such as creating a rectangle or an ellipse shape. Other operations that aren't a challenge to program are to move and resize objects on the page, change object colors, rotate, extrude, and do almost anything you would normally do in CorelDRAW by click-dragging and using menu commands.

VBA, more precisely, *CorelDRAW's Object Model*, gives developers access to control most of CorelDRAW through programmed code. Using VBA, you can record and write small, ten-line programs that perform specific tasks. You can then assign macros to buttons, menus, or shortcut keys in CorelDRAW for easy access. Alternatively, you can build mini-applications of many hundreds or thousands of lines of code for performing complex tasks that are otherwise difficult or even impossibly time-consuming to execute with a mouse and keyboard alone.

Introducing VBA in CorelDRAW

The big news here for programmers (beginners most likely won't immediately benefit from this new feature) is the addition of Corel Query Language (CQL) to your toolset. Programmers will appreciate the deployment of Corel Query Language, so they can use CQL in their VBA to search for objects, including text that has specific properties, such as shape, fill, outline, color, and other properties.

At the core of using VBA within CorelDRAW is the CorelDRAW *Object Model*. An Object Module shows parent and child relationships of objects (features) that you can control using VBA. You can get a visual overview of the CorelDRAW Object Model by taking a look at the *CorelDRAW VBA Object Model.pdf* file in the Program Files\Coreel\CorelDRAW Graphics Suite X6\Data folder.

Upgrading VBA Macros to X6

Most VBA macros written for earlier versions of CorelDRAW should work in CorelDRAW X6 without any modifications. Copy the GMS files from one folder to the other. However, some macros might fail to work because of minor changes to the Object Model, or because they explicitly reference the CorelDRAW object instead of referencing the CorelDRAW object. If a macro doesn't run, you have to edit it using the Macro Editor (ALT + F11).

In Windows 7, user-created GMS files are stored in Users*(the current user)*\AppData\Roaming\Corel\CorelDRAW Graphics Suite X6\Draw\GMS folder. The GMS files that X6 ships with can be found in C:\Program Files\Corel\CorelDRAW Graphics Suite X6\Draw\GMS. If you're running Windows XP, it's easy to perform a wildcard search for the GMS files: double-click My Computer on your Desktop, navigate to your boot drive (usually C), and then click the Search button toward the top of the folder window. Click All Files And Folders in the "What do you want to search for?" area, and then type *.gms in the All Or Part Of The File Name entry field. Click Search.

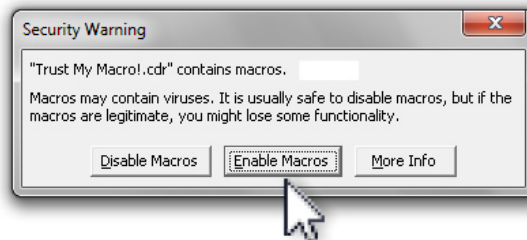
Working with Existing Macros

CorelDRAW ships with some macros; the Internet is a great place to find macros; and you may even be given a document that contains macros that a colleague created. So how do you get them to work?

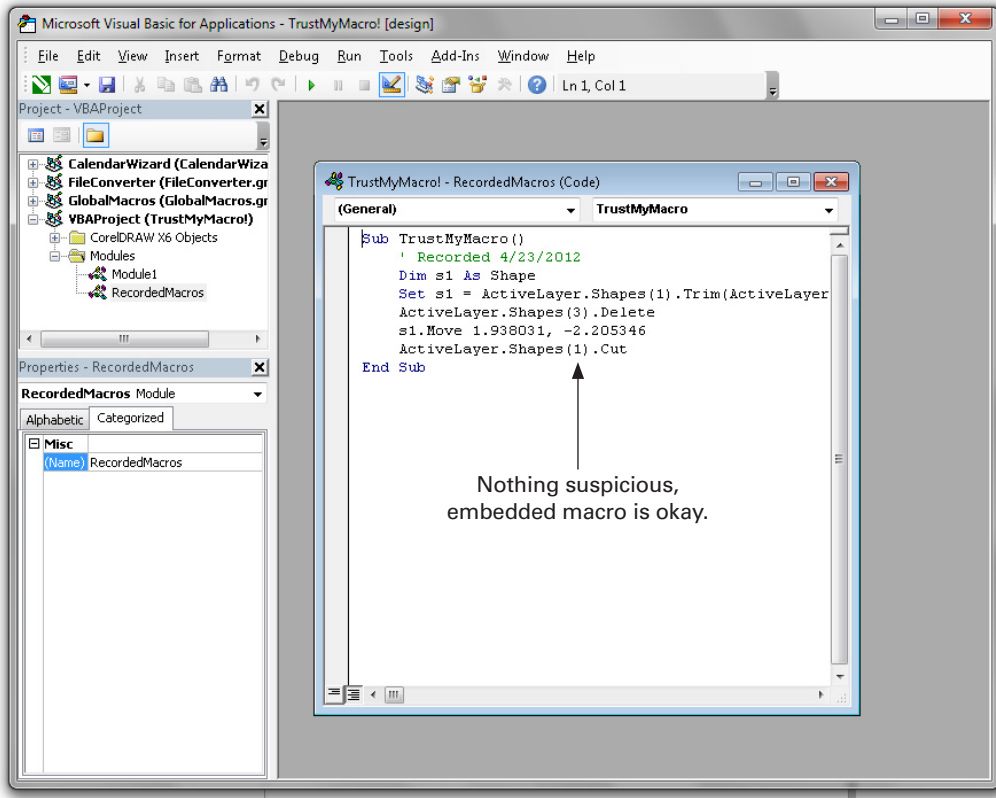
Installing an Existing Macro

Macros are distributed as single project files in the *.GMS file format ("Global Macro"). They may also have accompanying "helper" files. You might download a macro in a zip archive where you extract and copy the included files into your GMS folder, or the developer may have packaged it up with an install program that automatically puts the GMS file and any related files in the correct place for you.

The other way to obtain a macro is to receive it embedded in a CorelDRAW drawing; the CDR file format can hold onto a macro, a very convenient way to share macros. When you open a CDR file that has a macro embedded in it, a Security Warning dialog box appears, asking if you want to Disable Macros or Enable Macros or get More Info, as shown here.



If you've received the document from a trusted source, in other words, from a coworker and *not* as an e-mail attachment with a subject line telling you you've won a lot of money, and you think that you will need to use the macros contained in the document, you *should* choose to Enable Macros. But if you're unsure, choose Disable Macros. Disable Macros does what it implies, but you can use the Macro Editor to view the macros so you can determine if they are safe and you want to use them, as shown here. To use a macro you've chosen to disable, close the CorelDRAW file, reopen the file, and then choose Enable Macros.



Running an Existing Macro

To run a macro, choose Tools | Macros | Run Macros to bring up the CorelDRAW Visual Basic for Applications dialog. From the Macros In drop-down list, choose the VBA project that contains the macro you want to run. Any macros stored in the VBA project file you chose are displayed in the Macro Name field. If the macro was created entirely in the VB Editor and not recorded, this macro is displayed at the top of the list. Recorded macros you've created appear later in the list and the name of the macro appears as RECORDEDMACROSMACRONAMEYOUGAVEYOURMACRO. Select the macro you want to run in the Macro Name list and then click the Run button.

If the macro was created by programming, it might pop up a dialog box or interface for you to set options for what the macro does, or where in the document the macro performs its task. If no interface is provided, the macro probably performs its actions in the active document. Some macros may only work if there is an active selection in the document at the time you run the macro, so if nothing appears to have happened (an extrude, a move, a color change, for example), try selecting an object and then run the macro again.

CorelDRAW comes with very useful CalendarWizard.gms and FileConverter.gms project files, that are VBA macro-based mini-applications. You really owe it to yourself to open a new file and run these macros; the Calendar Wizard alone can shave hours off designing a custom calendar for business or the home. However, make your *own* macro is not difficult; you know best what your own automation needs are. Don't be intimidated by lines of seemingly incomprehensible code: *recording* a macro is the easiest way to create a macro, the topic of the following sections.

