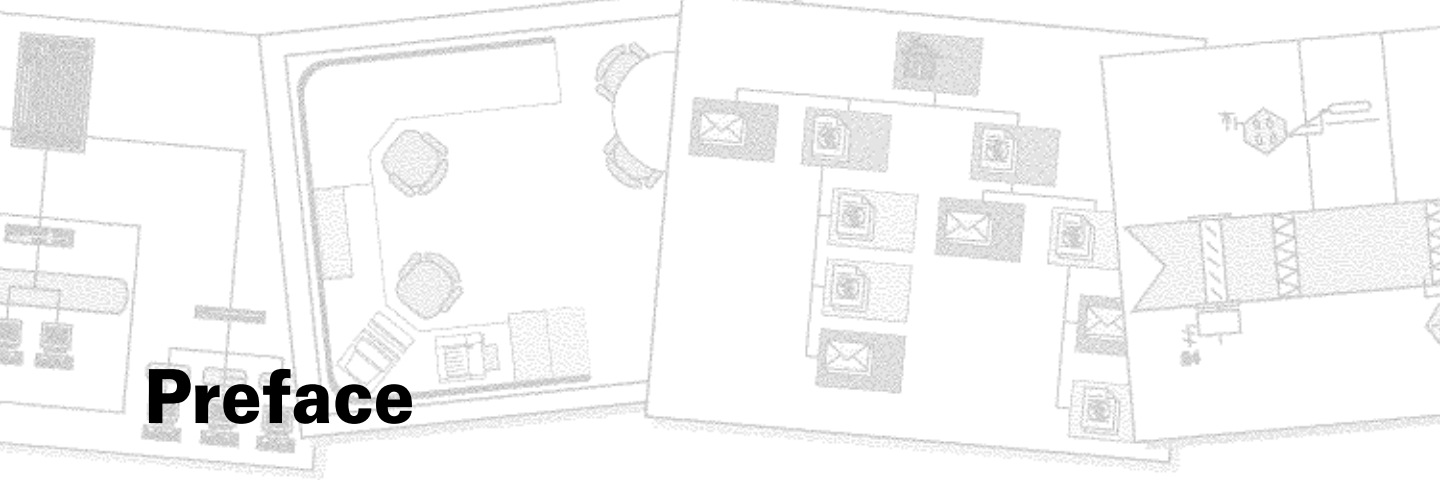


Preface

Developing Visio Solutions is a complete guide to creating graphic solutions with the Visio® Professional, Visio® Technical, and Visio® Standard drawing applications. It includes:

- A conceptual introduction to the Visio development environment and the tools you can use to create real-world solutions.
- Details about developing your own shapes, stencils, and templates.
- Code samples, tips, and techniques for using Automation (formerly OLE Automation) to extend Visio or use Visio as a component in your own applications.

This Preface defines the book's intended audience, purpose, and assumptions.

Topics

Who this book is for

This book is for architects, engineers, users of CAD programs, application developers, system analysts, programmers, and anyone who wants to customize Visio shapes or solutions. We assume you are already familiar with drawing techniques and with the Visio menus, tools, and commands.

We also assume a high-school–level knowledge of basic geometry and Cartesian coordinate systems. An understanding of transformations, trigonometry, and analytic geometry can also be helpful.

In the chapters that explain how to use Automation to control Visio, we assume you are familiar with the programming language you'll be using. Most of the examples in this book are written in Microsoft Visual Basic for Applications (VBA).

What this book is about

This book is about developing solutions—combinations of shapes and programs that model the real world and solve specific drawing problems.

Using Visio shapes to create solutions

A Visio solution may not always involve a program or even a template, but it will almost always involve shapes. The first thing to know about Visio shapes is this: There's more to a shape than what you see on the drawing page—there's Visio SmartShapes® technology.

Every Visio shape includes an assortment of formulas that represent its attributes, such as its width and height, and its behavior, such as what the shape does when a user double-clicks it. You can create your own formulas that model a shape's appearance and behavior after the real-world object you want it to represent. So, for example, you can associate important data—part numbers, names, manufacturers—with shapes representing office equipment. Your shapes can then become powerful components whose unique behavior within a larger solution is provided by the formulas you write.

You can view and edit a shape's formulas in the Visio ShapeSheet™ spreadsheet. For details about working in the ShapeSheet window, see Chapter 2, "Tools for creating solutions." For details about techniques you can use to control shapes with formulas, see Chapters 3 through 9 in Part II, "Developing Visio Shapes."

Using SmartShapes technology

A solution rarely consists of a single shape. More often you'll develop a suite of shapes that support a particular kind of drawing, and you'll assemble these shapes as *masters* in a Visio *stencil*.

A master is a shape in a stencil that you use to create *instances*, or *shapes*, based on the master. Instances inherit many of their characteristics from the master.

When a user drags a master onto a drawing page, Visio automatically creates a copy of that master in the drawing's *local stencil*. The local stencil is stored in the drawing file itself. This has two major benefits:

- First, the drawing is entirely self-contained and portable. Once the user creates the drawing, he or she no longer needs your stencil.
- Second, characteristics of instances can be changed in the drawing by editing the master in the local stencil.

To help the user create the drawing, you'll often provide a *template*. A template can set up the drawing page with a uniform grid and scale, include specific styles and layers, and provide shapes already on the drawing page. A template can also open one or more stencils. When the user creates a drawing based on a template, Visio opens the stencils and creates a new drawing file, copying the template's styles and other properties to the new file. As with the stencil, once the user creates the drawing, he or she no longer needs the template.

For details about creating and testing stencils and templates, see Chapter 9, "Packaging stencils and templates."

Writing programs to control Visio

Some solutions require more than shapes, stencils, and templates. For example, you might need to create drawings based on data that changes from day to day, or perform routine shape development tasks over and over. You may support users who need to create drawings but don't want to become Visio experts, or you may use their drawings as a source of data for other purposes.

You can automate such tasks by using Automation to incorporate the functionality of Visio, simply by using its objects. If you're familiar with VBA, you use objects all the time—controls such as command buttons, user forms, databases, and fields. With Automation, you can use other applications' objects as well. Drawings, masters, shapes, and even the Visio menus and tools can become components of your programs. A program can run within an instance of Visio or start Visio and then access the objects it needs.

Visio Standard, Visio Technical, and Visio Professional include VBA, so you don't need to use a separate development environment to write your programs. However, you can write programs that control Visio in any language that supports Automation as a controller. Most of the examples in this book are in VBA, but the principles apply to any programming language.

For suggestions on how to plan an Automation solution that uses Visio, see the chapters in Part III, "Extending Visio with Automation" beginning with Chapter 10, "Automation and Visio." For an introduction to the Visio object model, see Chapter 11, "Using Visio objects." Chapters 12 through 18 explain various programming techniques you can use to create and work with drawings and shapes, customize the Visio user interface, and run your programs. For details about using other programming languages, see Chapter 19, "Programming Visio with Visual Basic," and Chapter 20, "Programming Visio with C++."

New features for developers

The 5.0 release of Visio Standard, Visio Technical, and Visio Professional gives you a single platform for your custom solutions. Take advantage of these new features and tools:

Visio as an ActiveX container. Add ActiveX controls directly to Visio 5.0 drawings to make your Visio solution interactive. Or add custom controls that you develop or purchase to incorporate more complex functionality.

Connection events. Use the new **ConnectionsAdded** and **ConnectionsDeleted** events to monitor the addition and removal of connections between shapes.

Paths and Curves objects. Use these new objects to report the points along a shape's strokes.

Custom patterns. Create your own fill patterns, line patterns, and line ends that your users can apply to shapes as they do the built-in Visio styles.

Dynamic connectors. Design your solutions to work with the built-in dynamic connector and automatic layout technology or create your own routable connectors.

Named connection points. Assign names to the Connections.Row cell for clearer cell references, and use the new data cells in a connection points row to store associated formulas.

Developer toolbar. Use the Developer toolbar for quick access to the ShapeSheet window and VBA.

Improved ShapeSheet help. Press F1 in a ShapeSheet cell to display an online help topic for that cell.

Open Document Management API (ODMA) support. Take advantage of support for ODMA with third-party tools that manage document workflow, so you can track revisions and provide check-in and check-out services.

Hyperlinking. Use the improved ShapeSheet support to create hyperlinks from Visio shapes and drawing pages.

Sample shapes and code

The Visio CD includes sample shapes and source code for the examples shown in this book, so you can study and work with them as you read. These files are installed in the \VISIO\DVS folder (or the folder that contains Visio). For details about the \DVS folder contents, see the file \DVS\README.TXT.

Summary of \DVS folder contents

Folder	Contents
\DVS	OBJECT TABLE.DOC, lists of methods and properties by object and by name, and OBJECT MODEL.VSD, the Visio object model.
\DVS\LIBRARIES	Folders of C++ source code described in Chapter 20 and Visual Basic utility programs.
\DVS\SAMPLE APPLICATIONS	Files for the Stencil Report Wizard, the sample application described in Chapter 12.
\DVS\SHAPE SOLUTIONS	Stencil of shapes described in Chapters 3–9.
\DVS\VB SOLUTIONS	Visual Basic source code described in Chapter 19 and other sample programs.
\DVS\VBA SOLUTIONS	Template containing VBA macros described in Chapters 10–18 and other useful macros.

Online reference material

In addition to the information provided in *Developing Visio Solutions,* you have online access to the following detailed reference information:

Shape developer's reference. A complete list of ShapeSheet cells and Visio functions. To display it, from the Help menu, choose Visio Help, click Contents, and then choose Reference.

Visio Automation Reference. The PROGREF.HLP file that describes the Visio objects, properties, and methods that you can access from an external program. To display it, from the Help menu, choose Visio Help, and then choose Automation Reference.

"DevWeb". The Visio Solutions Development Web page, which you can access from the Visio home page at http://www.visio.com/devweb/.

Installing help and sample files

If while using Visio you receive a message that PROGREF.HLP cannot be found, or if you cannot locate a sample file referred to in this book, you can install it from the Visio CD. PROGREF.HLP, or the Automation Reference, is installed with the Help files when you install Visio Standard, Visio Professional, or Visio Technical. The sample files are installed with the component called Developing Visio Solutions. These options are included on the Setup program's Select Components screen.

Conventions

This book uses the following typographical conventions.

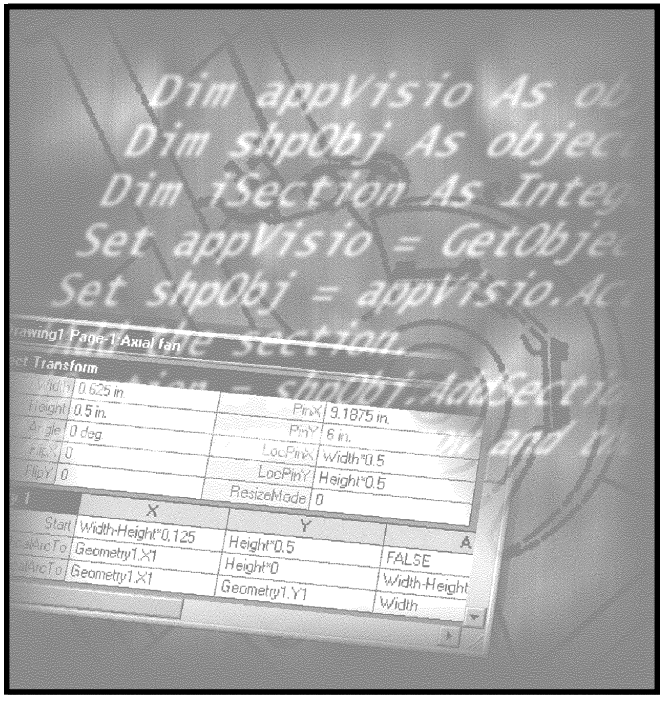
Typographical conventions

Convention	Description
bold	Programming language terms in text.
italic	Variables in text or terms defined in text. In syntax, italic letters indicate placeholders for information you supply.
EmbeddedCaps	Capitalization for readability in Visio and VBA. Language terms are not case-sensitive in Visio or VBA, but they are case-sensitive in C++.
SMALL CAPS	File names in text.
`monospace font`	Code examples.

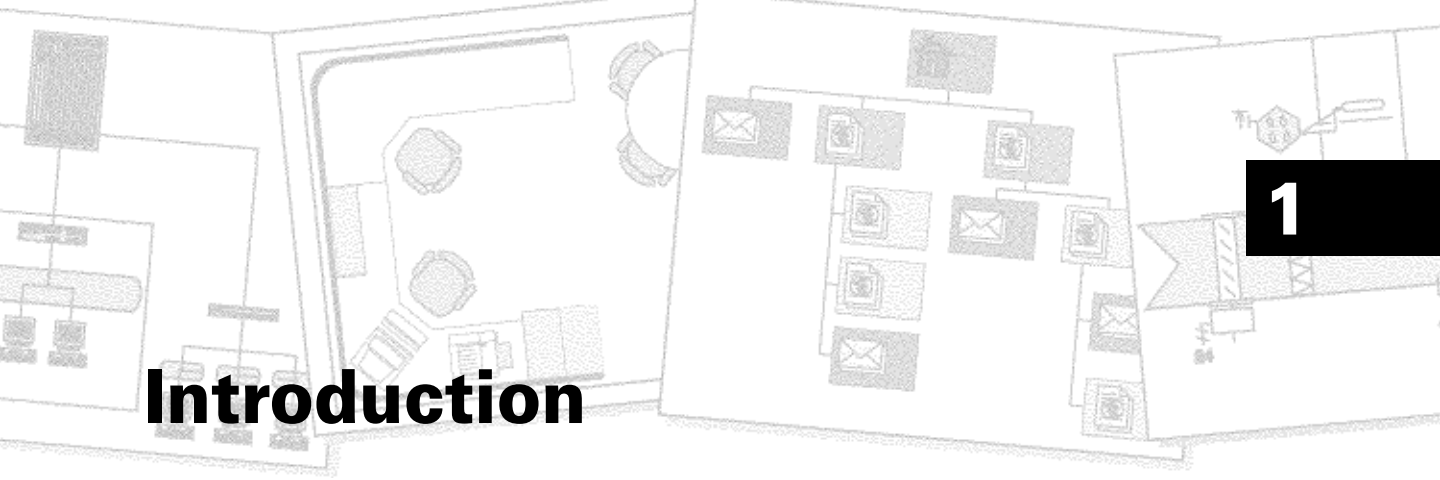
In addition, to enhance readability of formula and code samples in this book, these conventions are followed:

- Within Visio formulas, we have inserted spaces before and after operators and equal signs (=). These spaces are not required and are removed by Visio if you enter them with your formula.
- In code examples, we have used numeric and string constants where you would ordinarily use variables or global constants, especially if you intend to localize your programs.

PART I



The Visio Development Environment



1

Introduction

A software solution typically combines a custom program with one or more shrink-wrapped applications. Rather than developing functionality from scratch, the solution developer uses functionality that is built into a shrink-wrapped product and exposed by a mechanism such as Automation (formerly OLE Automation). Visio offers the solution developer programmable shapes and easy access to sophisticated graphics functionality.

This chapter introduces the features that Visio offers and some concepts that will help you decide how to use them. For details about the tools Visio provides for implementing a solution, see Chapter 2, "Tools for creating solutions."

NOTE To get the most out of this chapter and the rest of this book, you should be familiar with Visio menu commands and tools and with the programming language you intend to use to develop your solution. The best way to get acquainted with Visio is to create a drawing or two. If you haven't yet done this, we recommend that you do it now before continuing with this book. Also, locate the Visio online help so you can find out more about the basics if you have questions about them while reading this book.

Topics in this chapter

Modeling with Visio

A *model* helps you analyze and solve a problem using objects that resemble things in the domain of the model, whether that's the organization of people in your department, the arrangement of desks and chairs in a floor plan, the network you're selling to a customer, or a state diagram for an integrated circuit.

Obviously, Visio shapes can represent the objects in a model graphically. However, they can do much more:

- Visio shapes are programmable through ShapeSheet formulas, so you can make them behave like the objects they represent in the real world.
- Visio shapes, drawings, and even Visio itself can be controlled by external programs through Automation, so you can automatically generate drawings, extract data from them, and check their correctness.
- Visio shapes can have meaningful data associated with them, so they can more closely represent real-world objects.

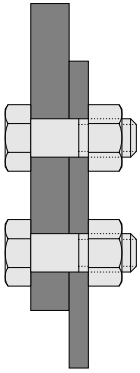
Visio is an excellent tool for solutions that involve modeling, because you can design Visio shapes to behave like objects in your problem domain. A Visio drawing and the shapes it contains are reusable tools that help you analyze, communicate, and make decisions.

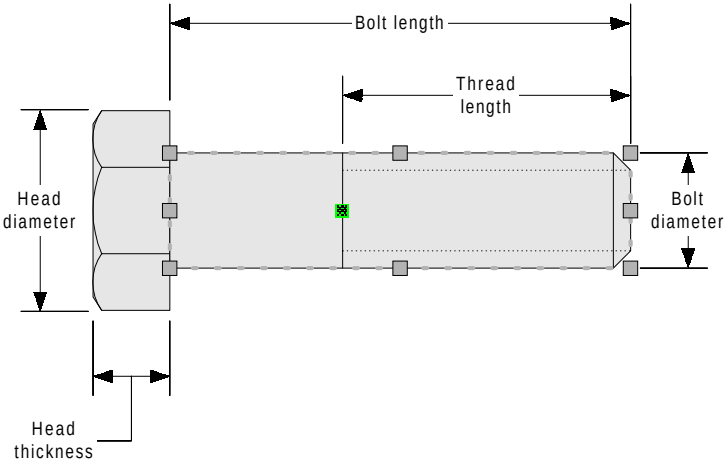
Designing a Visio solution

In a well-designed solution, shapes correspond to objects in the problem domain, and creating a drawing constructs a model. Shapes are designed to encourage correct modeling and graphical representation, but they don't prevent the user from overriding default behavior to produce a readable representation. Other parts of the solution work together with the shapes to help the user create a correct model.

Drawing with objects

If you're accustomed to thinking about graphics as a collection of vectors, Visio lets you think about graphics in a whole new way. Visio shapes are *parametric*—that is, they can adjust their geometry and other attributes according to the values of certain parameters. Instead of fixed geometry based on hard-coded *x,y* coordinates, a shape's geometry is based on formulas that recalculate dynamically as a user manipulates the shape. Instead of drafting with lines, you're drawing with objects.





In this bolt shape, the bolt length, thread length, and bolt diameter are parameters that are controlled by formulas. The head diameter and head thickness are derived from these parameters.

These parameters are independent of each other, within practical physical limits. The user could set them by dragging the selection handles to change the bolt length or bolt diameter, or by dragging the control handle to change the thread length. A program could set them with numerical data from a manufacturer's database of available sizes.

Designing shapes

The best solutions in Visio often begin right on the drawing page, where you design shapes. Although you could define much of the custom behavior your solution needs in an external program, you'll get superior results faster by taking advantage of the built-in functionality of Visio shapes. If you design intelligence into your shapes, you can build a more flexible solution that requires less coding and maintenance in the long run.

Chapters 3 through 9 in Part II, "Developing Visio Shapes," discuss techniques for developing custom shapes.

SmartShapes technology

Using Visio SmartShapes technology, you can develop shapes that behave like the objects they represent in the real world, modeling the characteristics that are meaningful for the kinds of diagrams you need to create. You do this by defining formulas that make the shapes behave the way they should according to the design rules, codes, or principles that apply to the corresponding objects.

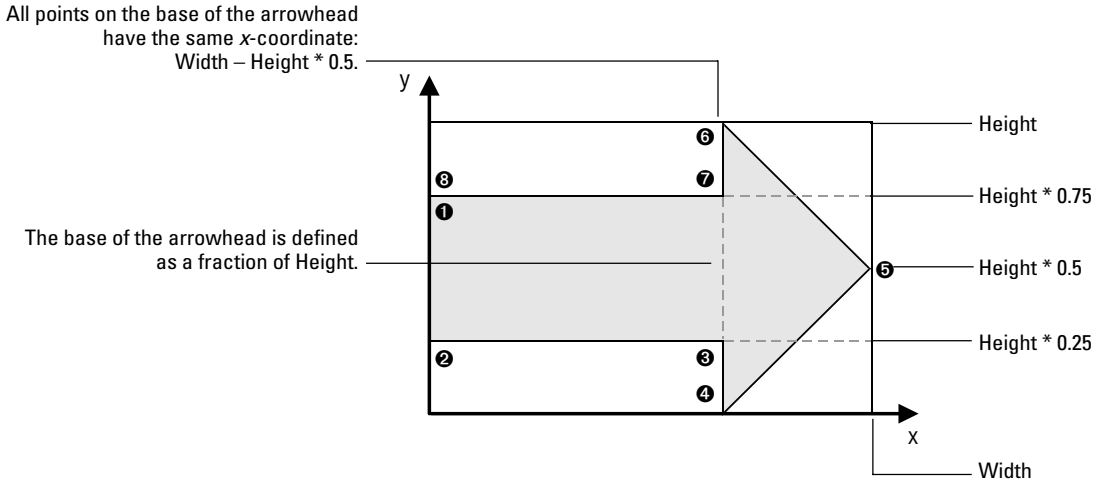
Every Visio shape has its own ShapeSheet spreadsheet, which defines the shape's unique behavior and capabilities. Think of the ShapeSheet spreadsheet as the property sheet of a shape, in which each property is set by a value or formula that is recalculated dynamically as the user works with the shape.

Many features that you might expect to require external programming can be controlled through the ShapeSheet window. For example, you add menu items to a shape's shortcut menu by defining formulas in its ShapeSheet window. Formulas can control other attributes of a shape, such as:

- Geometry (flipping, rotation, visible or hidden paths).
- Color, pattern, and line weight.
- Text, including font, paragraph formatting, and orientation.
- Control handles that help users adjust the shape.
- Connection points where other shapes can be glued.
- Custom properties that can contain user data.

The spreadsheet interface makes it easy to use cell references to link one shape property to another, which means that shape properties can influence each other in subtle and powerful ways. For example, you might link the color of a shape such as a part in a mechanical drawing to its dimensions, to indicate whether the part is within tolerance.

This arrow shape is a classic example of controlling a shape with ShapeSheet formulas. Its formulas override the default behavior given to shapes by Visio, which is to size proportionately when the shape is stretched horizontally or vertically. When this arrow shape is sized horizontally, its custom formulas allow the tail to stretch or shrink horizontally but leave the arrowhead unchanged.



Custom formulas in the ShapeSheet Geometry section

Geometry1	X	Y
❶ Start	= 0	= Height * 0.75
❷ LineTo	= 0	= Height * 0.25
❸ LineTo	= Width – Height*0.5	= Height * 0.25
❹ LineTo	= Geometry1.X3	= 0
❺ LineTo	= Width	= Height * 0.5
❻ LineTo	= Geometry1.X3	= Height
❼ LineTo	= Geometry1.X3	= Height * 0.75
❽ LineTo	= Geometry1.X1	= Geometry1.Y1

For a detailed discussion of this example, see "Controlling how shapes stretch and shrink" in Chapter 3, "Controlling shape size and position."

Shapes as components

Just as a procedure in a program encapsulates functionality so that it is easier to use and reuse, Visio shapes encapsulate behavior on the drawing page. Think of a Visio shape as a component whose default behavior is provided by Visio, and whose unique behavior is provided by the ShapeSheet formulas you write.

You provide shapes to users by giving them a standalone stencil that contains your shapes as masters. Users (or your programs) can drag and drop masters from the stencil into a Visio drawing. The stencil makes your custom shapes easy to reuse—the same shapes can be used by an engineer to simulate a product configuration, by a salesperson to show customers what they're buying, or by a graphic artist to create a catalog of your product line.

Dropping a master into a drawing creates an instance of the master and, the first time the master is dropped, adds a local copy of the master to the drawing. Each instance inherits from the local master, which means that the instance can support a lot of complex behavior while remaining relatively small. Any formula can be overridden at the instance level, but global changes can be propagated to instances by altering the local master. And the drawing is portable because it has local copies of masters—the stencil or stencils that originally provided the masters are no longer required. All that's needed to view the drawing is a copy of Visio.

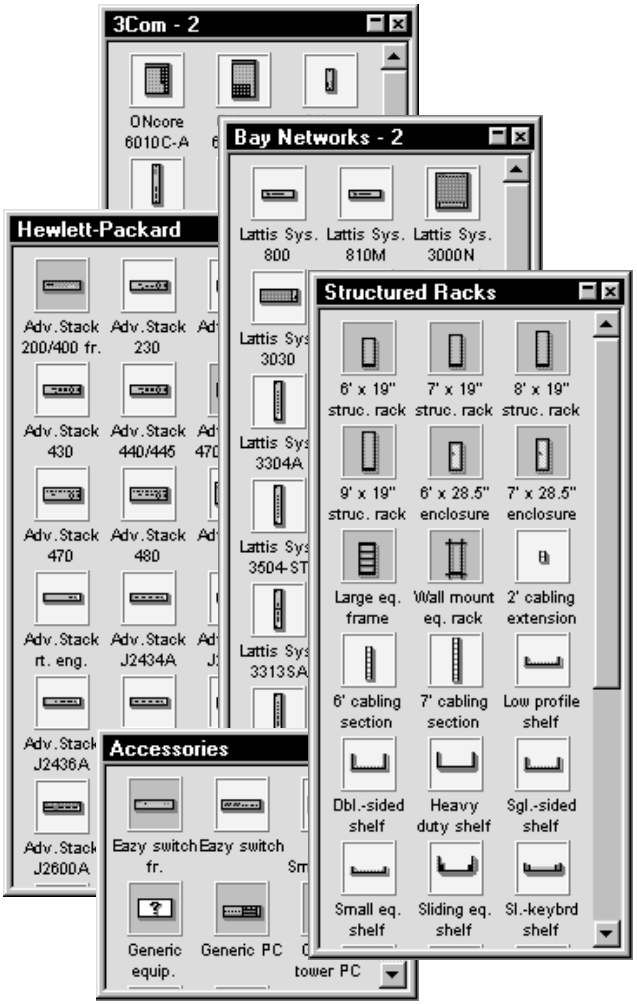
Packaging shapes in a solution

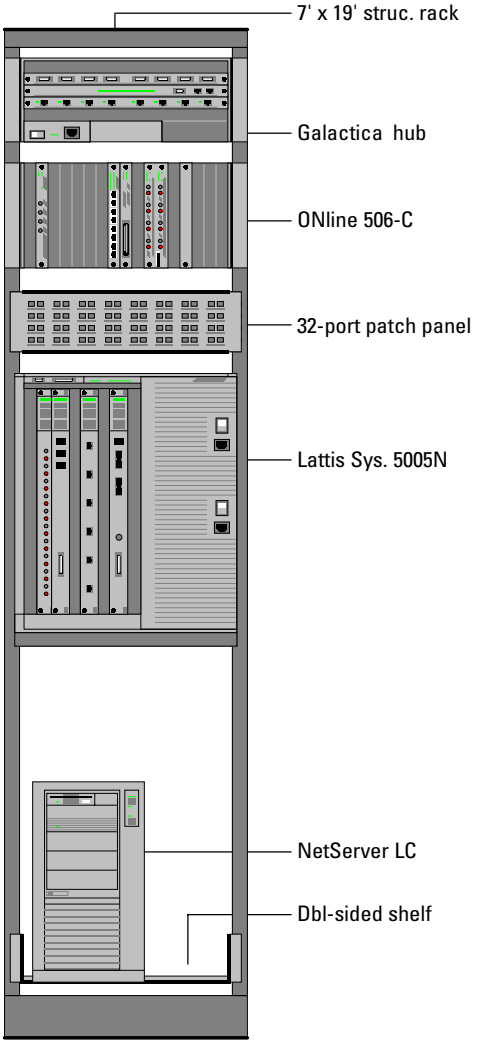
A stencil is like a catalog or gallery of components—your custom shapes—from which the user constructs a model. Design the masters so that the user does not have to draw anything by hand, if possible. For details about creating masters and stencils, see Chapter 2, "Tools for creating solutions."

In addition to a stencil, a Visio solution typically includes styles, a template, and online help:

- Styles give shapes consistent text, line, and fill formats. Shapes formatted with styles inherit their formatting from the definition of the style in the drawing file, so you can reformat drawings quickly and easily by redefining the style. For details about defining styles, see Chapter 7, "Managing styles, formats, and colors."
- A template is a Visio file that is used to create new drawings. A template can open the correct stencils, and establish the correct scale, drawing size, and paper size for a drawing. It can contain predrawn elements like title boxes, logos, and frames. A template can also provide cells, values, and formulas that are added to the ShapeSheet window of each new drawing page, and provide VBA macros that are copied to new drawings. For details about creating templates, see "Creating templates" in Chapter 2, "Tools for creating solutions."
- Shapes and templates can be linked to topics in a standard Microsoft Windows help file to guide the user toward correct usage of your shapes. For details about creating Windows help files, see the documentation provided with your development environment. For details about linking Visio shapes and templates to online help, see "Adding help to masters" in Chapter 9, "Packaging stencils and templates."

These network equipment shapes are designed to align and connect with the equipment rack shapes, so a network designer can create an accurate model of a server room. Individual shapes match the manufacturer's specifications for a precise fit, and the shape designer customized the shapes' alignment boxes and added connection points to make the shapes easier to use.





For details about customizing how shapes connect and align, see Chapter 5, "Making shapes connect: 1-D shapes and glue" and Chapter 8, "Scaling, snapping, and aligning."

Integrating data with shapes

The appearance of a shape in a drawing, however sophisticated, is rarely the whole story. The real-world object that a shape represents often has important data associated with it—part numbers, prices, quantities ordered or in the warehouse; names, dates, addresses, telephone numbers; manufacturers, suppliers, customers; dimensions, materials, tensile strength. Having this kind of data in a drawing makes it a powerful tool for analysis and communication.

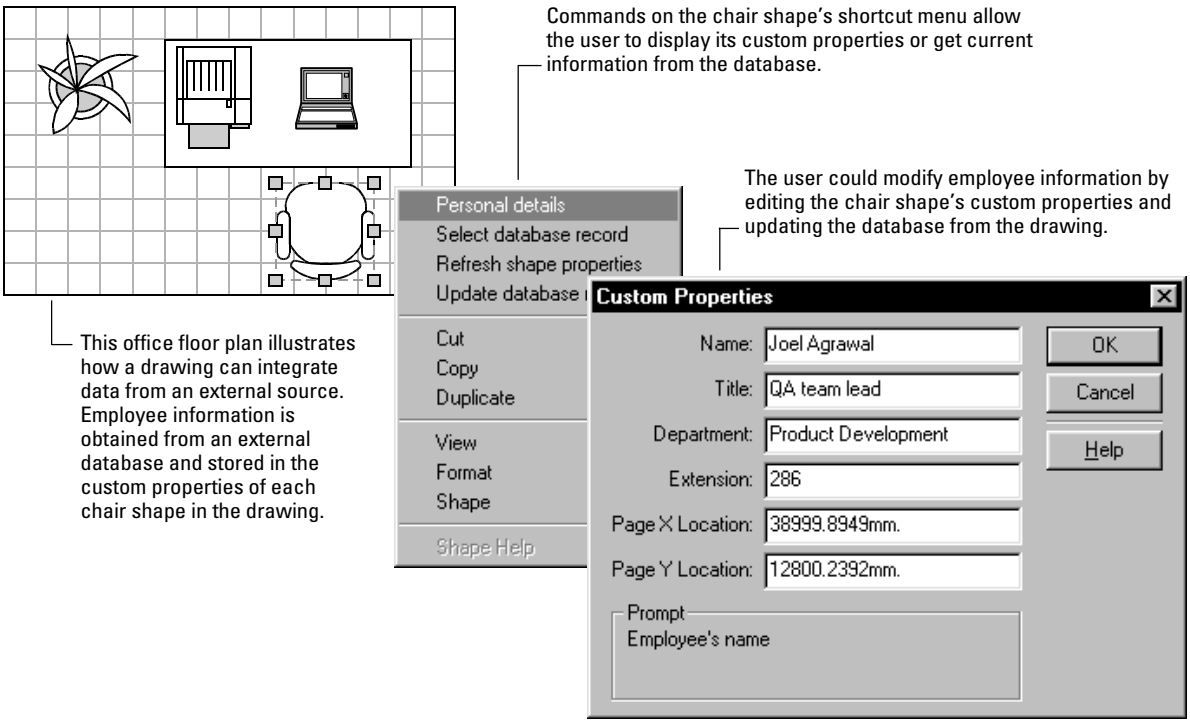
You can associate data with a Visio shape by defining custom properties in its ShapeSheet window. You give each custom property a unique name and optionally define other characteristics, such as data type, format, and default value. You can insert data into custom properties in any of these ways:

- **Add the data when you create the shape.** For example, you might fill in custom properties for resistance, voltage, and amperage in masters that represent electronic components. When the user drops one of the shapes in a drawing, the data accompanies the shape.
- **Collect the data from the user.** Visio can prompt the user to fill in custom properties of a master each time it is dropped in a drawing, encouraging the user to enter the data you need. The user can also display and edit a shape's custom properties from its shortcut menu.
- **Transfer data from an external source.** You can set custom properties to data from an external source, such as a spreadsheet or database, by writing an intermediary program that uses Automation to direct the flow of data. For more about programming as part of a solution, see "Automating Visio" later in this chapter.

Sometimes data stays behind the scenes, but often you'll want to display data in the drawing or change the drawing as the data changes. You can use a shape's custom properties in any of these ways:

- **Display data in a shape's text.** Insert a text field in the shape's text to display the result of a custom property's formula. A text field can display a value, the result of a formula, or any global value that Visio provides, such as file summary information or the current date and time.

- **Control the shape's behavior.** Because custom properties are stored in ShapeSheet cells, they can play a role in other formulas—for example, a shape's geometry can be linked to its custom properties, allowing the shape to respond to user input.
- **Extract the data from the drawing.** Just as you can use Automation to set custom properties from an external source, you can obtain data from a shape's custom properties and write it to an external destination such as a spreadsheet or database.



Designing shapes for data

Custom properties can serve as containers for data from an external source, or they can provide a data-entry interface for shapes in a drawing. Whether custom property data resides only in the shape or comes from an external source is up to you.

The Database Wizard, provided with Visio 5.0, streamlines the process of connecting your database with Visio shapes. For details about using the Database Wizard, see *Using Visio Products*. For details about defining custom properties, see "Defining custom properties" in Chapter 4, "Enhancing shape behavior."

Custom properties are only one example of how you might integrate external data with Visio drawings and diagrams. For example, a parts database might contain many records representing variations on a theme, such as different head styles, thread lengths, and overall lengths of bolts. You might create a single Visio shape with multiple geometries and the ability to display the correct configuration based on a record from the database. For an example of defining a shape with multiple geometries, see "Defining shortcut menu commands" in Chapter 4, "Enhancing shape behavior."

Automating Visio

Visio supports Automation (formerly OLE Automation), so you can integrate the functionality of the Visio graphics engine and SmartShapes technology with other programs, using a common and familiar programming language. The Visio version 5.0 product line integrates Visual Basic for Applications (VBA), so you don't need a separate development environment. And you can write VBA programs in Visio that control other applications. However, you can control Visio from programs written in any Automation controller. For details, see Chapter 10, "Automation and Visio."

Your program controls Visio by accessing objects that represent the items you want to control, then using their properties and methods to manipulate the objects. Most objects in the Visio object model correspond to items you can see and select in Visio. For example, a Shape object represents a shape in a drawing.

Like the shapes you create, the kind of program you write depends on the solution you're developing. Programs that control Visio can:

- **Generate drawings from external data**. You might create drawings based on user input or records in a database. For example, a business process modeling solution might generate diagrams from a repository of process information, using custom shapes that follow a strict set of rules.
- **Read drawings to extract information or validate the model**. You might read diagrams created with custom shapes to gather information for a database, check accuracy, or create other kinds of documents. For example, a solution that supports field sales might analyze a diagram at a customer site to flag errors or generate an order.
- **Synchronize a drawing with the data it represents**. A model is most effective when you automate the connection between the drawing and the data it represents. For example, a visual corporate directory might use data from an employee database to show each employee in an office space plan, and update the database with a new location when the employee is moved in the drawing.

You can achieve a high level of interactivity and control between your program and Visio by handling events. For details, see Chapter 15, "Handling events in Visio." You can also add ActiveX controls directly to Visio 5.0 drawings. For details, see Chapter 18, "Using ActiveX controls in a Visio solution."