Global vs. Local Projects

When you make your own macros you have a choice of making it a *local* or *global* macro. If you want to make the macro you're recording available for use only in the current open document, you store it in the document. However, if you want the macro available whenever you use CorelDRAW—or you want to share the macros with others—you need to create a new VBA project and store the project file in the GSM folder. Storing your macro work in any of the projects that Corel has provided, or those from third-party developers, is not a good idea. Make your own project file for your work, and back it up regularly.



Note

Project files are not created in CorelDRAW or in the VBA Editor. To create a project file, close CorelDRAW, and open a text editor such as Notepad. Open a new file in Notepad and *without typing anything in the file*, save it to C:\\Users*your user name*\\AppData\\Roaming\\Corel\\CorelDRAW Graphics Suite X6\\Draw\\GMS. When saving it, give it a filename (up to 40 characters) that contains only letters, numbers, and underscores, and no spaces. Be sure to give the file a .gms file extension, for example **MyMacroProjects_1.gms**; note there are no spaces in this example filename. Restart CorelDRAW and the MyMacroProjects_1 will appear in the Save Macro dialog as one of the places you can save the macro.

Recording a Macro

You might have recorded a macro in other programs before; if so, you won't find the process very different here in CorelDRAW—it's quite easy. Macros can be recorded using the Macros Toolbar, which contains VCR-like controls for starting, pausing, and stopping the macro recorder. Or you can skip the recorder and make a macro by performing your actions and then saving your macro using the Undo docker. Both methods produce similar results.

Next is a walk-through on recording a macro using the Macros Toolbar.



Tip The macro recorder cannot work with text. It only works with simple shapes. If you need to do something to text you have entered the text and then convert it to curves before you start recording your macro. The recorder also doesn't work well with other complex objects, so you may have to break things apart and ungroup them before they can be part of a recorded macro.

Tutorial

Recording a Macro: Filling a Page with Confetti

1. Open a new document. This is the document where you'll perform the actions that make up the macro. Save the file as **Macro for making confetti.cdr**.
2. With the Ellipse tool, create an ellipse that's about an inch wide. Then fill it with a color on the Color Palette, give it a 4-pixel outline width by using the Property Bar, and then assign the outline a color. The colors you choose will be written to the macro, and when you run the macro, these colors will be used; choose colors that appeal to you! Leave the object selected for the following steps.
3. Right-click on any of the toolbars at the top of the application window. From the pop-up menu choose the Macros Toolbar. The list of macro commands is shown in Figure 1. The Tools | Macros commands can be used to record a macro, but it's easier to use the Macros Toolbar.
4. Click the red Start Recording button on the Macros Toolbar to open the Record Macro dialog.

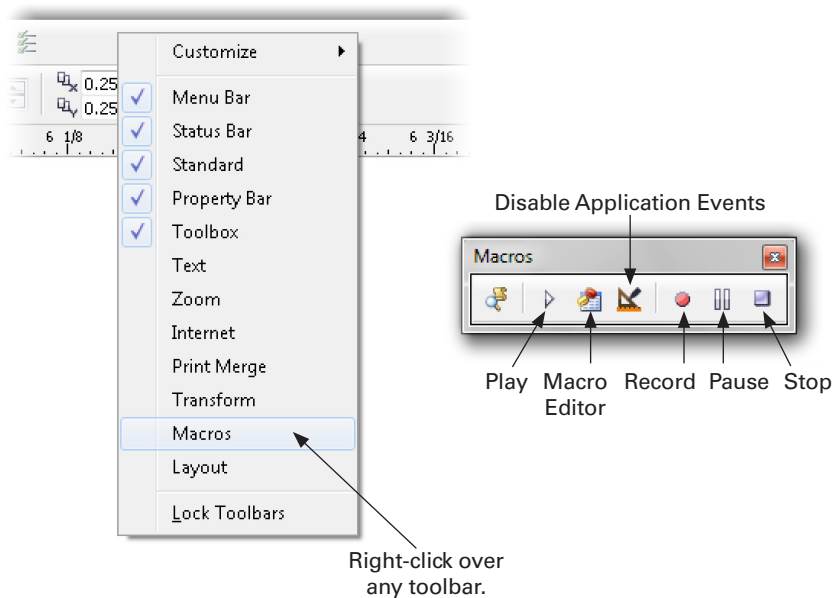


FIGURE 1 Use the Macros Toolbar much in the same way you use any onscreen or even physical recording device. The buttons are labeled with universal symbols.

5. In the Macro Name field, enter a name for your macro. It must be less than 40 characters, start with a letter, and contain only letters, numbers, and underscores. No spaces are allowed in the Visual Basic programming language. Click the icon in the Save Macro In field that represents the project or file in which you want the macro saved. In these steps you'll save locally in the current document; click the VBAProject (Macro for making Confetti.cdr) icon. Write any notes about the purpose of the macro, the date, the version number of the macro, or any other useful information about the macro in the Description section. Click OK and you're recording now.
6. With the Pick Tool, click-drag the ellipse to a different location on the page, and then before releasing the left mouse button, right-click to drop a copy of the ellipse.
7. Scale the duplicate ellipse to about 50 percent, and then recolor it.
8. Marquee-select both ellipses and then repeat Step 6.
9. Repeat Steps 6 and 7; you're populating the page with different-colored, different-sized ellipses. The beauty of this macro is that in the future you don't have to repeat these steps! Additionally, when you marquee-select multiple ellipses try rotating the group as you reposition them by clicking within the group as they're selected to put the bounding box for the group into Rotate/Skew Transformation mode. In Figure 3, you can see the steps used in this example macro.

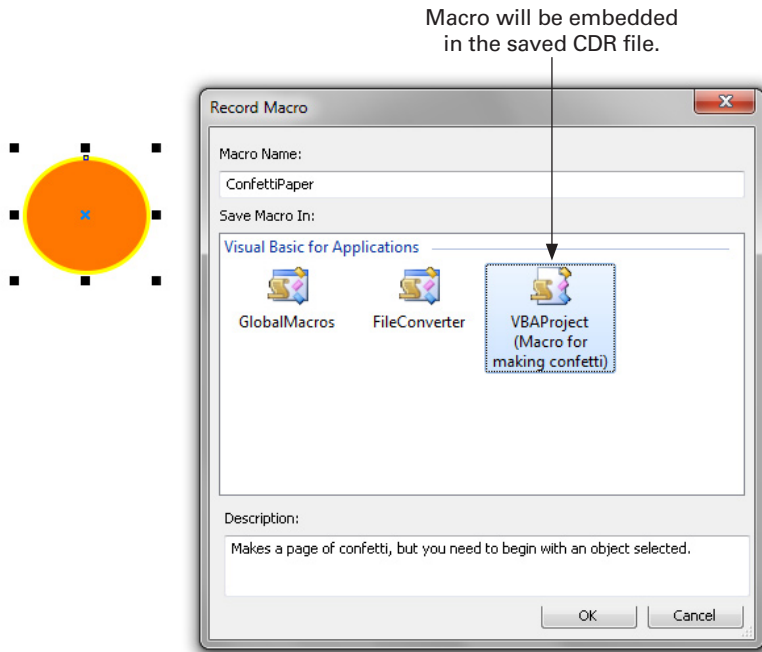


FIGURE 2 Click the icon for the save options for the macro you'll record and write project notes to yourself and others.