Designing a program to control Visio

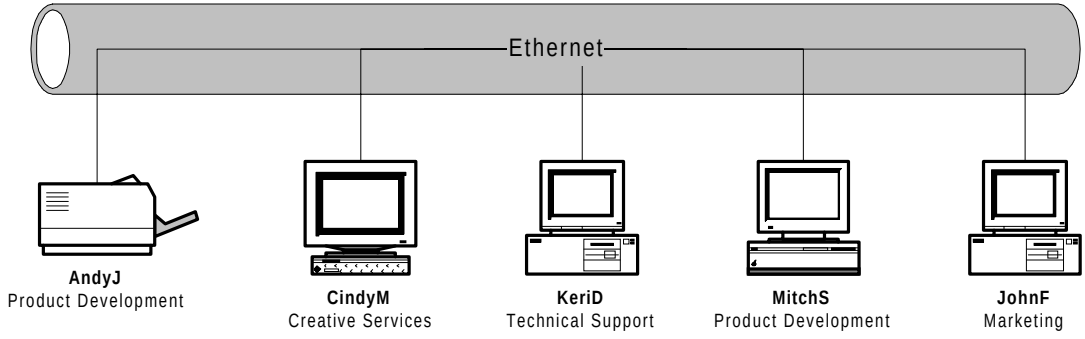
A program can help create a drawing, analyze a drawing, or transfer information between a drawing and external data sources. Or a program can simply enhance the behavior of a particular shape. Whatever your program does, design it to manipulate masters and shapes in drawings and provide data that changes dynamically at runtime, rather than create shapes with code and static data.

The chapters in Part III, "Extending Visio with Automation," discuss techniques for writing programs that control Visio through Automation in VBA, Visual Basic, and C++.

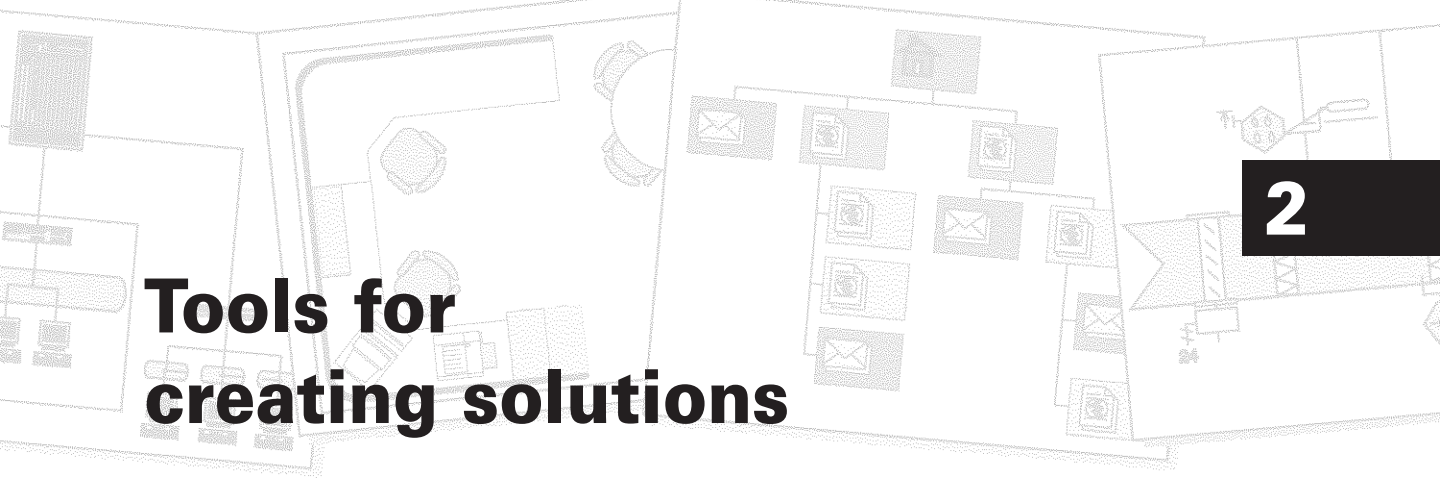
The following code is from the program NetDB, which creates a simple network diagram from a database table, using masters from a stencil of network shapes. The Visio template that contains the program is in \DVS\VBA SOLUTIONS.

Code that generates a Visio drawing

```
Set NetStencil = Visio.Documents("netdb.vss")
Set NetDiagram = ThisDocument.Pages(1)

While Not NetInfo.EOF
   NodeType = NetInfo.Fields("Node")
   Set Master = NetStencil.Masters(NodeType)
   Set Shape = NetDiagram.Drop(Master, XPos, 0.875)
   XPos = XPos + 1.5

   Label = NetInfo.Fields("Name")
   Label = Label & vbCrLf
   Label = Label & NetInfo.Fields("Dept")
   Shape.Text = Label
   Shape.Data1 = NetInfo.Fields("Spec")

   Set ControlCell = Ethernet.Cells("Controls.X" & Chr$(Digit))
   Digit = Digit + 1
   Set ConnectCell = Shape.Cells("Connections.X5")
   ControlCell.GlueTo ConnectCell
   NetInfo.MoveNext
Wend
```



For a detailed discussion of this program, see Chapter 12, "Creating Visio drawings from a program."



2

Tools for creating solutions

Most often, a graphic solution starts with a roughed-out shape. As you work with a shape, you think of better ways for it to address your drawing task—perhaps by adding special formulas that control shape behavior. If you want the shape to provide data to or read data from an external program, you can write an Automation program using Visual Basic for Applications (VBA). To make your shapes reusable, you can save them as masters on a stencil, and then distribute your solution as a self-contained template that includes VBA macros, customized stencils, and unique drawing page settings. Through a process of successive refinement, you can create a complete graphic solution.

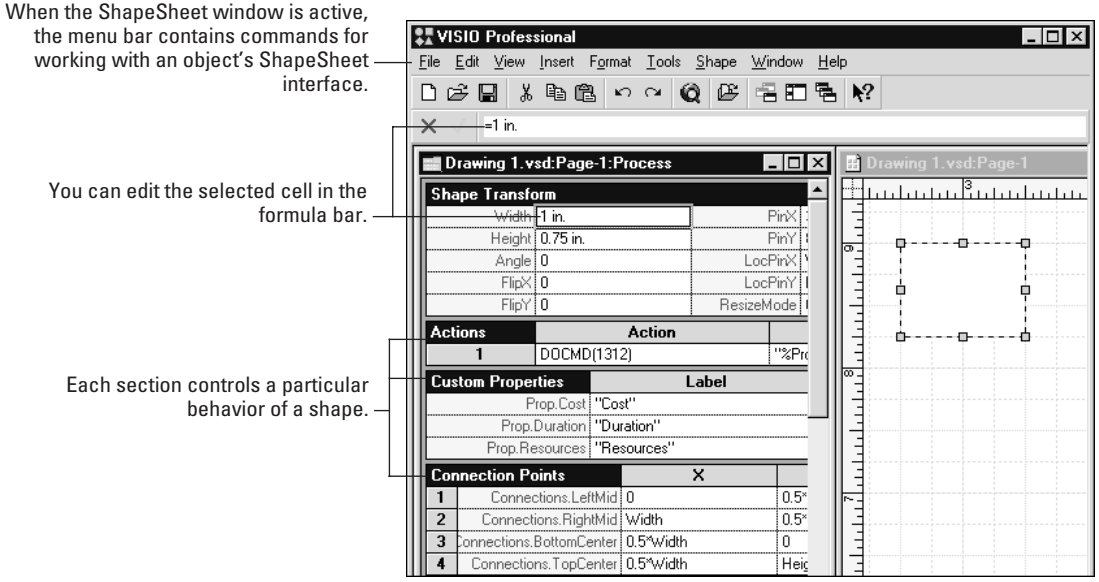
Visio provides the tools you need to create your solution. This chapter describes how to work in the ShapeSheet window; how to create masters, stencils, and templates; how the Visio file format affects your solution; and how to use the VBA editor.

Topics in this chapter

Working with the ShapeSheet window

The shape you see on a drawing page is created according to a set of instructions that appear as values and formulas in cells of the ShapeSheet window. This window provides another view of a shape—a view that gives you precise control over the appearance and behavior of a shape. You can edit these cell values and write new formulas to dynamically recalculate shape attributes.

Shapes, groups, masters, objects from other applications, pages, guides, and guide points are represented by a ShapeSheet interface. In groups, the group is represented by a ShapeSheet interface, as is each object in the group. In this section, *object* refers to anything in Visio that is represented by a ShapeSheet interface.



Parts of a ShapeSheet interface

To display the ShapeSheet interface for a shape, group, guide, guide point, or object from another application:

1. Select the object.

   To select a shape within a group, open the group window and select the shape there. For details, search online help for "groups: editing."

2. From the Window menu, choose Show ShapeSheet. Or click the Show ShapeSheet button on the Developer toolbar.

TIP To add the Show ShapeSheet command to shapes' context (right-click) menus, choose Options from the Tools menu, click the Advanced tab, and check Run In Developer Mode. This option also adds the Add-Ons submenu to the Tools menu.

To display the ShapeSheet interface for a page:

- Make sure nothing is selected, then from the Window menu, choose Show ShapeSheet. Or click the Show ShapeSheet button on the Developer toolbar.

To display the ShapeSheet interface for a master:

1. Open the stencil file containing the master you want.

   Make sure Original or Copy is checked in the Open Stencil dialog box.

2. In the Visio stencil window, double-click the master.
3. In the master drawing window, select the shape, then from the Window menu, choose Show ShapeSheet. Or click the Show ShapeSheet button on the Developer toolbar.

Displaying ShapeSheet sections

The ShapeSheet window is divided into sections of labeled cells that define related aspects of object behavior and appearance. Only sections that are available for the selected object appear in the ShapeSheet window. For example, every shape has a Shape Transform section, which contains general positioning information for the shape, but a page sheet does not include or need this section. To save space, Visio does not display all the sections available for an object. When you want to work with cells in a section that is not visible, you can show that section if it is available for the selected object.

You can expand and collapse sections in the ShapeSheet window by clicking the section name.

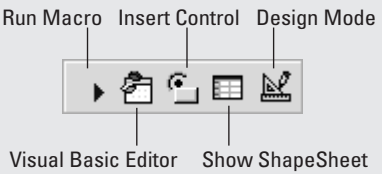
To show or hide sections:

1. Click the title bar of the stencil window, then choose Sections from the View menu.
2. In the Sections dialog box, check the sections you want to show, or uncheck the sections you want to hide, and then click OK.

   If a section is dimmed, it is not available for the selected shape.

Using the Developer toolbar

The Developer toolbar buttons are shortcuts for various Visio menu commands.



To display the Developer toolbar, choose Toolbars from the View menu, then choose Developer.

The Run Macro, Visual Basic Editor, and Show ShapeSheet buttons are described in this chapter. For details about the Insert Control and Design Mode buttons, see Chapter 18, "Using ActiveX controls in a Visio solution."

Some ShapeSheet sections appear only after you explicitly add them. For example, to create a command that appears on a shape's shortcut menu, you can add the Actions section.

To add a new section in the ShapeSheet window:

1. From the Insert menu, choose Section.
2. In the Insert Section dialog box, check the sections you want to add, and then click OK.

For details about each ShapeSheet section and the cells it contains, search online help for "shapesheets: sections."

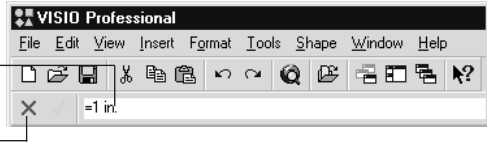
Entering and editing formulas

The key to controlling shape actions is to write *formulas* that define the behavior you want. A formula is the expression in a cell that can contain constants, cell references, functions, and operators, and that evaluates to a *value*. In the ShapeSheet window, you can display cell contents as either values or formulas by choosing the appropriate command on the View menu.

You can edit a cell's formula to change how the value of the cell is calculated and, as a result, change a particular shape's behavior. For example, the Height cell in the Shape Transform section contains a formula that you can edit to change the shape's height. You enter and edit formulas in the ShapeSheet window much the same way you work in any spreadsheet program, with two key differences:

- Visio regards anything in a cell—even a numeric constant, string, or cell reference—as a formula.
- Many cells assume that a value is dimensional, so anything you enter in them implies a unit of measure. (For details, see the following section.)

To enter a formula, you select a cell and then start typing in the formula bar, as the following figure shows.



Entering a formula in the formula bar

For details about entering and editing formulas or working in the formula bar, search online help for "formulas" or "formula bar."

TIP Right-click a ShapeSheet cell to display its shortcut menu, which contains commands you can use to edit the cell.

Referencing another cell in a formula

You can create interdependencies among ShapeSheet formulas by means of a cell reference. You can refer to a ShapeSheet cell of either the current shape or another shape on the same page. Cell references give you the power to calculate a value for one cell based on another cell's value.

A reference to a cell in the same shape specifies only the cell name, such as Height. For a cell in a section with indexed rows, the section and row index are part of the cell name. For example:

```
Connections.X5
```

which specifies the cell in column X, row 5, of the Connections section. A reference to a cell in another shape requires an object name or ID (identifier) followed by an exclamation point and cell name. For example:

```
Sheet.2!Width
```

How shapes inherit formulas

When you open the ShapeSheet window, the formula you see in a cell can be one that is inherited from a master or a style (called an *inherited formula*), or one that you entered in the cell (called a *local formula*). Rather than make a local copy of every formula for a shape, an instance inherits formulas from its master on the local stencil and from the styles applied to it.

This behavior results in smaller files, but also allows changes to the master's formulas or the style definition to be propagated to all instances. When you write a local formula in the sheet of an instance, you are replacing the inherited formula in the cell, which then no longer inherits its value from the master.

An exception is when you apply a style, which always writes new values into the corresponding ShapeSheet cells and so can overwrite even locally edited cells. For details, see Chapter 7, "Managing styles, formats, and colors."

Black text in a cell indicates an inherited formula. Blue text indicates a local formula—either the result of editing the formula in the ShapeSheet window or some change to the shape that caused Visio to reset the formula for that cell. To replace a local formula with an inherited formula, delete the local formula from the cell. Visio restores the inherited formula from the master.

A shape, group, guide, or guide point always has an ID, whether or not it has a name. Visio assigns the ID when the object is created. This ID does not change unless you move the object to a different page or document. To display the ID or enter a name, choose Special from the Format menu.

A reference to a cell in the page sheet, the ShapeSheet interface for the drawing page, requires the name ThePage followed by an exclamation point and the cell name. For example:

```
ThePage!PageScale
```

In a master, a reference to ThePage refers to the page of the master drawing window.

The following table summarizes rules for cell references in formulas.

Example cell references

Sheet	Cell Reference	Example
Same shape	*Cellname*	Width
Named shape, group, or guide	*Shapename!Cellname*	Star!Angle
Named group or guide in which more than one shape at the same level has the same name	*Shapename.ID!Cellname*	Executive.2!Height
Any shape on the page	*Sheet.ID!Cellname*	Sheet.8!FillForegnd
A cell in a named column with indexed rows	*Sectionname.Columnname[Index]*	Char.Font[3]
A cell in an unnamed column with indexed rows	*Sectionname.ColumnIndex*	Scratch.A5
A cell in the page sheet	*ThePage!Cellreference*	ThePage!User.Vanishing_Point

Cell reference shortcut

To quickly refer to another cell in the same shape, place the insertion point in the formula bar, then click the cell you want. Visio inserts the cell name at the insertion point.

How units are expressed in a formula

Visio evaluates the result of a formula differently depending on the cell you enter it in. In general, cells that represent shape position, a dimension, or an angle require a *number-unit pair* that consists of a number and the qualifying units needed to interpret the number. For example, a formula in the Width cell that evaluates to 5 means 5 units of measure, such as inches or centimeters. Many other cells are unitless and evaluate to a string, to true or false, or to an index. For example, the formula *=5* in the FillForegnd cell means color 5 from the drawing's color palette, and in the LockWidth cell means TRUE (and locks the shape's width).

Always specify a unit of measure when you enter a formula in a cell that expects a dimensional value. Doing so makes it easier to identify the number-unit pairs in your calculations, so that you don't inadvertently divide one number-unit pair with another number-unit pair, or combine incompatible units, such as adding angles to lengths. In addition, specifying units of measure makes it easier to localize your formulas for international use, if that's a consideration.

If you don't specify units of measure, Visio evaluates a number using the default units defined for the cell, which can be page units, drawing units, or angular units. *Page units* measure sizes on the printed page, including typographic measurements. *Drawing units* specify the real-world measurement, such as a 50-meter pool (drawing units) that appears 10 cm long (page units) on paper. For example, if you enter the formula *=50* into the Width cell, which expects a number-unit pair in drawing units, Visio supplies the default drawing units currently set for the page and evaluates the formula accordingly.

Example calculations using number-unit pairs

Do	Don't
`5 in.`	`5`
`Width + 0.5 in.`	`Width + 0.5`
`7 in. * 1.5`	`7 * 1.5`
`DEG(MODULUS(Angle, 360 deg.))`	`MODULUS(Angle, 360 deg.)`

Specifying units of measure

Because many drawings represent physical objects, Visio accepts units of measure in the English and metric systems, and you can specify angles in either radians, decimal degrees, or degrees, minutes, and seconds. You can also use standard typographical measurements such as picas, points, ciceros, and didots.

ShapeSheet sections for writing formulas

Most ShapeSheet sections have a predefined purpose: Their cells contain shape attributes or behaviors. To write a formula that serves as an intermediate calculation or that stores values used by another formula or add-on, use the cells in the User-Defined Cells section or the Scratch section. These two sections contain cells that do not correspond to specific shape attributes or behaviors, so you can use either or both to contain any formula. These sections do not appear in the ShapeSheet window unless you add them by using the Section command on the Insert menu.

You can add a cell whose value and name you specify in the User-Defined Cells section. A user-defined cell can contain any formula, such as a constant referenced in other formulas or a calculation used by an add-on. For example, a master can refer to a user-defined cell in a page. When an instance of the master is created, the instance refers to the user-defined cell of the page it is on.

The name you give to a user-defined cell must be unique within a section. To refer to the value of a user-defined cell in the same shape, use the syntax User.*name*. For example, to refer to a user-defined cell Constant of the shape with the ID Sheet.2, use this expression:

```
Sheet.2!User.Constant
```

NOTE The User.Prompt cell, Action.Prompt cell, and certain other cells are designated by default to contain a string. When you type in these cells, Visio automatically encloses the text in quotation marks. If you start the formula with an equal sign (=), though, Visio treats the entry as it treats any other formula (and doesn't add quotation marks).

The Scratch section is displayed with six columns labeled X, Y, and A through D. Place calculations involving dimensions or shape coordinates in the X and Y cells of the Scratch section, which use the drawing's units of measure. The A through D cells are unitless, and so are appropriate to use for logical and string expressions. To refer to cells in the Scratch section, specify the section name and the column and row label; for example, Scratch.A1.

User-defined and Scratch cells

You can use either a user-defined or a Scratch cell for assorted calculations, but there are times when it makes more sense to use one or the other.