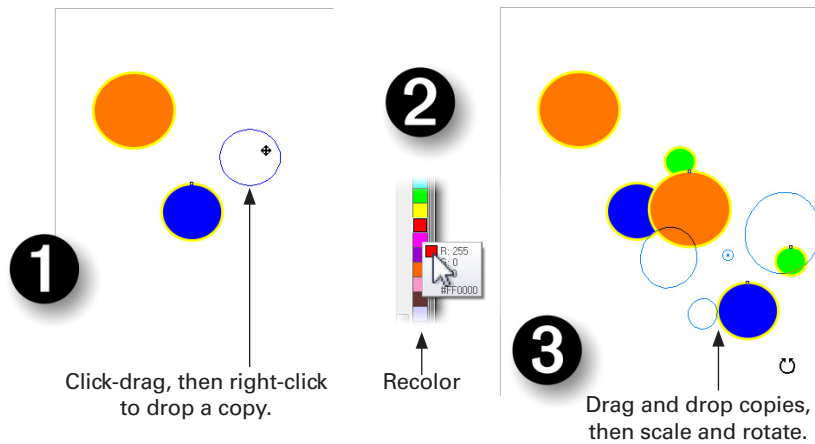
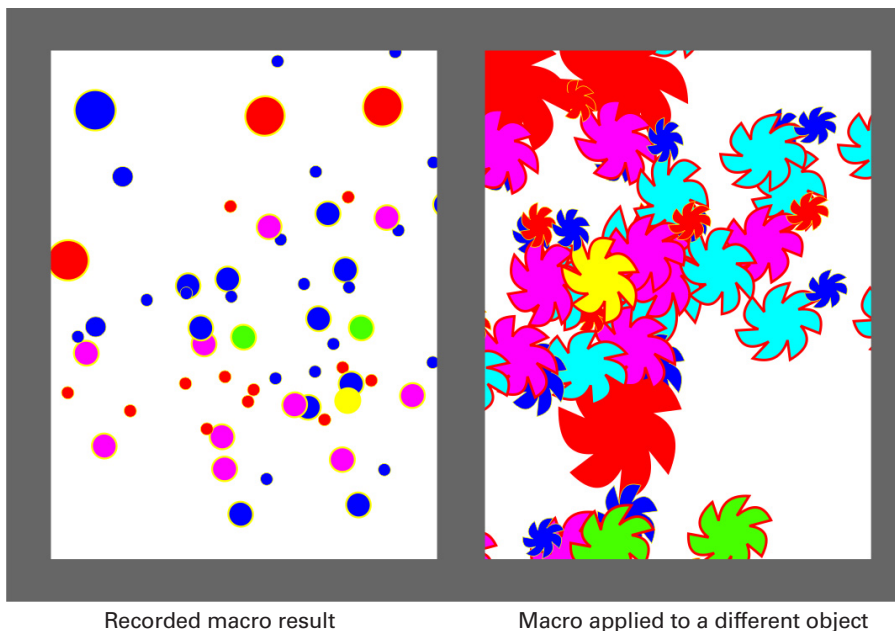


FIGURE 3 Start recording, make some transformations that might be tedious in the future to repeat, and then stop recording. You now have a local macro.

10. Finally, click the Stop Recording button on the Macros bar.

Try now deleting all the objects on the page, and then create a polygon or a rectangle. Select it and then run the macro you just created. Your recipe is the same, but you're using a different ingredient, as shown here. This is a simple macro, but you can see now the potential for incredible time-savings with macros and a little head work before you begin recording. Because the macro was saved in this document, it is only available for use in this document.



Recorded macro result

Macro applied to a different object



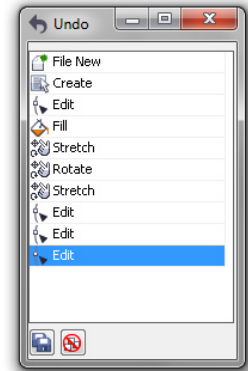
Tip If you later decide that you want to use a macro that is stored in a user document with other files, you can use the Macro Editor to copy the macro into other VBA project documents.

If you'd like to experiment with a document that contains a finished macro, open the Macro for making confetti.cdr document you downloaded at the beginning of this chapter.

Saving Undo Lists as VBA Macros

Lists of Undo Actions can be saved as a VBA macro from the Undo docker as an alternative to recording them. To open the Undo docker, shown at right, choose Window | Docker | Undo, shown here after several actions have been performed on an object.

The list that you see contains the available actions that you can undo or redo using Edit | Undo and Edit | Redo for the active document; Undo and Redo are also registered on the Undo docker when you use these buttons on the Standard Bar. The last-action highlight indicates the last action you performed, which is also the action that is undone if you choose Edit | Undo. If you choose Edit | Repeat, the highlighted item is repeated, if it's repeatable. Clicking any action in the list undoes or redoes any actions between it and the last action highlighted on the docker.



If you click the Save List button, the list of commands starting at—but not including—File New and up to the selected command are saved to a VBA macro. Clicking the Save List button opens the Save Macro dialog. Choose a suitable name, project, and description, and then click OK.

The Clear Undo List button removes *all* Undo and Redo items in the list, an action that in itself *can't* be undone. If you clear the list, you will not be able to undo anything already done in CorelDRAW. However, if you want to save an Undo list as a macro, clearing the list before you perform your actions is a good idea because it removes all the other actions that you don't want included in your macro.

Playing Back Recorded or Saved Macros

To play back a macro that comes with CorelDRAW, a third-party macro, or any valid macro that you have written yourself, click the Play button on the Macros Toolbar, or choose Tools | Macros | Run Macro. The Run Macro dialog appears, where you can choose the macro you want to run.

Choose the correct project in the Macros In list, and then select the macro you want to run in the Macro Name list. Then click the Run Macro button.

Writing a Macro

Creating macros by recording is easy, but unfortunately, there are a lot of things that cannot be accomplished by recording and that only can be done through programming. Therefore, a *taste* of programming language, and programming possibilities, using VBA is covered in the following sections.

Programming Term Definitions

Programming, in a nutshell, is less art and much more science. The English language can state ideas ambiguously. Therefore, it's important to understand a few VBA key words. Some of the following definitions are subtle, but they make the discussion easier as you progress through these sections.



Tip You'll find a more complete explanation of terms in the CorelDRAW Graphics SuiteX6\Data\Macro Programming Guide.pdf.

- **Project** *Projects* group together related *modules* and *forms* into a single file. Project files are stored as separate files with the extension .GMS.
- **Module** A *module* is a VBA document that contains the individual macros. Modules can be *normal* modules, *class* modules (not discussed here), or *forms*.
- **Form** A *form* is a window that contains the user interface for your macro. Forms are also known as dialogs and can contain buttons, boxes, options, text input areas, static text, drop-down lists, and more.
- **Shape** A *shape* is the term used for any object in a CorelDRAW drawing—many people call them objects, but *object* means something different in VBA.
- **Object** *Objects* have a special meaning in VBA: an object in VBA is the general name for any aspect of CorelDRAW that can be named and programmed, such as shape, layer, page, or document, and dozens of other items.
- **Object Model** The *Object Model* is the “wiring” between VBA (or any other programming language) and the CorelDRAW document—without the Object Model, it is simply not possible for you to control the shapes in the document, or even the document's other settings. The Object Model gives everything an object name, so you can get a reference to anything and then modify it.
- **Member** Objects are usually made from smaller objects called *members*. Members can be subobjects, properties, and methods. As an analogy, a car is an object, but it has an engine and four wheels that are subobjects.
- **Property** One aspect of objects is that they have *properties*. A property is a characteristic of an object, such as size, position, or color. You can access these properties with VBA.
- **Method** A *method* is a task that the object can perform when you tell it to: “Resize yourself to three inches wide,” “Move one inch to the right,” “Rotate 10 degrees,” “Group with the other shapes,” “Delete yourself.” Methods are often called *member functions*.

- **Macro, sub, function, and procedure** All these mean broadly the same thing: a block of code that has a clearly defined start point and end point. We use the word “macro” as a general term for *sub*, *function*, *method*, or *program*, although its original meaning is “a collection of keystrokes.” A *sub*, or *subroutine*, is a piece of code that performs some task and then returns control to the object that called it. A *function* is a piece of code that performs a task and then returns a value to the object that called it. You can see that subs and functions are basically the same things, but a function returns a value (a number or text), and a sub does not. (Note that this is different from languages like C++ or Java, in which all procedures are functions and always return a value, even if it is just zero.)

Introducing the VBA Editor

The VBA Editor is written by Microsoft and licensed to Corel for inclusion in CorelDRAW. This means that while the CorelDRAW Object Model is the responsibility of Corel, the VBA Editor is solely Microsoft's. The big advantage here is that the VBA Editor is mature, refined, and first class, as it derives so much from Microsoft's long line of software development tools. Another advantage is the fact that VBA is a variant of Visual Basic, so anyone with VB experience will be able to use VBA without additional learning. And if you are just starting out, what you learn here will give you a leg up when working with VBA or Visual Basic in other programs.

The VBA Editor includes many features designed to assist the programmer, including the following:

- **Syntax Check** This feature checks each line as you type it and immediately identifies any problems it finds by marking the code with red.
- **Auto List Members** A list of all valid members of an object pops up, from which you can choose one, or you can type it yourself.
- **Auto Indenting and Formatting** The Editor tidies up your code and automatically formats the code to maintain a uniform look, which makes it easier to read and debug.
- **Color Coding** The editor colorizes the code according to whether the words are reserved keywords (blue), remarks (green), errors (red), or normal code (black); the colors can be customized.
- **Form Designer** You can quickly design powerful custom forms (dialog boxes) for your macro's user interface containing any of the standard controls, such as buttons, lists, text-entry boxes, and labels.

You'll discover many other useful features of the VBA Editor once you start using it.



Tip Although CorelDRAW uses “VBA,” the editor is borrowed from full Visual Basic, and is known as the “Macro Editor.”

The Macro Editor Layout

The Macro Editor has three main areas: menus and toolbars, dockers, and code and Form Designer windows. These are shown in Figure 4.

The most important window in the Macro Editor is the code window, as this is where you do most of your hands-on programming. The code window is a text-editor window where your code is listed, and where you can enter new code and edit existing code. The next most important window is the Project Explorer, which enables you to navigate among all the modules and components of all your open projects. The Properties window is also important, particularly if you are editing forms.

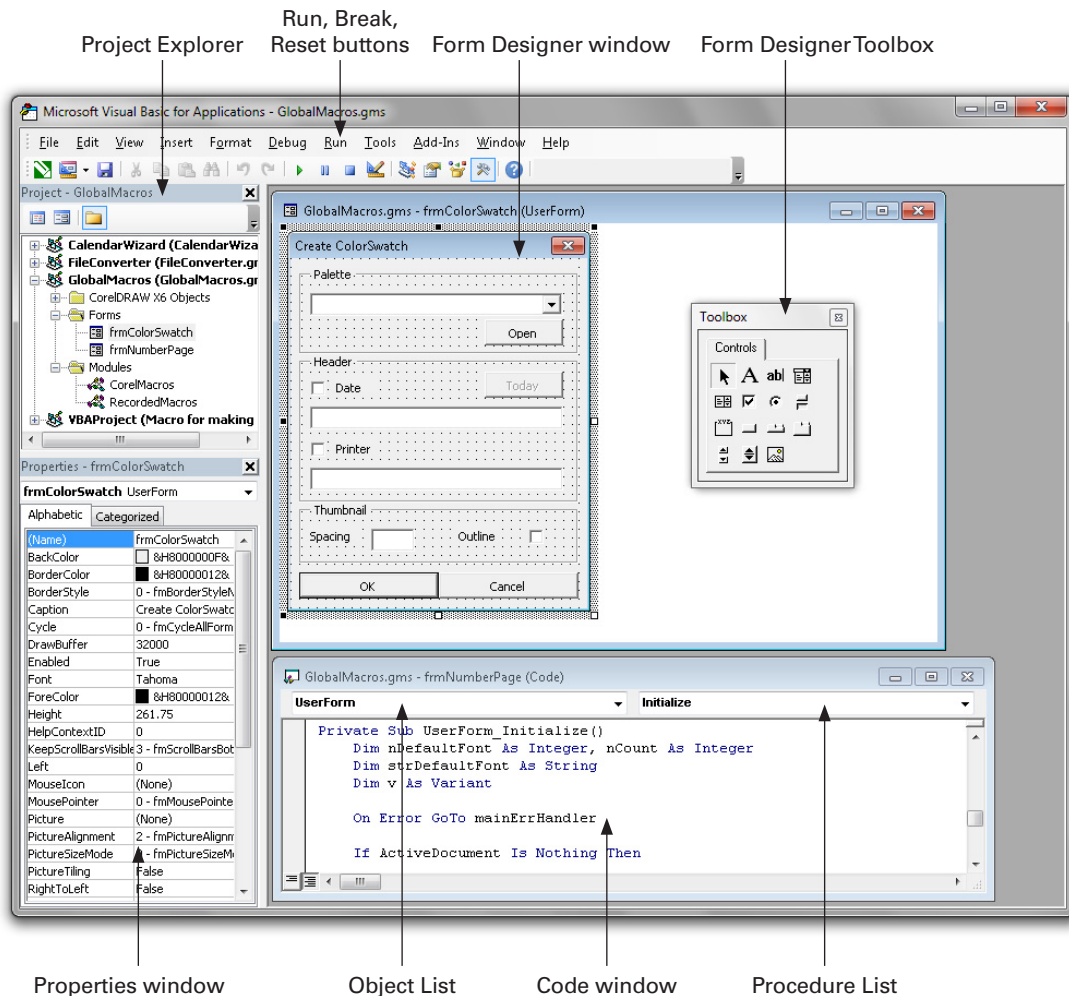


FIGURE 4 The window layout of the Macro Editor

The final powerful feature of the Macro Editor is the Object Browser. This is a fundamental tool you can use when you decide to start programming by hand, rather than just recording macros.

Each of these parts of the Macro Editor is described in the following sections. A more thorough description is presented in Corel's CorelDRAW Macro Programming Guide. Also, any good general VBA book provides additional detail.

The Project Explorer

The Project Explorer, shown in Figure 4, can be switched on by choosing View | Project Explorer or by pressing CTRL + R. It lists all the loaded projects (GMS files) and all the modules that they contain separated into folders. It is simple to use the Project Explorer to keep your VBA code organized.

You can perform various filing operations within the Project Explorer, including creating new modules, importing and exporting modules, and deleting modules. These are explained next:

- **Opening modules and forms** You can open a module or form simply by double-clicking it in the Project Explorer, or by right-clicking and choosing View Code from the pop-up menu. Forms have two parts—the visual controls and the code. Double-clicking displays the controls; right-clicking and choosing View Code displays the code. Or, you can press F7 to open the selected module or form.
- **Creating new modules and forms** Right-click a project you want and choose either Insert | Module or Insert | UserForm. The new module or form is added to the appropriate subfolder. You can name the module or form in the Properties window, and the naming convention is the same as for naming macros (no spaces or special characters).
- **Exporting** Right-click the module or form that you want to export and choose Export File from the pop-up menu. This is a simple way to share small parts of your work with other people.
- **Importing** Right-click anywhere within the project you want to import a module or form into and choose Import File from the pop-up menu.
- **Deleting modules** You can delete a module or form by right-clicking it and choosing Remove from the pop-up menu. This actually removes the module from the project file. So if you want to keep a copy of it, you must export it first.

The Code Window

The Macro Editor code window has several interesting features: the main code entry area, the Object List, and the Procedure List, as shown in Figure 4. The *code window* shows the code from a single module, a class module, or a form, although you can have as many code windows as you like open at the same time. Press CTRL + TAB to cycle through the windows, or choose a window from the window list at the bottom of the Window menu.

The Object List at the top of the code window lists the available objects for that module. If a module is displayed, only one object—the “(General)” object—appears. If a form's code window is open, all the form's controls (buttons, labels, text boxes,

list boxes, and so on) are listed as well as (General). The Procedure List names the available procedures for the selected object. For most modules, this is merely a list of all the subs and functions. (VBA records macros as subroutines, which it identifies with the keyword *sub*.) Basically, these two lists give you a table of contents for zooming around large modules. Some modules can grow to several hundred—or even thousands of—lines in length, so such assistance is welcome.

The Object Browser

The *Object Browser* is a central tool for programming CorelDRAW. The Object Browser shows you how every object, member function, and property within CorelDRAW fits together—and it shows you the exact syntax you must use (the exact words and variables). To start the Object Browser, shown in Figure 5, choose View | Object Browser, or press F2.

In the Object Browser, the Classes list at the lower left shows object types, or classes, that exist within CorelDRAW. Each class may exist as a subobject of CorelDRAW's main *Application* object, or it may be a subobject of a subobject.