For example, the Scratch section has X and Y cells, which are designated to contain a number-unit pair in drawing units. These cells are good places to put formulas involving shape coordinates.

Because you can provide a meaningful name for a user-defined cell, it's a better place to store constants and values referred to in other formulas. References to a meaningful cell name make formulas easier to read.

When you need to exchange information between an object and an external program, it may be safer to use a user-defined cell. Any program can write to a Scratch cell and so overwrite formulas you place there. This is less likely to happen in a cell with a unique name.

Creating masters and stencils

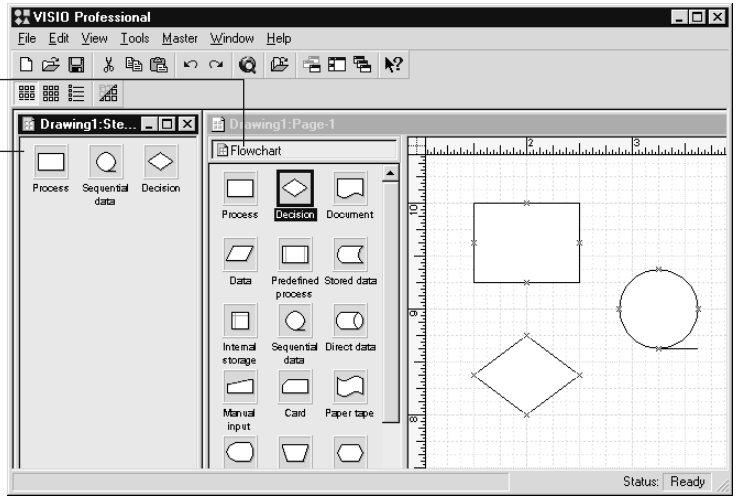
Your solution probably involves several customized shapes, which you can distribute as reusable *masters* (or master shapes) on a stencil. A master is a shape, group, or object from another application that is saved on a stencil, which can be opened in other drawings. You can create a shape on the drawing page, then drag it into a stencil to create a new master. Or you can use commands available in the stencil window to create a new master. These commands are available only when you open a stencil as an original file with write access.

To reuse the masters you create, you save them on a *standalone* stencil, which is a file with the extension .VSS. You can save any Visio file as a standalone stencil. By default, Visio opens a standalone stencil as read-only. When you want to work with a stencil or the masters it contains, you must open the original file or a copy of it. Unless otherwise specified, when we talk about stencils in this book, we are referring to standalone stencils.

When a user drags a master from a stencil onto a drawing page, Visio creates a copy of that master on the *local* stencil of the drawing and creates an *instance* of the master on the drawing page. A drawing file always includes a local stencil that contains copies of the masters used in the drawing, even if the corresponding shapes are later deleted from the drawing page. An instance is linked to the copy of its master on the local stencil and inherits its behavior and appearance from that master.

Typically, when you open a template, its standalone stencil is opened as a read-only file in a docked window.

To display the local stencil for a drawing page, choose Show Master Shapes from the Window menu.



Stencils in a Visio drawing file

Opening a stencil

One way to create a stencil is to open a new, empty file as a stencil. Because the new file's drawing page is empty, you can more easily keep file size to a minimum, and the file contains only the default styles until you add masters to the stencil.

To open a new, empty stencil:

- From the File menu, choose Stencils, then choose Blank Stencil.

  Visio opens a new stencil file with write access.

You can open the original version of an existing stencil so that you can add new masters to it or edit the ones already there, and then save the revised stencil as a new file.

To open a stencil as an original file with write access:

1. From the File menu, choose Stencils, then choose Open Stencil.

   The Open Stencil dialog box appears.

2. Select the stencil file you want to revise.

3. Under Open, select Original, then click Open.

   Visio opens the stencil as an original file with write access.

You can quickly create a new stencil with the masters already in it by saving the local stencil of a drawing file as a .VSS file. The stencil you create this way will contain all the masters used during the drawing session, including masters whose instances you have since deleted from the drawing page. You may want to edit the local stencil and clean up the drawing page before saving it as a new stencil file.

To create a new stencil from a drawing's local stencil:

1. If you want to view the masters before saving them, from the Window menu, choose Show Master Shapes.

2. From the File menu, choose Save As.

3. Under Save As Type, choose Stencil (*.VSS). Enter a name and location for the file, then click Save.

Saving shapes as masters on a stencil

Just as you can drag a master into a drawing to create a shape, you can drag a shape or group into a stencil to create a master. The stencil must be opened as an original file. You can create a master from an object that you have pasted or imported into Visio from another program. However, when users create an instance of such a master, they won't be able to edit the shape's vertices, rotate it, or add text to it.

To create a master from a shape in a drawing:

1. In the drawing window, display the page that contains the shape you want to use as a master.
2. Make sure the drawing window is active, then drag the shape from the drawing window into the stencil window. Or hold down the Ctrl key to drag a copy of the shape.

   Visio creates a default name and icon for the shape in the stencil window.
3. To realign icons after adding a master, choose Arrange Icons from the View menu.
4. To save your changes to the stencil file, make sure the stencil window is active, and then choose Save from the File menu.

   If you are creating a new stencil, type a new name for the stencil. Under File Type, select Stencil. To protect the stencil from accidental changes the next time it is opened, under Save, select Read Only. Click OK.

Masters in local and standalone stencils

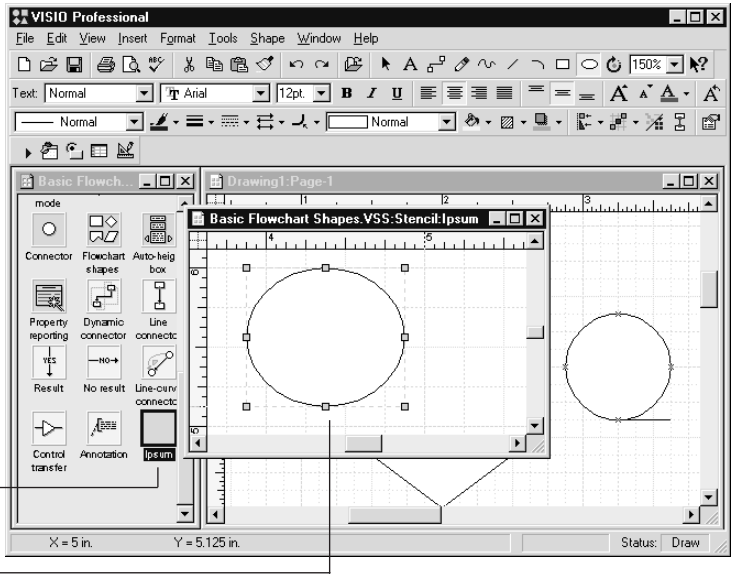
A local stencil is the stencil associated with a template's drawing page. A standalone stencil is a document with the .VSS file extension. When you drag a master from a standalone stencil onto the drawing page, Visio copies the master to the local stencil, then creates an instance of the master on the drawing page. In this fashion, the local stencil receives copies of all masters used during drawing sessions. An instance of a shape inherits its styles from the master on the local stencil. As a result, each Visio document can be moved easily from machine to machine.

When you drag an instance of a master onto the drawing page, Visio decides whether to copy the master to the local stencil. If the Match Master By Name On Drop option is checked in the master's Properties dialog box, then Visio links the instance to the master on the local stencil with the same name. Otherwise, Visio uses the master's time stamp to determine if the master needs to be copied.

Each master on a standalone or local stencil is internally stamped with the date and time it was last updated. When you edit the copy of a master on the local stencil, its time stamp no longer matches that of the master on the standalone stencil. When you next drag an instance of the original master from the standalone stencil, Visio makes a new copy of the master on the local stencil and gives it a unique name. The new instance is linked to the new master on the local stencil.

Working with masters on a stencil

You can add new, blank masters to a stencil or edit the existing masters. When you open a stencil with write access, you can edit masters by opening the master drawing window as the following figure shows. To specify the attributes of master shapes and icons, you can use commands on the Master menu, which Visio displays in the menu bar when an original stencil is active.



You can add a new, blank master to a stencil, then edit the master and its icon.

You can draw and edit on the master's drawing page using the same techniques you use on the document's drawing page.

The master drawing window displays the drawing page for a master.

To create a new, blank master:

1. Make sure the stencil window is active, then choose New Master from the Master menu.
2. In the New Master dialog box, under Master Name, type a name for the master, then click OK.

   Visio creates a blank master and adds an icon to the bottom of the stencil. Edit the master and its icon using the commands on the Master menu or on the master's shortcut menu.

   For details about other options in the New Master dialog box, click the Help button in the dialog box.

To edit a master:

1. In the stencil window, double-click the master you want to edit.

   The master drawing window appears, which contains the drawing page associated with the master.

2. When you are finished editing the master, close the master drawing window.

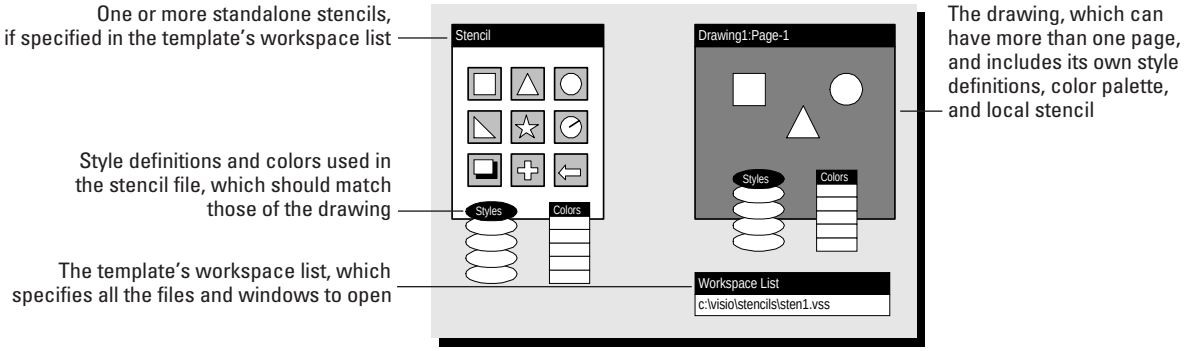
   Visio asks if you want to update the master. Click Yes. Visio also updates the master icon to reflect the changes you have made, unless the Manual option is checked in the master's Properties dialog box. For details about refining master icons, see "Finishing and testing a stencil" in Chapter 9, "Packaging stencils and templates."

Creating templates

To work more efficiently in Visio, you can save the page settings, styles, shapes, macros, and stencils you use most as a template. With a template, you can put everything in one place, which makes it simple to deliver your custom solution to users. In general, to create a template, you open a new or existing drawing file, set the options you want, open the stencils you want, and then save the file as a template. The drawing page of a Visio template file is typically blank, but your template can include shapes on the drawing page, such as a title block or a company logo, or even more than one drawing page.

You can save any Visio file as a template, which can include:

- A workspace list identifying one or more stencils, which are opened when you open a new drawing file with the template.
- One or more drawing pages, including backgrounds. Each page can contain a drawing and can use a different size and scale.
- VBA macros.
- Print settings.
- Styles for lines, text, and fill.
- Snap, glue, and layering options.
- A color palette.
- Window sizes and positions.



Typically, when you open a file as a template (.VST), you open at least two documents, a stencil file and a drawing file, which contain the elements shown.

To create a template:

1. Open the drawing file on which you want to base the template. Or open a new drawing file.
2. Open the stencil file (or files) that you want to open with the template.

   Open each stencil file as read-only. If you open the stencil file as an original, it will be saved that way in the template's workspace list.
3. Activate the drawing window, and then change or define options and settings that you want to include in the template.

   For example, you can define the styles you want to include, set page display options, and select a drawing scale.
4. If you want a drawing page to contain any standard elements, create the appearance you want. You can insert additional pages as either foreground or background pages.
5. From the File menu, choose Properties. In the Properties dialog box, type information about the template, then click OK.

   The text you type under Description appears when you select the template in the Open and Choose A Drawing Template dialog boxes.
6. From the File menu, choose Save As.

   Under Save, check Workspace. From the File Type list, select Template (*.VST). In the File Name box, type a name for the template, and then click OK.

Workspace or workspace list?

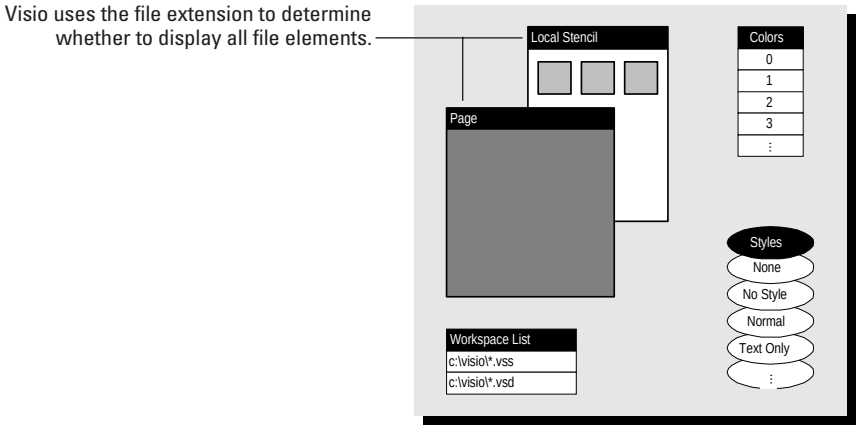
Visio saves the workspace list with a file if the Workspace option is checked in the Save As dialog box. A workspace list differs from a workspace (.VSW) file, which Visio creates when you use the Save Workspace command on the File menu.

Visio typically opens a template's stencils in docked, read-only windows. However, a template can open some stencil files docked and others floating, some as read-only and others as original. A template's workspace list stores the names of the stencil files to open as well as the type, size, and position of window to display them in based on their appearance when you saved the template.

NOTE If you are creating a template for scaled drawings, the page scale is set by the template's drawing page. The master scale is determined by the scale at which the shape is drawn. To avoid unexpected behavior, masters and drawing pages should use the same or a similar scale. For details, see Chapter 8, "Scaling, snapping, and aligning."

Opening and saving Visio documents

When you work in Visio, the document you open, how you open it, and how you save any changes you make depends on the file type. A Visio document can be a drawing (.VSD), stencil (.VSS), template (.VST), or workspace (.VSW). In fact, all of these file types have the same format. Visio uses the file extension to determine what to display onscreen when the document is opened. This means, for example, that you can save a drawing file (.VSD) as a template (.VST), which you can then open and work with as a template.



Elements of a Visio document

Each Visio document always has:

- At least one *drawing page*.
- A *local stencil* that contains copies of any masters used on the drawing page (or, in the case of a .VSS file, that displays the masters).
- A *workspace list*, which identifies all of the windows and files that are opened with the current file.
- A list of styles defined for the file, including at least the four default Visio styles (No Style, None, Normal, and Text Only).
- A color palette of 24 user-modifiable color slots and up to 232 additional colors defined by RGB (red, green, blue) or HSL (hue, saturation, luminosity) formulas in the document.

A document can also contain shapes on the drawing page, with styles and colors applied from those stored in the document, as well as VBA projects with modules, class modules, and user forms.

Opening a Visio file

You can open a Visio file as an original document with read/write access, as an original document with read-only access, or as a copy of the original document. When you choose Open from the File menu, these options appear in the Open dialog box. For example, when you open a stencil file with read/write access, Visio displays the Master menu with commands for editing masters and icons.

In addition, Visio uses the file extension to determine which windows in a document should be active. For example, when you open a stencil file, its drawing window is closed and only its local stencil is displayed. When you open a drawing file, its stencil window is closed and only the drawing page is displayed. You can display the windows that are closed by default for a Visio file:

- To display the drawing window for a stencil file, choose Show Drawing Page from the Window menu.
- To display the local stencil for a file, choose Show Master Shapes from the Window menu.

The following table shows how items appear by default for each file extension when opened.

How Visio opens a file

File	Default contents
.VSD	Drawing. Opens all windows and files listed in the workspace, if it was saved with the file. If not, Visio creates a drawing window and displays the page that was open the last time the file was saved.
.VSS	Stencil. Opens the stencil as read-only (in a docked window, if a drawing window is active). If a drawing window is not active, Visio creates a stencil window and displays the file's stencil.
.VST	Template. Opens an untitled copy of the drawing in a drawing window, and opens all windows and files listed in the workspace. If there is no workspace, Visio creates a drawing window and displays a new empty drawing.
.VSW	Workspace. Opens in the appropriate windows all files listed in the workspace, then closes the .VSW file.

Saving your work

You can take advantage of the different Visio file types to work more efficiently. Here are some tips for saving your work:

- You can save a file's local stencil as a .VSS file to quickly create a new standalone stencil of frequently used shapes.
- If you have two or more .VSD files open at once, you can save the arrangement of all the open windows by using the Save Workspace command on the File menu. That way, you can open the .VSW file to open all the drawing windows in the positions you last left them. (Checking Workspace in the Save As dialog box saves only descriptions of the open windows in the workspace list of the file being saved.)
- If you're saving stencil and template files that are meant to work together, make sure that their drawing page settings, styles, and colors are compatible. For details, see "Using styles in stencils and templates" in Chapter 7, "Managing styles, formats, and colors."

- If you're working on a document that you want others to review but not change, save the file as read-only. To do this, check Read-Only in the Save As dialog box.

  Users can open and edit a copy of a read-only file, but the original file cannot be edited. After you have saved a file as read-only, to make the file read/write again, use the Save As command to save the file to another name.

The Visio online help contains procedures for saving different types of files and workspaces. For details, search online help for "saving."

Programming Visio with VBA

Previous sections of this chapter have described development tools available in Visio. This section discusses a VBA program that controls Visio by accessing the objects that Visio exposes. Then, your program manipulates the objects by getting and setting their properties, such as shape text or fill; invoking methods on the objects, such as adding or deleting shapes; and triggering code when an event occurs, such as when a document is created or a page is deleted.

To build your VBA program, you can add modules, class modules, and user forms to the default project contained in every Visio file. A Visio file can store only one project, but that project can consist of any number of modules, class modules, and user forms. A *project* is a collection of items to which you add code. A *module* is a set of declarations followed by procedures—a list of instructions that your program performs. A *class module* is a special type of module used to create an object definition that contains the object's properties and methods. A class module acts as a template from which an instance of an object is created at runtime. A *user form* is a container for user interface controls, such as command buttons and text boxes.