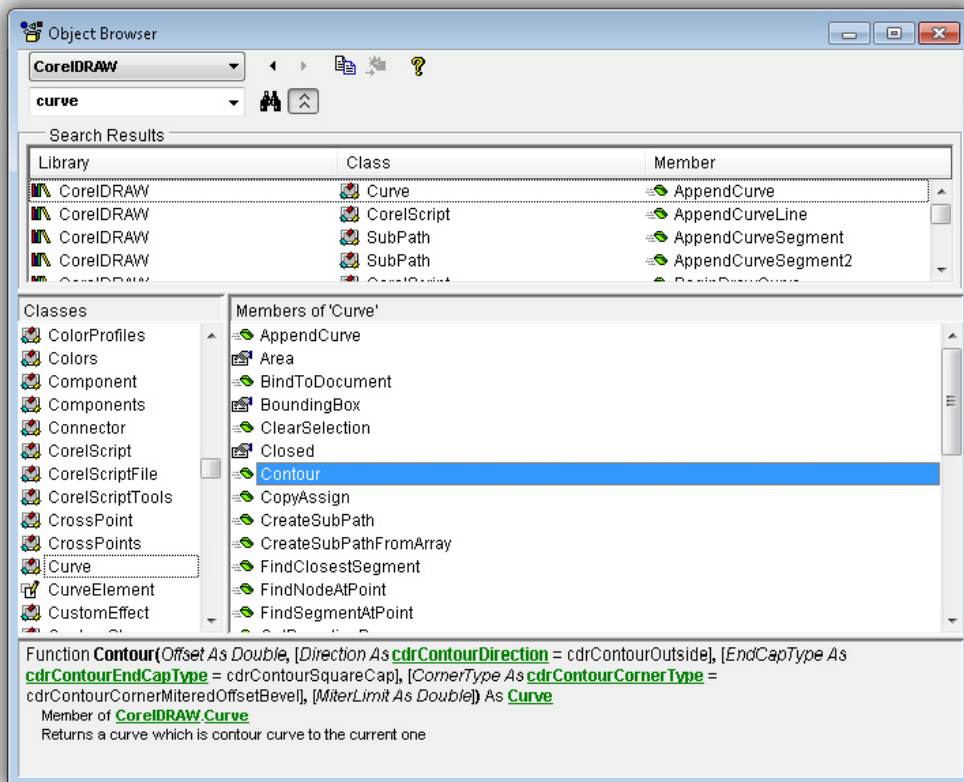


FIGURE 5 The Object Browser provides an insight into CorelDRAW's Object Model.

The list to its right shows the members of the selected class, including all of its properties, methods (subs and functions), and events (not many). The bottom area of the window shows the member's definition, including its parameters. You can click any green words in this area, and you will be taken straight to the definition for that item. The Object Browser also offers a search button (the binoculars icon) to help locate the definition you are looking for. Clicking any item in the Search Results window takes you to that object's definition, as shown in Figure 5.

The Object Browser is a powerful tool, and once you are familiar with VBA, it is quick to use the Object Browser to find a particular definition.

Auto Syntax Check

An *Auto Syntax Check* occurs every time you leave one line of code to move to a different line. The Macro Editor reads the line from that you just moved and checks the *syntax*, or the code you just typed. If any obvious errors are found—perhaps you missed a parenthesis, or the word *Then* from an *If* statement—that line will be highlighted in red.

In its default state, the Macro Editor will use a pop-up message box to tell you about every error. Choose Tools | Options in the Macro Editor, and disable the Auto Syntax Check option: the line will still be highlighted in red, but you won't get any obtrusive messages.

Auto List Members

Each object in VBA has a set of associated members—properties and functions that the object “owns.” With Auto List Members enabled in the Macro Editor's Options dialog (Tools | Options), when you type a dot or period (.) after a valid object name, that object's member properties and functions pop up in a scrolling list. You can choose one of the properties from the list by scrolling down to it and clicking it, or by selecting it and pressing TAB.

If you want to use this feature on a new line, right-click and choose List Properties/Methods, or press CTRL + SPACE.

Form Designer

The *Form Designer* is used for designing objects called forms. Every dialog box consists of a form—a blank panel on which you can place buttons, check boxes, text boxes, title bar, lists, groups, labels, and other controls. Forms are effectively empty dialogs—until you put controls in them.

To design a good, easy-to-use form takes practice. Because this chapter only introduces the basics of VBA, and because forms are generally used only in complex macros and applications, designing forms is not covered here. However, if you need to design a form, keep the following points in mind:

- *Keep it simple.* There's nothing worse than a form that is too intricate or overly clever. Keep your forms simple. Use only necessary controls.
- *Make it usable.* If you make the form illogical and difficult to use, you and your users will be unhappy and unimpressed. A usable, logical layout is important.

- *Use a model of what has been done before.* You can learn a lot from professional applications. Look at what makes a good form and what makes a bad one—learn from the professionals.
- *Pay attention to detail.* When you have finished designing a form's layout, check the details of each control's properties carefully. Make sure every control has an *AcceleratorKey*. At least the most important controls should have a *ControlTipText* each. Make sure that the *TabIndex* order is logical and straightforward.

Recording and Playing Macros

The quickest way for any new programmer to learn how to program CorelDRAW, or any VBA-enabled application, is to record a few actions using the Record Macro and then examine the code. While it is recording, the Record Macro converts your actions into logical VBA code—you might not always get the result you expected, but that in *itself* is a good lesson to learn ... and accept!

Experienced programmers can also benefit from recording macros: CorelDRAW is vast, and finding the exact function or property name to do whatever it is you are trying to do can take some time. Developers often record the action they need to program, and then look at the code that the Macro Record creates, which tells them what they need to know. They delete the recorded macro, but might use some of that code in their own custom macro or program.

In a nutshell, the Record Macro records what it sees you doing. However, it often interprets your actions in roundabout ways. For example, if you create a shape, and then fill it, the recorder does not realize that each action occurs on the same shape; it adds extra, unnecessary code that does the job, but not as efficiently as if you had hand-coded it.

Customizing Macros in the User Interface

If you record or write macros that you use regularly, you might find it handy to assign them to a toolbar button, a menu, or a shortcut key in CorelDRAW. This puts your macros at your fingertips, and this is where you can use macros to optimize your workflow, saving you time and money. Only macros that are stored in macro project files (files with .GMS file extensions) in the GMS folder can be assigned to toolbars, menus, or shortcut keys.

To customize your workspace and assign macros to buttons, menus, or shortcuts, open the Options dialog (CTRL + J), and from the tree on the left, choose Workspace | Customization | Commands to open up the Command section. From the drop-down box, choose Macros. Select the macro you want to add to the interface from the scroll box. Enter a description or the macro name in the Tooltip Help field. Then either assign the selected macro a Shortcut assignment from the Shortcut Keys tab or click + drag the macro from the scroll box and drop it on the toolbar or menu in the location that