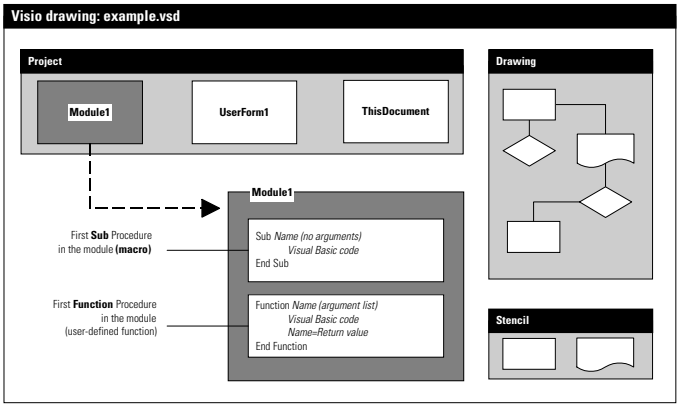
The following illustration shows how procedures are contained in modules that are collected in a project, and how a project is stored in a Visio file.

Other development environments

In addition to using VBA, you can write programs that control Visio in any development environment that supports Automation, such as Visual Basic and C/C++. If you use Visual Basic or C/C++, you can write a standalone executable program or, with C/C++, a Visio library (.VSL), which is a Visio-specific Windows dynamic-link library.

For details about controlling Visio from a Visual Basic program, see Chapter 19, "Programming Visio with Visual Basic."

For details about controlling Visio from a C++ program, see Chapter 20, "Programming Visio with C++."



An example of how a Visio drawing may look after adding items to the default project and saving the drawing

How you combine modules, class modules, and user forms depends on the type of solution you are providing. At a minimum, each project contains a ThisDocument class module by default. The ThisDocument class module represents the properties, methods, and events of the specific document associated with a VBA project. The ThisDocument class module is a custom object you can reference in other VBA programs. For details about ThisDocument, see Chapter 11, "Using Visio objects."

Simple projects may consist of just one user form or module. More complex projects may consist of multiple modules, class modules, and user forms. For example, to build a user interface for your program, you must design a dialog box, data-entry form, or wizard screen to tell the user what to do, and you must gather any information the program needs to run. You must then decide how the user will interact with your program. For example, users can choose your program from the Macros dialog box, or your program can trigger code in response to a document event, such as opening a document.

To build a VBA program, first insert a module, class module, or user form. Then insert a procedure or enter code for an existing event procedure in the item's *code window,* where you can write, display, and edit code. You can open as many code windows as you have modules, class modules, and user forms, so you can easily view the code and copy and paste between code windows.

Automation and VBA in this book

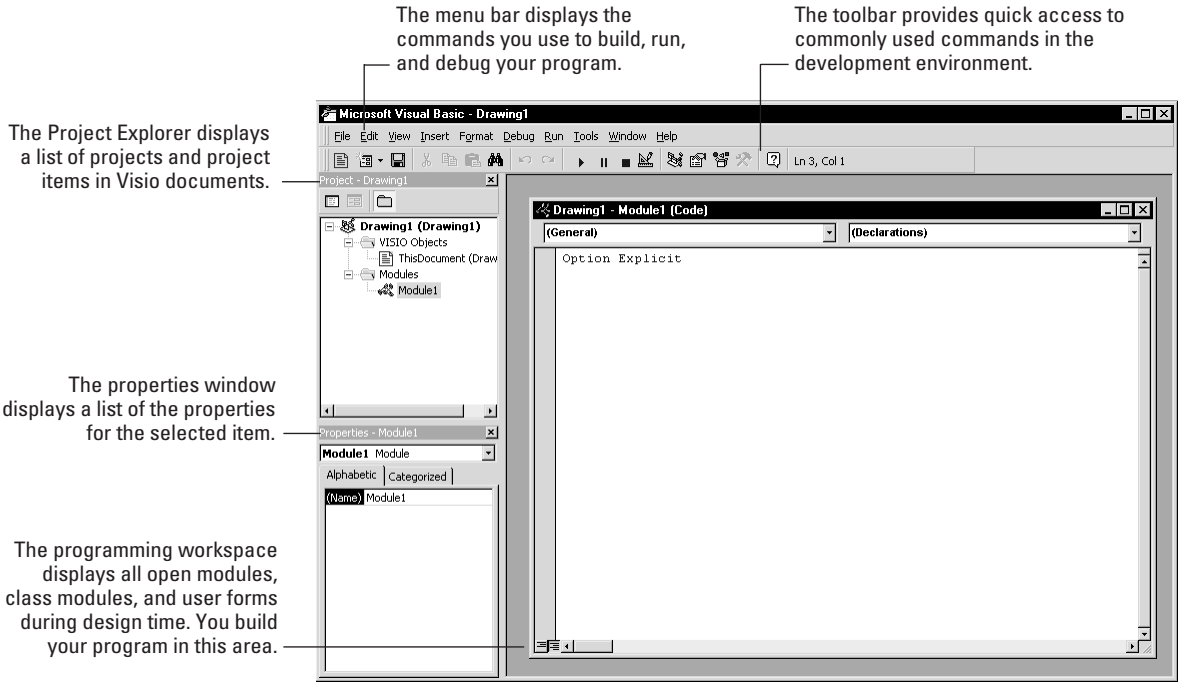
The chapters that explain how to use Automation and VBA to control Visio assume you are familiar with Visual Basic programming and terminology. They offer only brief definitions of important VBA terminology when necessary. For details about programming with Visual Basic and VBA terminology, see the Microsoft Visual Basic online help reference.

Getting started with VBA

To begin programming with VBA in Visio, you start the Visual Basic Editor. You can start the Visual Basic Editor without opening a Visio file, but you cannot open a document's project without opening the document first. To view a stencil's project, make sure you open the stencil as an original or copy.

To start the Visual Basic Editor:

1. Start Visio, then open a template, stencil, or drawing.
2. From the Tools menu, choose Macro, then choose Visual Basic Editor. Or click the Visual Basic Editor button on the Developer toolbar.



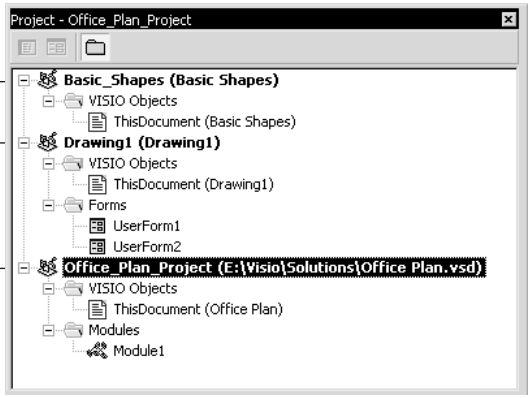
Visual Basic Editor: the VBA development environment

To navigate among projects in the Visual Basic Editor, use the Project Explorer. It displays a list of the modules, class modules, and user forms for the projects in all Visio files open with read/write access.

The Basic Shapes stencil and the items in its project; open with read/write access.

An open Visio drawing and the items in its project; this drawing hasn't been saved.

An open Visio drawing, saved as Office Plan, and the items in its project.



The Project Explorer displaying three Visio files open in an instance of Visio

You can customize a VBA project name, enter a project description, or lock a project by choosing <Project Name> Properties from the Tools menu and setting project properties. To save a Visio file and its VBA project, choose Save <File Name> from the File menu. After you save the file, the file name and location are displayed in the Project Explorer in parentheses after the project name.

Inserting modules and class modules

Many VBA programs contain one or more modules—a set of declarations followed by procedures. All Visio VBA projects contain the class module, ThisDocument, which is a custom Visio object that represents the specific document associated with a VBA project.

Modules and class modules can contain more than one type of procedure: sub, function, or property. You can choose the procedure type and its scope when you insert a procedure. *Inserting* a procedure is like creating a code template into which you enter code.

Scope is the area in which a procedure is accessible by other modules and programs. Every procedure has scope. A procedure with a *private* scope is limited to the module that contains it—only a procedure within the same module can call a private procedure, and a private procedure does not appear on any menus or in any dialog boxes.

Toggling between windows

To return to the Visio window from the Visual Basic Editor, choose Visio from the View menu. Or click the View Visio button on the Standard toolbar.

To return to the Visual Basic Editor, choose Macro from the Tools menu, then choose Visual Basic Editor.

Other programs and modules can access a *public* procedure. Visio displays public procedures that take no arguments on the Macro menu.

You can also declare the variables in your procedure as local or global. *Global* variables exist for the lifetime of your entire program, and *local* variables exist only while the procedure in which they are declared is running. The next time the procedure is executed, all local variables are reinitialized. However, you can preserve the value of all local variables in a procedure for the lifetime of your program by making them static (fixing their value).

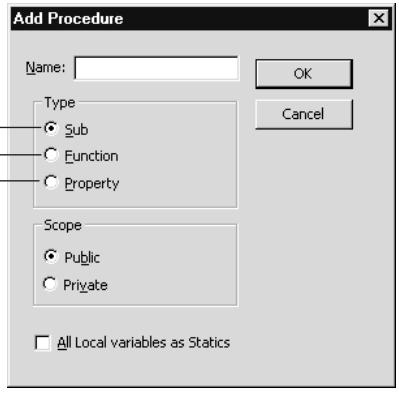
To insert and begin programming in a module or class module:

1. From the Insert menu, choose Module or Class Module.
2. From the Insert menu, choose Procedure.

To write a macro or procedure that takes no arguments, insert a Sub procedure.

To write a function that takes arguments and returns a value after running the procedure, insert a Function procedure.

To add properties to a class module, insert a Property procedure.



The Add Procedure dialog box

3. In the Name box, name the procedure.

   The name of a macro is displayed under its module's cascading menu in the Visio Macro menu. A procedure name cannot include spaces or reserved words that VBA uses as part of its programming language such as MsgBox, If, or Loop.
4. Under Type, select the type of procedure: Sub, Function, or Property.
5. Under Scope, select Public or Private.
6. To declare all local variables as static, check All Local Variables As Statics.

Displaying macros on Visio menus

Visio displays the name of a module on the Macro menu and the names of the macros it contains on the module's cascading menu. You can choose instead to display your macros prominently on the Macro menu so your users can find them more easily.

To display macro names on the Macro menu instead of the module's cascading menu, name the module that contains the macros *ShowInMenu*. A module named ShowInMenu does not appear on the Macro menu, but its public macros do.

For an illustration that shows the Visio Macro menu, see "Running a VBA program," later in this chapter.

7. Click OK.

   VBA inserts a *procedure template* into the item's code window into which you can enter code. The template contains the first and last lines of code for the type of procedure you insert.

8. Enter code into the procedure template.

For details about procedures, see the Microsoft Visual Basic online help reference.

Inserting user forms

If you want your program to prompt the user for information, you must build a user interface for your program by inserting user forms. A *user form* is a container for user interface controls such as command buttons and text boxes. A *control* is a Visual Basic object you place on a user form that has its own properties, methods, and events. You use controls to receive user input, display output, and trigger event procedures.

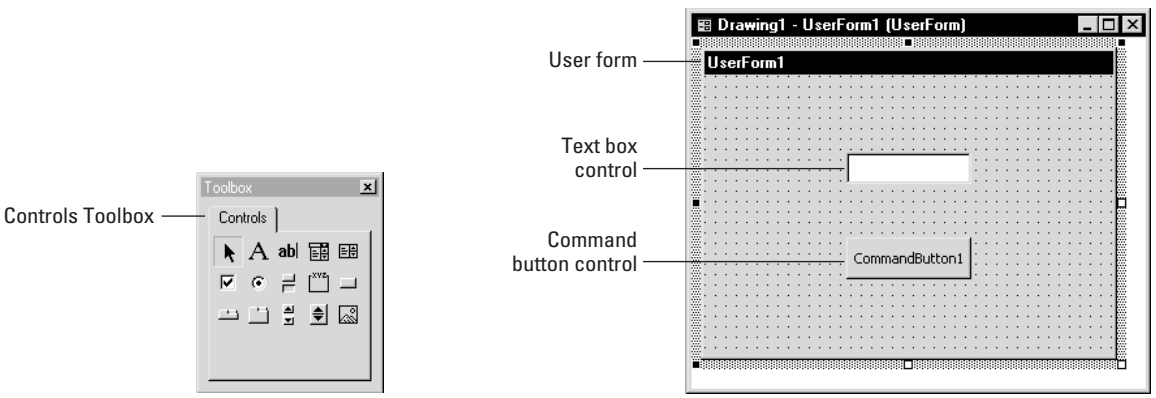
To insert and begin programming in a user form:

1. From the Insert menu, choose UserForm.

   VBA inserts a user form into your project and opens the Controls Toolbox.

2. Select the controls that you want to add to the user form from the Controls Toolbox.

   For details about adding controls, such as command buttons and text boxes, see the Microsoft Visual Basic online help reference.



Toolbox and user form containing controls

3. Add code to the user form or controls.

   To display the code window for a user form or control, double-click the user form or control. Then you can choose the event that you want your code to trigger from the drop-down list of events and procedures in the code window and start typing your code. Or insert a procedure and start typing your code in the procedure template.

Using the Visio type library

To increase the speed of your programs and make writing your programs easier, use a type library. The Visio *type library* is a file that contains Automation descriptions of the objects, properties, methods, events, and constants that Visio exposes.

Using the Visio object types declared in the Visio type library increases the speed of your program because VBA interprets Visio objects at design time rather than runtime. When you compile a program during design time, VBA checks for syntax and programming errors and matches object types against type libraries. If you use a general variable type, such as Object, VBA doesn't interpret it until runtime when it queries Visio about object references. This extra querying step decreases the speed of your program.

You can take advantage of the Visio type library to write code more effectively by:

- Using Visio object types instead of general variable types.
- Viewing Visio Automation descriptions for objects, properties, methods, events, and constants in the Object Browser.
- Copying code templates from the Object Browser.

Using Visio object types. Earlier versions of Visio didn't include a Visio type library; you defined an object variable as an *Object* and used it to hold a reference to a Visio object. For example:

```
Dim pagObj as Object
```

Setting a reference to a type library

To use a type library, your project must reference it. Visio VBA projects automatically reference the Visio type library, but when using an external development environment, such as Visual Basic, you need to set a reference to the Visio type library. For details, see Chapter 19, "Programming Visio with Visual Basic."

By using Visio object types declared in the Visio type library, you can declare variables as specific types, such as *Visio.Page:*

```
Dim pagObj as Visio.Page
```

Using a Visio object type such as *Visio.Page* enables your program to check the type of object it is referencing in the Visio type library at design time. In this example, you've used *Visio* to inform your program that you are referencing Visio object types in the Visio type library, and you've used *Page* to inform it that the *pagObj* variable is a Page object. You can more specifically define any Visio object with the Visio type library. For details about using Visio objects, see Chapter 11, "Using Visio objects."

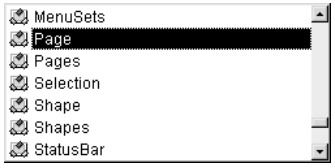
Here are a few common Visio object types:

```
Dim docsObj as Visio.Documents  'a Documents
                                'collection
Dim docObj as Visio.Document    'a Document object
Dim shpsObj as Visio.Shapes     'a Shapes collection
Dim shpObj as Visio.Shape       'a Shape object
Dim mastObj as Visio.Master     'a Master object
```

To see a list of Visio object types, browse the pop-up box that appears after you enter a period after an object or library type. It lists available object types, properties, and methods specifically for the preceding object or variable type. You can use this pop-up box to quickly choose a Visio object type. To do this, scroll through the list until you find the appropriate entry and double-click the entry to enter it in your code.

The pop-up box in the following example lists some Visio object types. In this example, the appropriate object type is Page.

```
Dim pagObj as Visio.
```



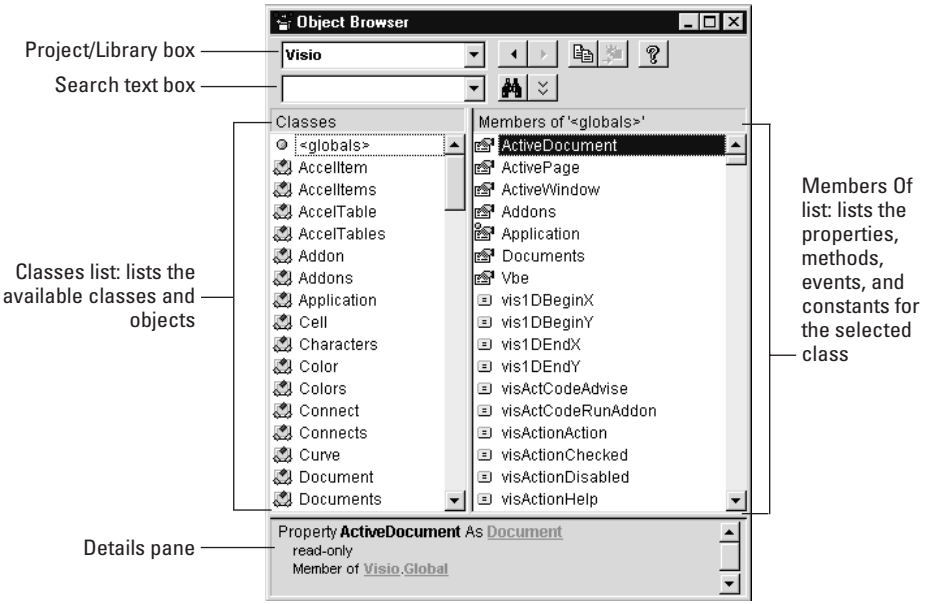
VBA code examples

The VBA code examples in this book and the samples on your Visio 5.0 CD use Visio object types.

Viewing Automation descriptions. You can view the Visio Automation descriptions and copy code templates using the Object Browser. The Object Browser displays constants, classes (objects), and class members (properties, methods, and events) of type libraries referenced by a project.

To display the Object Browser, choose Object Browser from the View menu. To browse or search the Visio objects, properties, methods, events, and constants, type the name of the object, property, method, event, or constant you want to search for in the Search text box or just click any member in the Members Of list.

NOTE To view the class and members of the Visio type library only, choose Visio from the Project/Library box.



The Object Browser

Copying code templates. The Object Browser displays Visio properties, methods, events, and constants as *members* in the Members Of list. The Details pane displays the syntax for the members as a code template that you can copy and paste or drag and drop into a module and substitute the appropriate variables and arguments. Using code templates decreases the chance of typing errors.

Managing a VBA project

Writing code is only the first step in the process of creating a VBA solution. When you are working on several projects, you can streamline your efforts by managing those projects well. To work effectively as well as minimize maintenance tasks down the road, you can use these project management practices:

- Reuse modules, class modules, and user forms to save time writing code.
- Remove project items that are no longer needed to save file space.
- Protect your code, if necessary, from being viewed or modified by users.