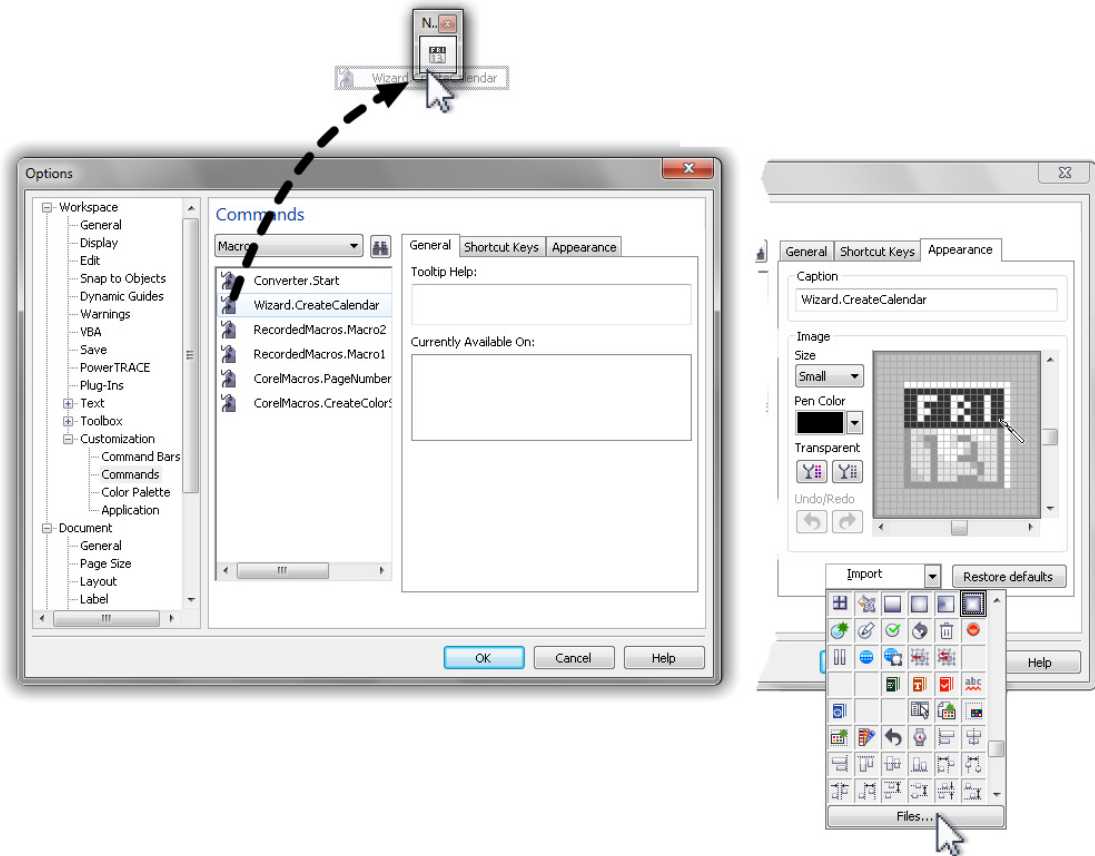


FIGURE 6 Assign a macro to a shortcut key or put it on a toolbar as a button.

you want the macro to appear in the interface. Figure 6 shows a user-recorded macro dragged to the Standard Bar area where it then becomes a single button on a new toolbar. In the Appearance area, you can even draw or choose an imported icon to represent your macro button: the default icon will get confusing after you've put several custom macros on a toolbar. You only have 16×16 pixels for your button's canvas, so plan accordingly and keep the design simple. Figure 6 shows a calendar icon created with PHOTO-PAINT and imported in the BMP file format.

Looking at the Macro Code

If you are at the beginning of the learning curve for programming CorelDRAW, a useful exercise is to record a few macros and see how they work and what the code looks like.

Before you start to program in VBA properly, let's examine a VBA you'll create in the follow section. The term “macro” in this context means “a series of recorded commands that imitate the user's actions.” A macro can be run (*executed*) any number of times, and the result should be the same every time.

Recording Creating New Shapes

You can look at any macro, but it is easier to understand how the code is written if you start with something simple that you record yourself. So record a drawing of a rectangle, about two-inches wide by one-inch high. Here are the steps to record and save this action:

Tutorial Creating a Macro for a Rectangle

1. Open a new document and then first save it with the name **Learning Macros.cdr**. Naming the file first makes it easier to find, run, and edit the macro later.
2. Click the Record button on the Macros Toolbar, or choose Tools | Macros | Start Recording.
3. Give the macro an obvious name, such as **DrawRectangle**. Remember: don't use any spaces or punctuation; the name must start with a letter and can *only* contain the characters A–Z, a–z, 0–9, and _ (underscore). You'll be reminded with an attention box if you don't type a valid macro name.
4. In the Save Macro In section, click the VBAProject (Learning Macros)—the recorded macro will be stored in the current document instead of a global project file.
5. Type in a brief description, such as **Simple test macro; creating a rectangle**.
6. Click OK. Don't be fooled by the lack of activity at this point—VBA records every action you take from now until you stop.
7. Choose the Rectangle Tool (F6), and then draw a two-inch-by-one-inch rectangle in the approximate center of the page; accuracy is not important in this example.
8. Choose Tools | Macro | Stop Recording, or press CTRL + SHIFT + O, or click Stop Recording on the Macros Toolbar.

That's it; you can now play the macro back to create a new rectangle each time at the same position as the original. This is not the endgame, however—the goal is to generate a few macros you can view to gain an understanding of what's written in the macro document.

Let's quickly make another macro; you'll use these simple macros as a basis for your work in the rest of this chapter, when you *write* a macro that performs functions that cannot be captured by the macro recording process.

With the rectangle from the preceding section selected, record the application of a 3-pt-thick blue outline and a Uniform red fill; call this macro **RedFill_BlueOutline**. Stop the recording. Create a different shape, select it, and run this macro: the red fill and blue outline are applied to the new shape.

The lesson you are about to work through is an example of how you can record *approximately* what you require as a macro, and then, through a little text editing, modify the macro later.

Solving Macro Needs with VBA

Now you'll take a look through the code that you recorded a few sections ago. Shortly, you'll optimize the RedFill_BlueOutline macro. Then, you'll create an advanced macro, using the optimized RedFill_BlueOutline as the foundation. This advanced macro will apply the fill and outline to the selected objects. However, it applies the fill *only to those objects that have a fill already*, and it will apply the outline part *only to those objects that already have an outline*. This macro demonstrates how a little VBA programming can be a more powerful resource than the Find And Replace in CorelDRAW when it comes to automating your work.

Viewing Macro Code

To view the VBA code that CorelDRAW created when it recorded your macro, you need to open the project that the macro is stored in in the Macro Editor, also called the Microsoft Visual Basic for Applications Editor. Open the Macro Editor by pressing ALT + F11, or click Macro Editor on the Macros Toolbar. Look for VBAProject (Learning Macros) in the Project Explorer pane on the left side of the Editor window. Each open document that has a macro stored in it, as well as the global macro project files that are stored in the GMS folder, is listed here.

Expand the listing for the VBAProject (Learning Macros). Click the plus button to expand the Modules entry. Under the modules section, you see a listing *RecordedMacros*. Each time you record a macro in this document, it is added to the *RecordedMacros* module. Double-click the RecordedMacros entry in the tree listing and a document window opens to the right, with the name of the VBAProject file, *Learning Macros - RecordedMacros (Code)* on the title bar. This document window contains the actual programming code that you recorded, which makes the macro work.

Taking a Look and Understanding the DrawRectangle Code

In the Learning Macros - Recorded Macros (Code) window you just opened, you should see the following code. Note that the underscore character is used to break the lines to fit within this book's width; this is valid to actually write in code; and using underscores is also legitimate in Visual Basic code.

```
Sub DrawRectangle()  
    ' Recorded 11/28/2007  
    '  
    ' Description:  
    '     Simple test macro; creating a rectangle  
    Dim s1 As Shape  
    Set s1 = ActiveLayer.CreateRectangle(2.240205, 6.824638, 4.25, 5.81974)
```

```

s1.Fill.ApplyNoFill
s1.Outline.SetProperties 0.006945, OutlineStyles(0), CreateCMYKColor(0,
0, 0, 100), ArrowHeads(0), ArrowHeads(0), cdrFalse, cdrFalse,
cdrOutlineButtLineCaps, cdrOutlineMiterLineJoin, 0#, 100, MiterLimit:=45#
End Sub

```

Here's what the lines mean:

- **Sub DrawRectangle()** This is the name you gave your macro. The word “Sub” tells VBA that this is the beginning of a *subroutine*. The parentheses here are empty, but sometimes they have information inside, programmer code, but never in a recording.
- ' Wherever you add an apostrophe, VBA completely ignores the rest of the text on the line after the apostrophe. The green text is *comment information* (notes to yourself or a coworker, for example) that is ignored by VBA.
- **Dim s1 As Shape** *Dim* is short for *dimension*; it reserves enough space in memory for the *variable* called *s1*, which is of type *Shape*. *Variables* are containers for something that is not known until the program is running. For example, if you ask the user his or her name, you won't know the result until the program is run and asks the question; the answer would be stored in a *string* or text variable.