Reusing modules, class modules, and user forms. To import an item into your project, choose Import File from the File menu. You can choose any VBA module (.BAS), user form (.FRM), or class module (.CLS) to add a copy of the file to your project. To export an item from your project so that it will be available for importing into other projects, select the item you want to export in the Project Explorer, then choose Export File from the File menu and enter the location in which you want to save the file. Exporting an item does not remove it from your project.

You can also drag and drop projects or project items between Visio files by selecting the project or project item you want to move in the Project Explorer and dragging its icon onto a Visio project icon. A project item is automatically stored in the correct project folder. A project is referenced in the References folder because a Visio file can contain only one project, but that project can reference other projects.

NOTE You cannot drag and drop the ThisDocument class module between Visio files, but you can drag and drop or copy and paste code from it into other project items.

Removing project items. When you remove an item, it is permanently deleted from the project list—you cannot undo the Remove action. Make sure remaining code in other modules and user forms doesn't refer to code in the removed item. To remove an item, select it in the Project Explorer, then choose Remove <Name> from the File menu. Before you remove the item, you are asked if you want to export it. If you click Yes in the message box, the Export File dialog box opens. If you click No, the item is deleted.

Protecting your code. To protect your code from alteration and viewing by users, you can *lock* a project. When you lock a project, you set a password that must be entered before the project can be viewed in the Project Explorer. To lock your VBA project against viewing, choose <Drawing Name> Properties from the Tools menu, then click the Protection tab and select Lock Project for Viewing. Then enter a password and confirm it. Finally, save your Visio file and close it.

The next time you open the Visio file, the project is locked. If anyone wants to view or edit the project, he or she must enter the password.

Saving a VBA project

VBA projects are stored in a Visio file that can be a template (.VST), stencil (.VSS), or drawing (.VSD). When a user creates a new document from a Visio file, Visio copies the VBA project and its items to the new document. To save your Visio file and your VBA project, choose Save from the Visio File menu or Save <File Name> from the File menu in the Visual Basic Editor. Both commands save your Visio file with the project and its items stored in it. After saving the Visio file, its file name and location are displayed in the Project Explorer in parentheses after the project name.

Running a VBA program

While building your program, you can run your program within VBA to test and debug it. This section discusses one common way to run your program in the Visual Basic Editor during design time. For details about running and debugging a VBA program, such as adding breakpoints, adding watch expressions, and stepping into and out of execution, see Microsoft Visual Basic online help.

Customizing the VBA environment

You can customize your working environment in VBA by setting options such as font size, code color, syntax error options, and variable declaration requirements.

To set environment options, choose Options from the Tools menu, click the Editor or Editor Format tab, and then set the options you want.

Add-ons and macros combined

From a user's point of view, whether the program he or she runs is an add-on or macro doesn't make a difference, so Visio combines these programs in dialog boxes. For example, you run a macro or add-on from the Macros dialog box or from the Macro menu.

To run your program in the Visual Basic Editor:

1. From the Tools menu, choose Macros.
2. In the Macro list, select the macro you want, then click Run.

If the macro you want is not listed, make sure you've chosen the correct project, module, or drawing in the Macros In box. Private procedures do not appear in any menus or dialog boxes.

To run only one procedure in a program in the Visual Basic Editor:

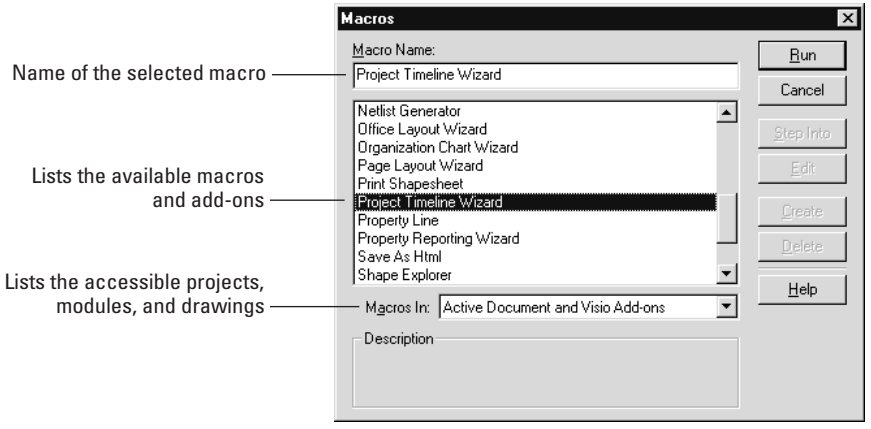
1. In the Project Explorer, open the module that contains the procedure that you want to run.
2. In the code window, click an insertion point in the procedure code.
3. From the Run menu, choose Run Sub/UserForm.

   Only the procedure in which your cursor is located runs.

After you have finished writing a program, users can run it from Visio. To do this, they choose Macro, then Macros from the Tools menu. Or a program can run in response to events or in other ways that you design. For details about running a program in response to events, see Chapter 15, "Handling events in Visio." For other ways to run a program, see Chapter 17, "Running and distributing a solution."

To run your program from the Visio Macros dialog box:

1. From the Tools menu, choose Macro, then choose Macros.
2. In the Macro list, select your program, then click Run.



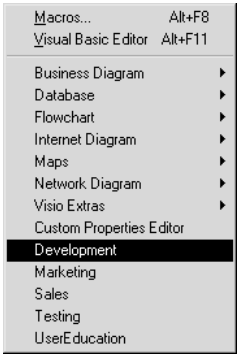
If a macro is not listed, make sure you've chosen the correct project or drawing in the Macros In box. Private procedures do not appear in any menus or dialog boxes.

To run your program from the Visio Macro menu:

1. From the Tools menu, choose Macro.
2. In the Macro list, choose the module that contains your program, then choose your program.

This illustration shows how your module appears on the Visio Macro menu and its macros appear on the module's cascading menu.

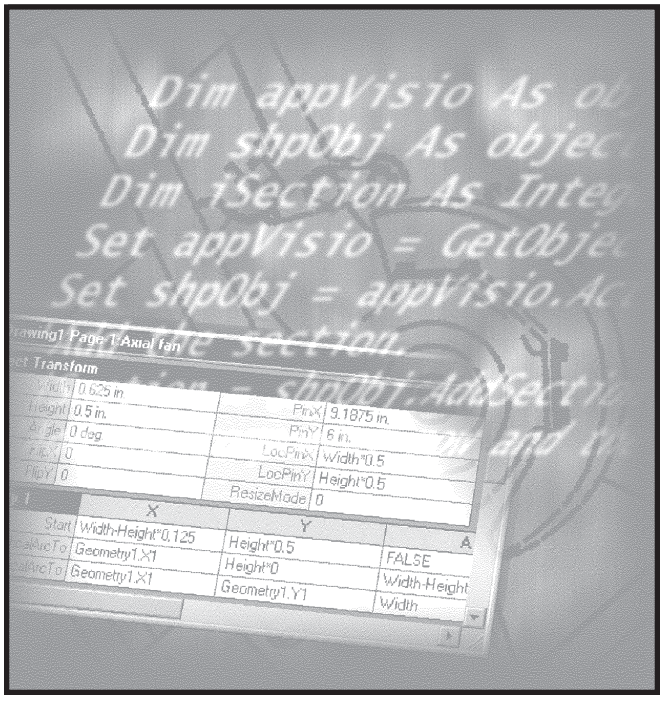
If the module that contains your macros is named ShowInMenu, the ShowInMenu module does not appear on the Visio Macro menu, but its macros do, as the following illustration shows.



Macro description

To enter a description of your macro that appears in the Macros dialog box, open the Object Browser in the Visual Basic Editor, then choose the project that contains the macro in the Project/Library list. In the Class List, select the module that contains the macros, then right-click the macro in the Members Of list and choose Properties. In the Description box, enter a description.

PART II



Developing Visio Shapes

3

Controlling shape size and position

When you design a shape, you must decide how it will respond to user actions, the most common of which are resizing or repositioning the shape. You can define a shape's response by writing formulas in the ShapeSheet cells. Visio records the location of each shape vertex within the shape's coordinate space. These vertices, and the paths that connect them, define the shape's *geometry*. By writing formulas to control shape geometry, you determine how a shape looks and behaves in response to user actions.

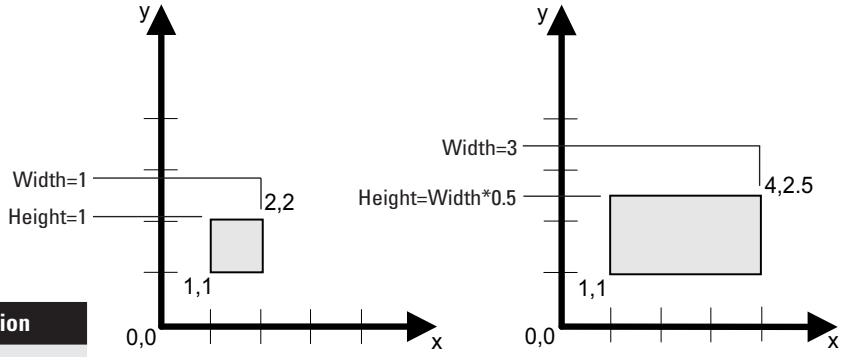
This chapter defines shape geometry and the Visio coordinate system. It then describes how to write formulas to control the size and position of single shapes and groups.

Topics in this chapter

Describing shape geometry

Most drawing programs are based on two-dimensional geometry. When you draw an object, the program records the object as a collection of horizontal and vertical locations. These locations, called *vertices* in Visio, are measured from a point of origin on the page and are connected with line segments, just as if you were drawing the object on a piece of graph paper. The sequence of line or curve segments that connect the shape's vertices is called a *path*. Each path corresponds to a Geometry section in the ShapeSheet window. Each vertex defining a path corresponds to a row of the Geometry section (except in the case of splines—for more on splines, see the sidebar).

Visio records each vertex as a pair of *x*,*y* coordinates. When you move the shape or change its size, Visio records the changes to the shape's vertices and redraws the object at its new position or size. What makes Visio different from other drawing programs is that you can use formulas to control the location of a vertex. Instead of simply recording a new position when a shape is moved or sized, you can calculate a vertex in relation to other vertices or other shapes, or constrain it to a fixed position on the page. The ability to describe shapes with formulas opens many possibilities for making shapes behave in complex and sophisticated ways.



When you create a formula for a shape, Visio recalculates the shape's vertices on the basis of your formula.

Splines and the Geometry section

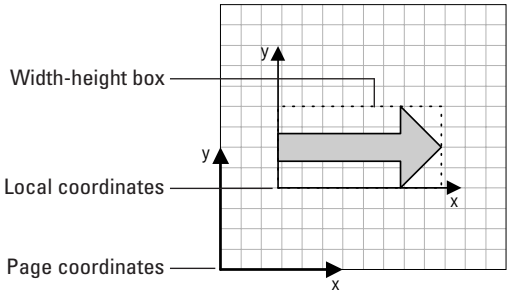
Each vertex of a shape created using the pencil, rectangle, arc, or ellipse tool is represented by one Geometry row. A shape created with the freeform tool, however, is different. The freeform tool creates a quadratic B-spline, a type of curve that includes Bézier curves. A spline is represented by several ShapeSheet rows, whose X and Y cells don't match the handles on the spline. For details about how splines are represented in the ShapeSheet window, see "Working with splines" in Appendix A, "Arcs and splines in Visio."

Describing shapes in a coordinate system

Among the most useful and powerful ShapeSheet formulas are those that control a shape's size or position. To move, flip, rotate, or resize Visio shapes from a formula, you must describe the shape in terms of the Visio coordinate system.

A shape's width and height form the two *local coordinate axes*. The origin is the lower-left corner of the shape's width-height box. The upper-right corner has the coordinates (*Width*, *Height*). The Geometry section uses formulas to describe the local coordinates of the vertices for the paths that make up a shape. By modifying these formulas, you can control a shape's appearance.

The location of a shape on the drawing page is described in *page coordinates*, which have their origin at the lower-left corner of the drawing page. Page coordinates are displayed on the ruler in the units of measure specified in the Page Setup dialog box.



Visio uses different coordinate systems to identify shape vertices and position.

Visio also identifies the position of an object relative to its parent—the group, if it is a grouped object, or the page. *Parent coordinates* are the local coordinate system of the parent. For a shape that is not grouped, the parent coordinates are the page coordinates. For a shape in a group, the parent coordinates are the group's local coordinates. The origin of the parent coordinate system is the lower-left corner of the parent's width-height box if the parent is a group, or the page origin otherwise.

Visio represents a shape's width, height, and position on the page with formulas in the Shape Transform section of the ShapeSheet window, which uses the parent coordinate system. In the Geometry section of the ShapeSheet window, Visio expresses the value of each vertex in a shape as a fraction of the shape's width or height. When you move, resize, or rotate a shape, Visio writes new values in the Shape Transform section, and then reevaluates the vertex formulas in the Geometry section. For example, a shape can have the formula *=3 in.* in its Width cell, and the formula *Width*1* in a Geometry cell. If the shape is stretched, the value of the Width cell can increase to 5 in., which changes the value of the local coordinates specified in the Geometry section. The Geometry formula, however, remains *Width*1*.

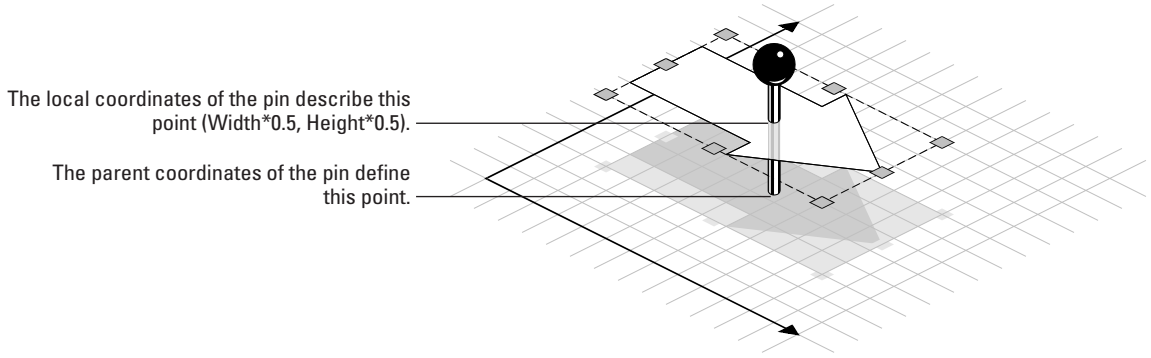
Page coordinates and the rulers

You cannot move the origin of the page coordinate system. You can, however, change the zero point of the coordinates displayed on the rulers by holding down Ctrl and dragging the crossbar at the intersection of the two rulers. Moving the zero point can be useful for measuring the distance between shapes, but it has no effect on the page coordinate system.

Positioning shapes on a page

To describe the position of a shape on the page, you refer to the shape's *pin*, or center of rotation. Visio uses two sets of coordinates in the Shape Transform section to store the location of a shape's pin:

- The PinX and PinY cells store the pin's *x* and *y* location with respect to the parent (the group or page); that is, its parent coordinates.
- The LocPinX and LocPinY cells store the pin's *x* and *y* position with respect to the shape; that is, its local coordinates.



The pin describes a shape's position in local and parent coordinates.

To visualize how the pin works, imagine attaching a 3-by-5 index card to a sheet of paper by pressing a pin through the card, and then through the paper. You can describe the location of the card on the paper with respect to the holes created by the pin. That's how the pin works in Visio. The local coordinates of the pin (the hole in the card) are (*LocPinX*, *LocPinY*). The parent coordinates (the hole in the paper) are (*PinX*, *PinY*). If you pin the card to a different part of the paper—the equivalent of moving a shape on a page—the card's hole doesn't move with respect to the card. That is, the pin's local coordinates do not change. However, a new pinhole is formed on the paper, because the pin's parent coordinates have changed.

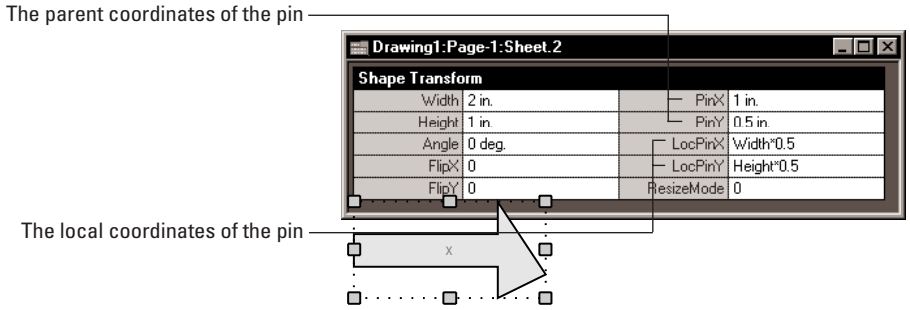
Using formulas to move a shape. To use formulas to move a shape, you set the values of PinX and PinY. For example, to move the arrow in the following figure up the page by 1 inch, use this formula:

```
PinY   = 1.5 in.
```

Or you could tie the arrow's position on the page to the width of the page with a formula such as:

```
PinX   = ThePage!PageWidth - 5 in.
```

By default, the pin is the center of the shape, which Visio expresses as formulas that use local coordinates (*Width*0.5*, *Height*0.5*). You can move the pin by writing new formulas in the LocPinX and LocPinY cells or by choosing a Position option in the grid of the Size & Position dialog box. If you move the pin by using the rotation tool, the values of LocPinX, LocPinY, PinX, and PinY all change so that the shape stays in the same position on the page.



The Shape Transform section includes the local and parent coordinates of the pin.

You can change the values of PinX and PinY by using the Size & Position command on the Shape menu. The X and Y options in the Size & Position dialog box correspond to the values of the PinX and PinY cells. If the formulas in these cells are guarded, however, these dialog box options do nothing. For more on guarding, see the sidebar.

Protecting pin formulas

When a user moves or stretches a shape, Visio writes new values to the Shape Transform section and overwrites the affected cells' formulas, including those in the PinX and PinY cells. To ensure that user actions on the page do not overwrite your pin formulas, you can use the GUARD function. If you guard PinX formulas, users won't be able to move the shape horizontally. If you guard PinY formulas, they can't move it vertically.

You can also set the LockRotate, LockMoveX, and LockMoveY cells to prevent users from rotating or moving the shape. For details about protection locks and the GUARD function, see "Protecting formulas" later in this chapter.

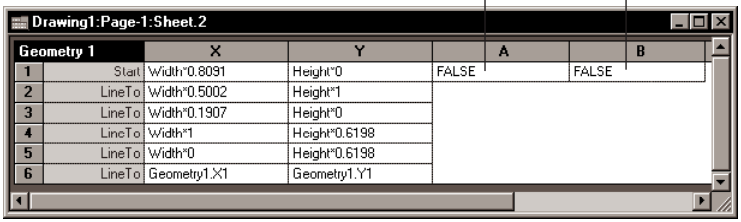
Hiding shape geometry

Every Geometry section includes a NoFill and NoShow cell that control whether a shape can be filled and whether its geometry is visible. If the NoFill cell is set to TRUE, the shape cannot be filled and appears hollow. To hide all shape geometry described by a Geometry section, set the NoShow cell to TRUE.

The NoFill and NoShow cells are not labeled with these names. Although they appear to be the Geometry.A1 and Geometry.B1 cells, the NoFill cell is represented internally as Geometry*n*.NoFill, and the NoShow cell is actually Geometry*n*.NoShow. In cell references, use Geometry*n*.NoFill and Geometry*n*.NoShow to access these cells.

The NoShow cell, which indicates whether the shape is invisible

The NoFill cell, which indicates whether the shape can be filled



Geometry 1		X	Y	A	B
1	Start	Width*0.8091	Height*0	FALSE	FALSE
2	LineTo	Width*0.5002	Height*1		
3	LineTo	Width*0.1907	Height*0		
4	LineTo	Width*1	Height*0.6198		
5	LineTo	Width*0	Height*0.6198		
6	LineTo	Geometry1.X1	Geometry1.Y1		

The NoShow and NoFill cells in the Geometry section

You can use these cells to design shapes for which the geometry is not visible or visible only at certain times. For example, if a shape has multiple Geometry sections, you can hide the shape component represented by one Geometry section, or control which component is filled. For details about adding a shortcut command that controls whether shape geometry is visible, see "Defining shortcut menu commands" in Chapter 4, "Enhancing shape behavior." For details about filling shapes with multiple Geometry sections, see "Merging shapes" later in this chapter.

Controlling how shapes stretch and shrink

Documenting your solution

You can save time and trouble if you document shapes as you develop them, especially those with smart formulas. When you're working with dozens of shapes, it's easy to forget the techniques you've learned while developing a particular shape. If you document your shapes, you're more likely to reuse formulas you've used in shapes, rather than reinvent them.

To document your shapes:

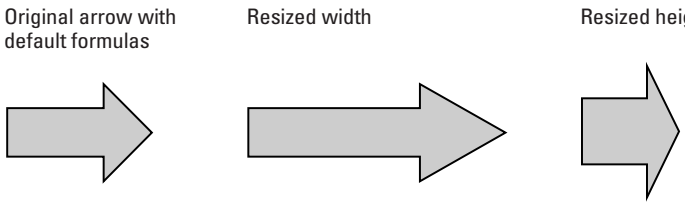
- Write specifications for shapes before you develop them.
- Keep notes about changes to shapes as you make them.
- Document your formulas.
- Keep an inventory of your stencils.

You can use ShapeSheet formulas to control the way a shape shrinks and grows in response to actions of Visio users. Users generally resize a shape by moving its selection handles, but they might also edit a shape's vertices with the pencil tool. The following sections describe how to create a shape that resizes in only one direction and how to create curved shapes that resize the way you want.

Resizing a shape in one direction

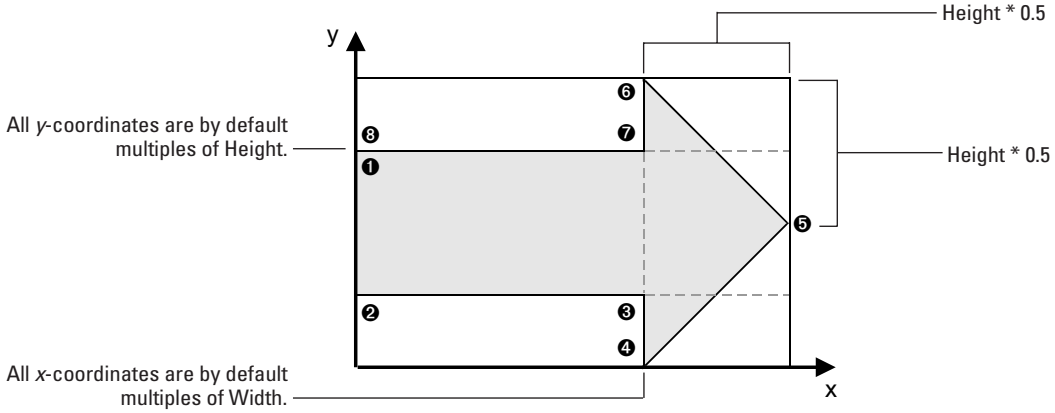
You can design a shape that uses different rules for stretching, depending on whether the user drags a width or a height handle. One such method is to use a *height-based* formula, which preserves a shape's aspect ratio by defining its width in terms of its height. To do this for only part of a shape, you can place a height-based formula in the relevant Geometry cells, depending on which part of the shape you want to control.

An example that shows this type of behavior is an arrow drawn with the line tool. Visio puts default formulas into the arrow's ShapeSheet cells that cause the arrow to resize proportionately when stretched horizontally or vertically, as the following figure shows.



Resizing the original arrow changes the proportions of the shape; arrows of different lengths have different-sized arrowheads, which looks inconsistent.

If you're using the arrow in a chart, its tail should stretch and shrink horizontally to represent different values, but the arrowhead should remain a constant size. If the shape is stretched vertically, the arrowhead should resize proportionately. Because the arrowhead's width is proportionate to its height, a height-based formula can describe the base of the arrowhead (the line connecting vertices 3, 4, 6, and 7 in the following figure) as a fraction of the shape's height.

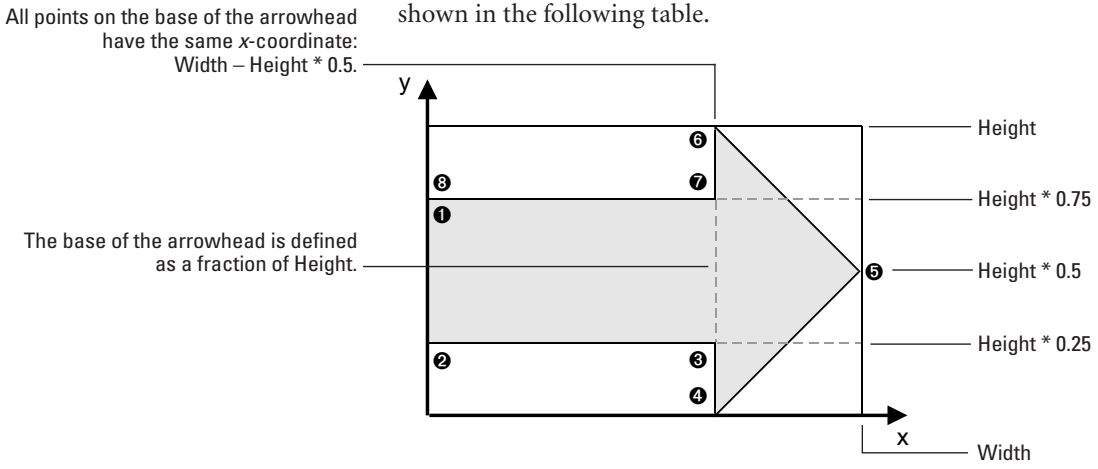


Each vertex corresponds to a line in the Geometry section.

Vertex 5 falls exactly halfway between the top and bottom of the shape, so its *y*-position can be shown as Height*0.5. If we define the *x*-distance from vertex 5 to the base of the arrowhead in terms of the height, the arrowhead will resize proportionately when the shape is stretched vertically, but it will not change when the shape is stretched horizontally. The base of the arrowhead is equal to the width of the shape minus the distance from vertex 5 to the base, or:

```
= Width – Height * 0.5
```

This formula describes the *x*-coordinate of each vertex along the base of the arrowhead (vertices 3, 4, 6, 7). For efficiency, we'll place the formula only in the cell for vertex 3 and refer the other cells to this value. The *x*-coordinate of vertex 3 corresponds to the X3 cell of the Geometry1 section. The resulting geometry for the proportionate arrow is shown in the following table.



Vertices and formulas that describe the geometry of the arrow

Custom formulas in the Geometry section

Geometry1	X	Y
❶ Start	`= Width * 0`	`= Height * 0.75`
❷ LineTo	`= Width * 0`	`= Height * 0.25`
❸ LineTo	`= Width – Height*0.5`	`= Height * 0.25`
❹ LineTo	`= Geometry1.X3`	`= Height * 0`
❺ LineTo	`= Width * 1`	`= Height * 0.5`
❻ LineTo	`= Geometry1.X3`	`= Height * 1`
❼ LineTo	`= Geometry1.X3`	`= Height * 0.75`
❽ LineTo	`= Geometry1.X1`	`= Geometry1.Y1`

Using variations on the height-based formula, you can control the way shapes resize. For details about using a height-based formula with a 1-D shape, see Chapter 5, "Making shapes connect: 1-D shapes and glue."

Reducing calculations in formulas

Because the arrow is symmetrical, you can use cell references to reduce the number of calculations, which makes the shape easier to customize. For example, the Geometry1.Y1 and Geometry1.Y7 cells, which contain this formula:

```
Height * 0.75
```

can also be expressed as:

```
Height – Height * 0.25
```

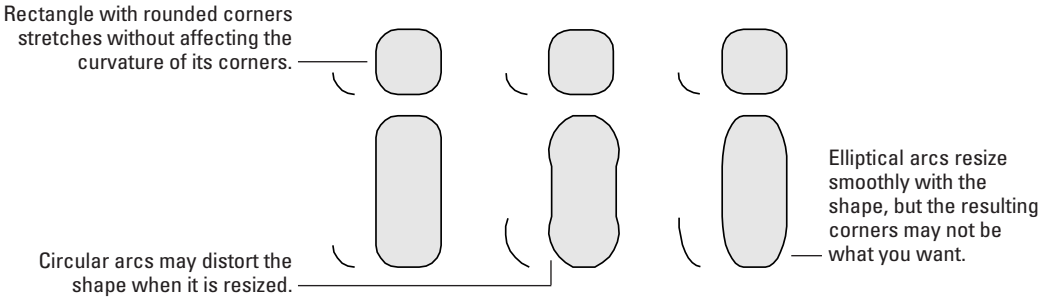
The Geometry1.Y2 cell already contains the formula Height * 0.25, so you can create a cell reference to it. The reduced formula in Geometry1.Y1 and Geometry1.Y7 is then:

```
Height – Geometry1.Y2
```

Now the arrow requires only the two custom formulas, Height * 0.5 and Height * 0.25, to calculate the vertices. And by changing only one formula (Height * 0.25), you can alter the arrow's look.

Creating curved shapes that resize smoothly

When you want to create a shape with rounded corners, you can apply a rounded corner style or create an arc, depending on how you want the shape to resize. The following figure shows the results of using these methods.



How shapes with different corners resize

If you use the Corners command, you apply a rounded style to a line's corners that users can later change by applying a different style. The shape's geometry does not change, only the way it is drawn onscreen. Although a shape with a rounded corner style resizes as expected, it can easily be overwritten by a new line style that specifies different (or no) corner attributes.

If you draw the shape with the pencil, line, arc, or rectangle tool, you can connect the straight portions with an elliptical arc segment. Using arc segments for this purpose is more reliable, because arc segments don't depend on the line or corner style. As a shape is stretched, the beginning and ending vertices of a curve generally move in proportion to the stretching. An elliptical arc can change its eccentricity to maintain smoothness.

You can convert a line or elliptical arc to a circular arc segment by editing the appropriate row in the Geometry section. (See the "Converting line and arc segments" sidebar on this page.) A circular arc tries to fit a circle between the beginning and ending vertices. The result can be a bulge or a sharp edge between a curve and a line. To prevent this distortion, you can control the bow of the arc with ShapeSheet formulas. Creating a shape with rounded corners in this way ensures that the shape's corners span a set angle, so that the corners resize smoothly. For details about writing formulas to control an arc, see "Useful arc formulas" in Appendix A, "Arcs and splines in Visio."

Converting line and arc segments

You can change a straight line segment to an arc segment and vice versa. In some cases you can use the pencil tool; otherwise, you can edit the ShapeSheet interface to convert a path to the line or arc segment you want. The method you use depends on whether you're working with an elliptical or circular arc segment. Do one of the following:

- To change a line to an elliptical arc, select the pencil tool, point to the line segment's control point, then drag to form an arc.
- To change a line to a circular arc, in the ShapeSheet window, select the LineTo row in the Geometry section that represents the segment you want to change. From the Edit menu, choose Change Row Type, then choose ArcTo.
- To change either an elliptical or circular arc to a straight line, select the pencil tool, point to the arc's control point, then drag until it "snaps" into a straight line.

Changing the row type can alter a shape's width-height box and overwrite proportional or height-based formulas. For this reason, you may want to set LockCalcWH to TRUE in the Protection section before changing a row type.

Controlling how shapes flip and rotate

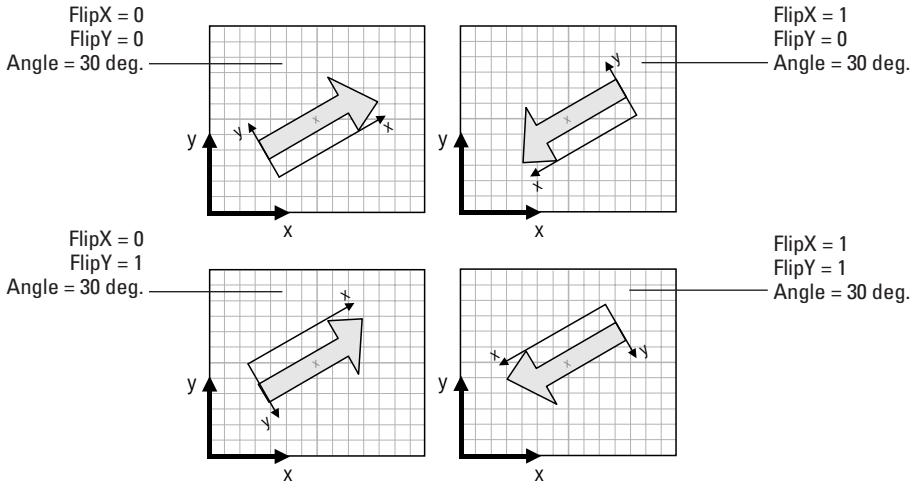
When you create a master, you need to anticipate how the user will flip or rotate the shape, and then design appropriate behavior. The Shape Transform section in the ShapeSheet window records a shape's orientation with respect to its parent. When a user flips or rotates a shape, the Shape Transform section reflects the actual transformation that occurs.

How flipping affects a shape

If you are designing shapes that users can flip, you need to be aware of the different behaviors that result depending on the method that was used. To flip a shape, users can:

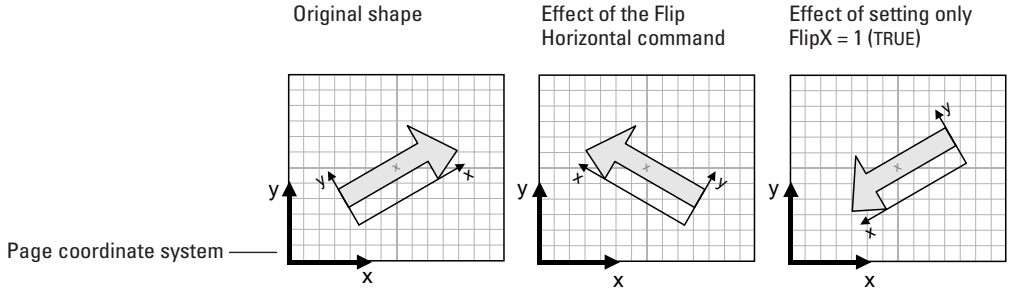
- Choose the Flip Vertical or Flip Horizontal commands from the toolbar or Shape menu.
- Choose the Size & Position command from the Shape menu, then use the Flip Vertical or Flip Horizontal checkboxes.
- Set the value of the FlipX or FlipY cell in the Shape Transform section.

When a shape is flipped, the value of its FlipX or FlipY cell changes to TRUE. The parent coordinates of the shape's origin change, but the location of the shape's pin doesn't change with respect to either its local or parent coordinates. In the following figure, the shape is rotated to show more clearly the interaction of the FlipX and FlipY cells.



Local coordinates of a rotated shape as FlipX and FlipY values are changed

Depending on which of the previous methods a user employs to flip a shape, two different shape transformations can result. Using the Size & Position command to flip a shape has the same effect as editing the values of the FlipX and Flip Y cells in the Shape Transform section. For example, setting the value of the FlipX cell to TRUE flips the shape horizontally by reversing the direction of the shape's local *x*-coordinate axis. However, when a user chooses the Flip Horizontal command from the toolbar or Shape menu, the shape appears to flip about a vertical line in the page coordinate system that passes through the shape's pin. The value of the FlipX cell is toggled between TRUE and FALSE, and the value of the Angle cell becomes *–angle*, a different shape transformation, as the following figure shows.



The Flip Horizontal command both flips and rotates the shape.

Using the Flip Vertical command on the toolbar or Shape menu has the effect of toggling the value of the FlipY cell and changing the value of the Angle cell to *–angle*.

How rotating affects a shape

To rotate a shape, a user can drag a shape handle with the rotation tool or use the Size & Position command on the Shape menu, which includes an editable Angle field. When a shape is rotated, the value in the shape's Angle cell describes the rotation of the local coordinate system with respect to the parent coordinate system. A shape rotates about its pin: The parent coordinates of a shape's origin change as the shape is rotated, but the location of the shape's pin doesn't change with respect to either its local or parent coordinates.

If page rotation is enabled, users can rotate the drawing page, which causes existing shapes and guides to appear to rotate as well. However, a shape's coordinates and Angle cell do not change regardless of the page's rotation. For details about rotating pages, see "Working with rotated pages" in Chapter 8, "Scaling, snapping, and aligning."