Variables of the type *Shape* are not shapes in themselves; they are a *reference* to a shape. It's more or less like a shortcut in Windows to the CorelDRW.exe file: you can have many shortcuts (references) to the same .exe file, but they are all *forward* links. The CorelDRW.exe file knows nothing of the shortcuts until one of them passes a command along the reference. *Shape* variables are the same: The shape to which *s1* later refers is not bound to *s1*; *s1* is only a *forward reference* to that shape.

- **Set s1 = ActiveLayer.CreateRectangle ...** This actually creates the rectangle. The first four parameters are the left, top, right, and bottom coordinates of the rectangle, in inches from the bottom-left corner of the page. You use *Set* each time the reference stored in *s1* is changed from one shape to a different shape.
- **s1.Fill.ApplyNoFill** This line states that the Fill property of *s1* is set to *no fill*.
- **s1.Outline.SetProperties 0.006945, OutlineStyles(0), CreateCMYKColor(0, 0, 0, 100), ArrowHeads(0), ArrowHeads(0), cdrFalse, cdrFalse, cdrOutlineButtLineCaps, cdrOutlineMiterLineJoin, 0#, 100, MiterLimit: = 45#** This last line is shown as several lines in this book, but is (and should be entered) all on one line in the Macro Editor. This describes the outline properties that have been assigned. In this line of code, you can see that the outline width is *0.006945*, the *Outline Styles(0)* means that it is a solid line, *CreateCMYK Color* is set to black, and other property assignments follow. The settings that are listed here correspond to the settings that are in effect in Outline Pen dialog.
- **End Sub** This closes the sub again and control passes back to the object that called the sub, which could be VBA, CorelDRAW, or another sub or function. It tells VBA to stop here.

Characters that appear in blue are *reserved keywords*, which are special to VBA; you cannot use these words as variable or procedure names. Characters that appear in green are comments or remarks; VBA ignores these completely. Characters that appear in red are lines with syntax errors—VBA cannot understand your fractured VBA “grammar.” All other characters appear in black.

Here are some important things to note:

- Recorded macros are created as enclosed subs, starting with a *Sub* statement and ending with an *End Sub* statement.
- Variables are dimensioned before you use them. This means that you have to name a variable and set aside memory for it (*dimension* it) before you can do anything with it. You are basically listing the “players” that will be participating in the program.
- You use *Set* to set a variable to reference that object. If the variable is a simple variable—if it just holds a number or a string (text) and not a reference to another object—you do not need the *Set* statement.

If you right-click *.CreateRectangle* and select Quick Info from the pop-up menu, the definition for the *.CreateRectangle* method is displayed, and you can see from the parameter names the meanings of each parameter.

If you have a look at the *Layer* class in the Object Browser, you’ll see quite a few members of *Layer* that start with *Create*. These are the basic VBA shape-creation procedures. Therefore, if you wanted an ellipse, you’d use *CreateEllipse*; for a polygon, you would use *CreatePolygon*; or some text would use *CreateArtisticText*.



Note Although you can program the creation of Artistic Text, you cannot record it using CorelDRAW’s Macros Toolbox.

Analyzing RedFill_BlueOutline

For this macro, you recorded two distinct actions, so the code is longer and has more parts to it:

```
Sub RedFill_BlueOutline()
    ' Recorded 4/1/2008
    ' Description:
    '     Simple test macro; creating a rectangle with a fill and outline
    Dim OrigSelection As ShapeRange
    Set OrigSelection = ActiveSelectionRange
    OrigSelection.ApplyUniformFill CreateCMYKColor(0, 100, 100, 0)
    OrigSelection.SetOutlineProperties Color:=CreateCMYKColor(100, 100, 0, 0)
    OrigSelection.SetOutlineProperties 0.041665
End Sub
```

- **Sub RedFill_BlueOutline()** This line declares the beginning of the macro and the name you assigned it. The three lines that follow, as before, are comment lines that hold useful information for you, but are not part of the code that performs any work.

- **Dim OrigSelection As ShapeRange** This first line of the program code declares (dimensions) the term *OrigSelection* to be a *ShapeRange*. A *ShapeRange* is a special VBA term that means a collection of shapes.
- **Set OrigSelection = ActiveSelectionRange** Here the reference *OrigSelection* is set to the object *ActiveSelectionRange*, which is a reference to all the selected objects in the active window.
- **OrigSelection.ApplyUniformFill CreateCMYKColor(0, 100, 100, 0)** This line sets the fill of the selected shape or shapes to a Uniform Fill using CMYK Red.
- **OrigSelection.SetOutlineProperties Color:=CreateCMYKColor(100, 100, 0, 0)** CMYK blue is the color specified here for the selected shape(s) outline.
- **OrigSelection.SetOutlineProperties 0.041665** Here is where the width (in inches) of the outline is specified.
- **End Sub** This is the end marker for this and all macros.

The next refinement to make is that the outline width as recorded in the macro code is in *inches*, regardless of the fact that in the CorelDRAW application window, you set the width in *points*. This is because CorelDRAW's default document units in VBA are inches, and you want to set the VBA document units to points, and then set the outline width to three points.

To make this change you'll edit the macro in the Macro Editor, not by making changes to settings in CorelDRAW and rerecording.

Place your cursor in front of the last program line (*OrigSelection.SetOutlineProperties 0.041665*) and press ENTER to create a space where you can insert an additional line of code. Insert your cursor in the blank line you made and type in this new line of code:

```
ActiveDocument.Unit = cdrPoint
```

Then in the next line change *.041665* to 3 so that the line now reads

```
OrigSelection.SetOutlineProperties 3
```

The macro should look like this now:

```
Sub RedFill_BlueOutline()
    ' Recorded 4/1/2008
    '
    ' Description:
    '     Simple test macro; creating a rectangle with a fill and outline
    Dim OrigSelection As ShapeRange
    Set OrigSelection = ActiveSelectionRange
    OrigSelection.SetOutlineProperties Color:=CreateCMYKColor(100, 100, 0, 0)
    OrigSelection.ApplyUniformFill CreateCMYKColor(0, 100, 100, 0)
    ActiveDocument.Unit = cdrPoint
    OrigSelection.SetOutlineProperties 3
End Sub
```

Click the Save button in the Macro Editor application before testing or running the changed macros.

CorelDRAW supports a lot of different units, including millimeters, centimeters, meters, feet, points, pixels (whose size is determined by the property *Document Resolution*), and picas that you can set in your macros using this technique.

Extending RedFill_BlueOutline

Let's now take the basic *RedFill_BlueOutline* macro and add some extra code. The fill is applied to the selected shapes that already have fills, and the outline is applied to the selected shapes that already have outlines, but neither is applied to those shapes that neither have one nor the other. To do this, you'll add two fundamental programming methods: the *loop* and the *decision*.

First, however, you should know about a powerful feature in VBA: the *collection*.

Collections

VBA provides the programmer with a strong method of handling many similar objects as one object—a *collection*. Say that you have selected ten shapes in CorelDRAW, and you run a script that starts like this:

```
Dim shs as Shapes
Set shs = ActiveSelection.Shapes
```

The variable *shs* is dimensioned as type *Shapes*. The *Shapes* type is a type of collection of many *Shape* objects. Think of it as a container that can hold many references to similar objects. The type *Shape* (singular) means *anything drawn in CorelDRAW*, so the *collection* of *Shapes* contains lots of references to *Shape*—to items drawn in CorelDRAW. Collections are a particular type of *array*, if you have done any programming before you're already familiar with arrays.

Here's the payoff: once you've set a reference to a collection of shapes, you can reference each shape in the collection individually using a loop, which is what you'll do next. The other advantage to referencing is that you do *not* need to know the size of the collection at any time; VBA does all of that for you. If the user doesn't select anything, the result of the operation is an empty collection. On the other hand, if the user selects 1000 objects, you get one collection of 1,000 shapes. The chore of always having to know exactly how many shapes are selected has been taken over by VBA, so you can get on with some clever coding.

Looping

A *loop* is a piece of code that is run, run again, and rerun until a condition is met or until a counter runs out. The most useful loop to us is the *For-Next* loop, of which there are two types: basic *For-Next* and *For-Each-Next*.

The basic *For-Next* loop might look something like this:

```
Dim lCount as long
For lCount = 1 To 10
    MsgBox "Number" & lCount
Next lCount
```

This loop counts from one to ten, displaying a message box for each number.

The *For-Each-Next* loop comes into its own *when dealing with collections*, however. The purpose of a collection is to allow the programmer to reference the collection without knowing what is inside. Thus, to step through all the shapes in the collection, the following code is used:

```
Dim shs as Shapes, sh as Shape
Set shs = ActiveSelection.Shapes
For Each sh in shs
    ' We will add our own code into this loop
Next sh
```



Tip This loop code is a fundamental algorithm when programming CorelDRAW. It is strongly recommended that you become very familiar with it because you will need to use it often.

Note the *Dim* line: You can dimension more than one variable on a single line by separating each variable with a comma. With large modules that have tens of variables, this helps to keep down the module length.

This code loops through all the shapes in the collection, and you can replace the remark line with your *own* code—of as many lines as needed. Each time the *For* line is executed, *sh* is set to the next shape in the collection. A programmer can then access that shape's members within the loop by referencing *sh*. For example, the following code sets the width of each shape to two centimeters:

```
Dim shs as Shapes, sh as Shape
Set shs = ActiveSelection.Shapes
ActiveDocument.Unit = cdrCentimeter
For Each sh in shs
    sh.SizeWidth = 2
Next sh
```

Because this code operates on each shape, one shape at a time, the size is set relative only to that shape and not to the selection, so each shape is now two centimeters wide; but the selection's width is still approximately the same.

Now it's time to move on to making decisions and then put loops and decisions together.

Decision Making—Conditionals

Decision making is what really sets programming apart from macros. Macros in their original sense are little pieces of “dumb code.” Macros do not make decisions; they just perform a series of actions without any understanding of what they are doing. However, as soon as you introduce decision making, a macro becomes a *program*.

Decisions in VBA are made using the *If-Then-Else* construction: *If* (something is true) *Then* (do this), or *Else* (do that). The conditional statement—the “something”—must be able to return the answer true or false, but VBA is very tolerant. For example, you could write code in this way:

```
If MsgBox ("Do you want to hear a beep?", vbYesNo) = vbYes Then Beep
```

As long as the Yes button is clicked, you will hear a beep. In this case, the result of the test in the conditional statement was True or False: The button clicked either *was* the Yes button (True), or else it *was not* the Yes button (False).

The statement does not have to reside all on one line and, usually, you would not write it so. Instead, you might write it something like this:

```
If sShape.Type = cdrEllipseShape Then
    MsgBox "Ellipse"
Else
    MsgBox "Some other shape"
End If
```

Like most things in VBA, what is opened must be closed; don't forget to add an *End If* statement. You don't always have to supply an *Else* if it's not necessary, but it is a good programming technique.

Notice also that you can use Boolean operators—such as And, Or, Not, and Xor—to combine results from two or more conditional statements.

Conditional Loops—Putting It All Together

Now you know how to assign an outline and a fill (RedFill_BlueOutline), and you've looked at looping through collections (*For-Each-Next*). You should also have a fairly good idea about decision making (*If-Then-Else*). The trick is to put all of this together, so you can apply the fill and the outline based on whether each object already has a fill or an outline.

Most of the code already exists for you in the previous code. The only missing part is a reliable decision-making routine for this particular instance. What you and your soon-to-be program need to determine for each shape are the following:

- *Does it have a fill?* If it does, *then* apply the new fill; *else* do nothing to the fill.
- *Does it have an outline?* If it does, *then* apply the new outline; *else* do nothing to the outline.

Fortunately, determining an object's outline is simple: The *Shape.Outline.Type* property returns either *cdrOutline* or *cdrNoOutline*. All that needs to be done is to ask whether the outline type is *cdrOutline*.

Determining whether a shape has a fill is slightly trickier: ten different fill types are possible, including Uniform, Fountain, PostScript, Pattern, and so on. Instead of asking, "Does the shape have a fill?" and having to ask it for all the different types, it is easier to ask, "Does the shape *not* have a fill?" and invert the answer, as in "Is the shape's fill *not* of type *cdrNoFill*?" For this, you use the greater-than and less-than symbols together (< >), which means, not-equal-to.

Put all this together and here is the resulting code:

```
Sub Apply_RedFill_BlueOutline()
    Dim dDoc as Document
    Dim sShapes As Shapes, sShape As Shape
    Set dDoc = ActiveDocument
```

```

dDoc.BeginCommandGroup "Apply Red Fill & Blue Outline"
dDoc.Unit = cdrPoint
Set sShapes = ActiveSelection.Shapes
For Each sShape In sShapes
    If sShape.Fill.Type <> cdrNoFill Then
        sShape.Fill.UniformColor.RGBAssign 255, 0, 0
    End If
    If sShape.Outline.Type = cdrOutline Then
        sShape.Outline.Color.RGBAssign 0, 0, 255
        sShape.Outline.Width = 3
    End If
Next sShape
dDoc.EndCommandGroup
End Sub

```

This code steps through the collection of selected shapes, one at a time. It first asks “Does the shape have a fill type that is not *cdrNoFill*?” and applies the new fill if the condition is true. Then it asks “Does the shape have an outline?” and applies the new outline if it does.

In the code above, notice that before *End Sub* there's *EndCommandGroup*, which deserves an explanation here. The *BeginCommandGroup* and *EndCommandGroup* pair of methods group all the commands in between together into a single Undo statement in CorelDRAW, with the name that you specify. After running this macro once, an item will appear on CorelDRAW's Edit menu called Undo Apply Red Fill & Blue Outline, which, when selected, removes all of the commands between the *BeginCommandGroup* and *EndCommandGroup* statements.

Developing from Scratch

Developing solutions from scratch requires experience and planning—experience with using CorelDRAW's Object Model, and a plan of what the solution is supposed to achieve. Most of all, it requires *planning*. Given sufficient clarity and completeness in the plan, usually the code for the solution is a simple step.

For very large projects, it is better to break the solution down into small, self-contained chunks that you can develop independently of the rest and test in isolation—testing that the code does do what you expect is important and is far simpler with small chunks of code than with whole projects in a single bite.

Where to Go for More Information

Using VBA to perform automation in CorelDRAW is a lengthy topic that extends well beyond this chapter. You now have a good understanding of how to record your own macros and how to play them back. That alone should be quite useful in future simple automation needs. However, it's important understand that there is a lot more you can do than just record a few keystrokes and mouse clicks and call it a day with VBA.

To attain a proficiency with VBA in CorelDRAW, two areas of expertise and knowledge lead to productivity—CorelDRAW's Object Model and VBA. You'll find a lot more information about all of the Macro Editor and command syntax in the VBA Help files. Also, CorelDRAW's Help files contain information on Visual Basic for Applications, and the Object Browser is also a great place to go for more information on the Object Model. There are plenty of other places to get help as well.

Newsgroups and Forums

Corel offers a large number of newsgroups and support forums for their products. You can find details on how to access Corel's newsgroups by going to Corel's website at www.corel.com. Navigate to the Community area and then click Newsgroups in the menu bar for details about Corel's newsgroups. If you prefer to use web-based forums, go to www.facebook.com/corel and from the page's menu bar, click Forums. There's also a community app for iOS at the App Store online.

You can seek help freely via Facebook and forums, and because lots of people use only Facebook *or* the forum, it is a good idea to *check out both* when you have a question. The forums and newsgroups are filled with knowledgeable, friendly professionals who are usually eager to jump in and lend you a hand.

Corel Web Sites

Corel has several websites that offer information for users. Conduct a search on the coreldraw.com site. You'll find free macros and other materials in the Downloads section that Corel engineers and users have posted.

If you are looking for information on what third-party developers have whipped up to enhance CorelDRAW, check out the Resources | More | Third-party Tools under the CorelDRAW product section at www.corel.com.

Visual Basic Web Sites

There are many excellent websites with information on how to program with Visual Basic, and some on Visual Basic for Applications. Because VBA is a pared-down version of Visual Basic, you can use the Visual Basic websites as an excellent learning resource.

To find other informational websites, use your favorite Internet search engine to locate some VB sites, and browse them—and any other sites they link to.