Designing shapes that flip and rotate

Will users flip and rotate your shapes? In some cases, you may want to prevent them from doing so, and in other cases you can design your shapes to accommodate these actions. You can prevent a shape from being rotated by guarding the value of its Angle cell with this formula:

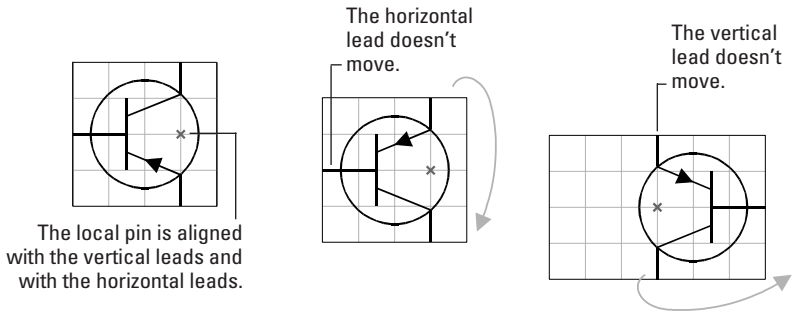
```
Angle     = GUARD(0 deg.)
```

The shape can still be flipped, but users will not be able to use any of the Visio tools to rotate it except by changing the value of the Angle cell. You can also prevent a shape from being rotated with the rotation tool by setting the LockRotate cell in the Protection section. Padlocks appear on the shape's rotation handles, giving users a visual clue that is not provided when you guard the value of the Angle cell. However, the lock doesn't prevent the shape from being rotated by means of the Flip Vertical and Flip Horizontal commands.

To prevent a shape from being flipped, guard the values of the FlipX and FlipY cells:

```
FlipX     = GUARD(0)
FlipY     = GUARD(0)
```

If you expect users to flip and rotate your shape, you can design the shape to work at different angles and orientations. For example, you can change the way a shape flips or rotates by moving its local pin. In the following figure, when a user flips the transistor symbol vertically, the horizontal lead stays in position. When the shape is flipped horizontally, the vertical lead stays in position. This behavior makes the transistor flip appropriately in electrical schematics with cascaded transistors.

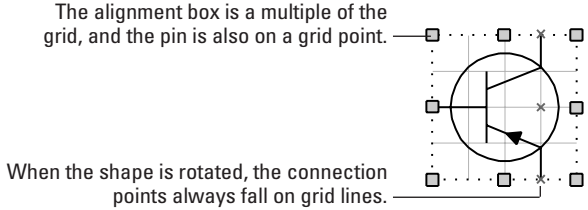


The transistor shape unflipped, flipped vertically, and then flipped horizontally

To change the location of a shape's pin, you can use the rotation tool to drag the pin to a new location. Doing this changes the values of PinX and PinY, but the LocPinX and LocPinY formulas also change to counteract the pin movement so that the shape doesn't jump on the page. You can also move the pin by changing only the values for the LocPinX and LocPinY cells, which changes the relationship between the local pin and the parent pin, so the shape also moves. For example, the transistor shape offsets the local pin with the following formulas:

```
LocPinX   = Width * 0.75
LocPinY   = Height * 0.5
```

Some shapes, such as the transistor symbol shown below, are commonly rotated by multiples of 90 degrees. You can create such a shape so that its alignment box coincides with the grid and its pin and any connection points lie on grid points. That way, the shape will snap into alignment more quickly when a user flips or rotates it. For details about working with the grid, see Chapter 8, "Scaling, snapping, and aligning."



A transistor symbol designed to rotate in multiples of 90 degrees

Grouping and merging shapes

When you need to draw shapes with complex geometry or control the behavior of multiple shapes, you can group or merge the shapes. The method you choose can affect performance for your users and affects how you work with the resulting shape's ShapeSheet interface. Depending on how you want the resulting shape to look and behave, choose one of these methods to join multiple shapes:

- Use the Group command (Shape menu, Grouping submenu) to combine several shapes or other groups into a new Visio shape whose components can still be edited and formatted individually.

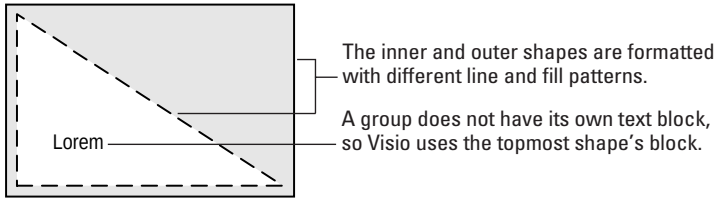
- Use an Operations command on the Shape menu (Combine, Union, and so on) to combine multiple shapes into a single new shape with multiple Geometry sections corresponding to the original component shapes.

For users, interaction with a group is similar to interaction with a merged shape that has multiple Geometry sections. With both groups and merged shapes, users can rotate and add text. However, there are significant differences between groups and merged shapes. When choosing between them here are some things to consider:

- Only with a merged shape can users edit vertices, unless they open the group in the group editing window.
- Only with a group can users apply different styles and text formats to the shapes that make up the group.
- A group can always be ungrouped to recover the individual shapes, unlike a merged shape. Moreover, merging a shape by using any of the commands on the Operations submenu will overwrite user-built formulas.
- A merged shape is represented by a single ShapeSheet interface and often responds faster to user actions than a group. In a group, each component shape retains its separate ShapeSheet interface, *plus* the group is represented by a ShapeSheet interface.

Grouping and ungrouping shapes

You should create a group when you want several shapes to move and size together, yet retain their individual formatting attributes. The Group command adds a ShapeSheet representation for the group and modifies formulas in the component shapes to refer to the group (specifically, it modifies formulas the Shape Transform or 1-D Endpoint sections), but it doesn't overwrite individual shape formatting.



A group composed of two shapes with different formatting attributes

Masters that are groups

To create a single master that is composed of several shapes, it is best to group or merge the shapes first. If you don't create the group or merged shape, Visio will group the shapes when a user drags the master into a drawing—an additional step that can increase the time required to create an instance of the master.

When you add a shape to a group, its parent coordinate system is no longer the page, but the group. When you ungroup shapes or remove the last shape from a group, the group is no longer the parent, and the group sheet is deleted. ShapeSheet formulas that refer to parent coordinates change when you group or ungroup the shape, and custom formulas that you define for a shape can be overwritten. For this reason, it's best to group shapes before adding connection points or defining custom formulas, which often must reference the group in order to work properly.

The following table shows the cells that are reset with new formulas when you group and ungroup shapes.

ShapeSheet changes when a shape is grouped or ungrouped

Section	Cell	What happens
Shape Transform	Width Height	After grouping, formulas reference the group to define the shape's size in proportion to group's size; for example, *Sheet12!Width * 0.5* (where Sheet12 is the group). After ungrouping, formulas reference the width and height of the new parent or are constant if the new parent is a page. Formulas protected with the GUARD function aren't affected.
	PinX PinY	Formulas base the pin coordinates on the group's or new parent's coordinate system. After grouping, formulas define the pin's location in proportion to the group width and height.
1-D Endpoints	BeginX, BeginY, EndX, EndY	Formulas base the coordinates of the begin and end points on the parent's coordinate system. After grouping, formulas define the endpoints' position in proportion to the group width and height.
Alignment	[all cells]	Formulas base the position of the alignment guide on the group's or new parent's coordinates.

When you group shapes that are connected to other shapes, Visio maintains the connections, unless a shape is connected to a guide that has—or as a result of the grouping will have—a different parent. If a shape is glued to a guide and you add the shape (but not the guide) to a group, Visio breaks the shape's connection to the guide. The reverse is also true: If you add a guide to a group, but don't also add shapes that are glued to that guide, Visio breaks the shapes' connections to that guide. If you include both the guide and the shapes that are glued to it, Visio maintains the connections.

Group coordinate system

A group's local coordinates are the ones shown on the rulers of the group editing window. To display this window, select the group, then choose Open Group from the Edit menu.

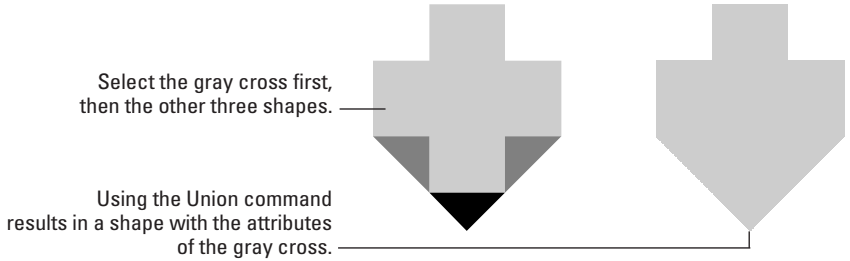
Viewing multiple Geometry sections

If a merged shape has multiple Geometry sections, it's not easy to tell by looking which part of the shape's geometry a Geometry section refers to. One method is to tile the ShapeSheet window and the drawing page window, and then click a Geometry cell. Visio highlights the corresponding vertex in the drawing page.

Merging shapes

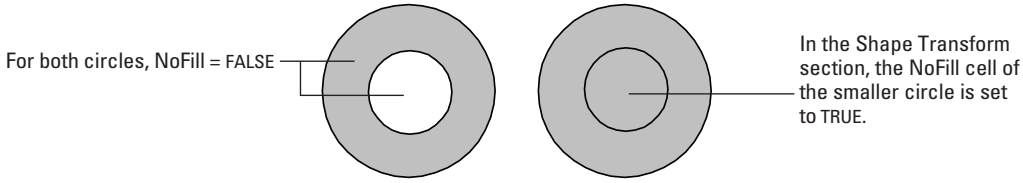
When you want to create a single shape with multiple Geometry sections, use the Union, Combine, Fragment, Intersect, Subtract, Join, or Trim commands on the Operations submenu of the Shape menu. Unlike the Group command, these commands merge the geometry of several shapes to create a single shape that can have more than one Geometry section for each path.

When you merge shapes, the original shapes, and any custom formulas in them, are not retained, and you cannot recover them by ungrouping as you can do with grouped shapes. A merged shape has only one text block and one set of formatting attributes, as the following figure shows. Otherwise, the shape behaves like any other single shape.



When you merge shapes with different formats, the resulting shape inherits the attributes of the first shape you select.

You can selectively control the geometry of merged shapes. For example, to create a doughnut shape, draw two concentric circles, then use the Combine command. The resulting shape has one Geometry section for the outer circle and one for the inner. If you apply a fill color to the shape, it fills as you would expect a doughnut to. But what if you wanted a filled circle with another filled circle inside? You can do this by setting the value of the NoFill cell in the Geometry section for the smaller circle to TRUE.

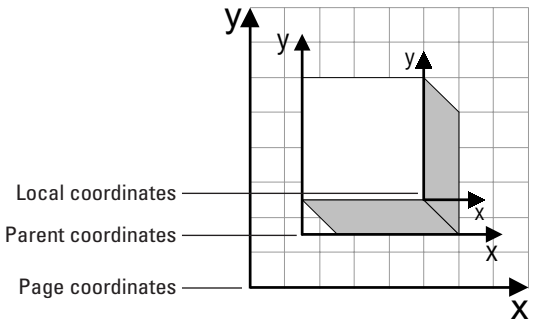


Two concentric circles joined using the Combine command, then filled with a color

Resizing shapes in a group

When you resize a group, you need to consider how the size and position of the component shapes should change within the group. As a group is resized, its component shapes are typically stretched and repositioned to maintain their proportions in the group coordinate system. However, some shapes represent objects with fixed physical dimensions. When the group changes size, you can define these shapes to change position only, but not change their size or proportions. You can set group attributes that govern how component shapes are resized and positioned.

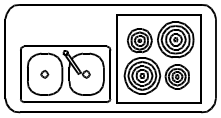
When you work with formulas in grouped shapes, you're working with local coordinates, parent coordinates, and page coordinates, as the following figure shows. Defining different resize behavior for a grouped shape can involve converting coordinates from one system to another.



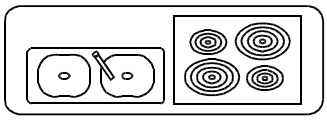
A shape in a group in the Visio coordinate system

Defining the resizing behavior of grouped shapes

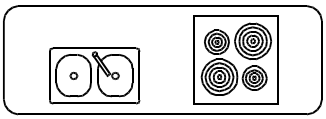
When a group represents a physical object, its resizing behavior should reflect the physical object's behavior in the real world. In some cases, that will mean when a group is resized, some component shapes will be resized and others will not. For example, in the following figure, the island group contains a countertop, range, and sink. The range and sink represent physical objects of industry-standard size that should not resize with the island. A countertop, however, can be constructed to any size and should resize with the island.



Original group



By default, component shapes resize when the group is resized.



If ResizeMode is 1, the sink and stove are only repositioned.

You can control a component shape's or group's resizing behavior with the ResizeMode cell in the Shape Transform section. To control a component shape in a group, set the value of ResizeMode for the component shape. For example, set ResizeMode to 1 for the sink, then group the sink with the countertop. The following table shows the resizing options you can use.

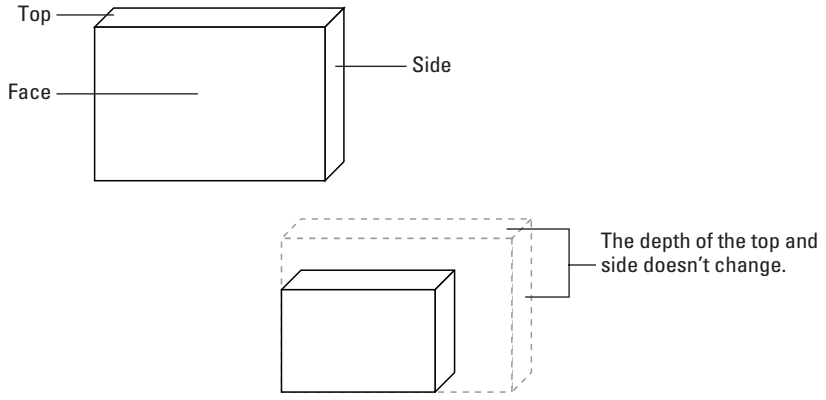
ResizeMode settings in Shape Transform section

Value	Description
0	Shape resizes according to the group's ResizeMode setting. Corresponds to Use Group's Setting in Behavior dialog box. (This is the default.)
1	Shape keeps its size when the group is stretched; only its location within the group changes. Corresponds to Reposition Only in Behavior dialog box.
2	Shape resizes proportionally when the group is stretched. Corresponds to Scale With Group in Behavior dialog box.

When you set a different resizing behavior, do it for the highest-level shape possible—for example, protect the stove rather than each burner. To keep users from accidentally resizing a shape in a group, set ResizeMode to 1, and also set LockWidth and LockHeight to 1 in the Protection section. If you set locks for a shape's width, height, or aspect ratio and then add the shape to a group, the shape's resizing behavior takes precedence over the locks. When a user opens the group in the group editing window, however, the shape's attributes will still be locked so that it can't be resized.

Resizing shapes in only one direction

When you want different shapes in a group to resize differently, you can customize the component shapes' resizing behavior. For example, the 3-D box shape in the following figure is a group made up of three shapes: one for the face of the box, one for the top, and one for the side, each of which resizes differently. When you resize the face, it stretches proportionately in width and height, but the top stretches only in width, and the side stretches only in height. This way, the shape maintains its 3-D look as it is stretched.



The top and side of the 3-D box stretch in only one direction when the box is resized.

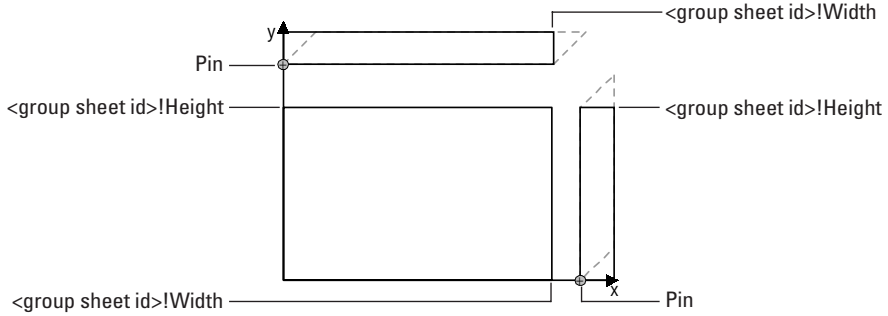
You use two key techniques to get this kind of resizing behavior in a group:

- You define the dimension of the component shape that doesn't resize as a constant value and the dimension of the shape that does resize in terms of the corresponding group dimension.
- You move the pin of the component shape to the origin in its local coordinate system (the lower-left corner of the local *x*- and *y*-axis). Then you define the shape's parent pin in terms of the group's width or height, so that the location of the component shape is always fixed with respect to the group. Otherwise, the component shape moves with respect to the parent coordinate system when the group is resized.

In the 3-D box shape, the top's height and the side's width are both constant values, because they shouldn't resize when the group is resized. The top's width is defined in terms of the group width, so the top can resize in the direction of width. Similarly, the side's height is defined in terms of the group height, so the side resizes in the direction of height.

The face shape defines the alignment box for the group, because its size and position determine the size and position of the top and side. The parent pin defines each component shape's position at the appropriate edge of the group alignment box. For the top, the *x*-coordinate of the parent pin is *0 in.*, and its *y*-coordinate is the same as the group's height. For the side, the *x*-coordinate of its parent pin is equal to the group's width, and its *y*-coordinate is *0 in.*

It's easiest to see the relationship between the component shapes' width and height and the group's width and height if you draw the shape without angled vertices, as in the following figure.



Exploded view of the 3-D box shape

Creating a 3-D box: an example

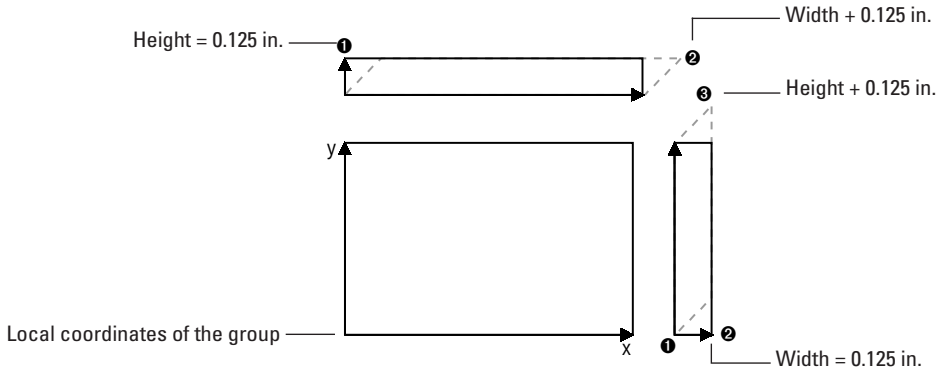
Using the techniques described in the previous section, you can create a shape with resizing behavior similar to the 3-D box. One shape defines the alignment box for the group, and the component shapes are fixed in position with relation to the alignment box. In addition, the component shapes resize in only one direction as the group is resized.

To draw the actual 3-D box group, you need to:

- Define the group's custom alignment box by drawing the face of the 3-D box first, grouping it, then locking the alignment box.

- Roughly draw the top and side shapes as simple rectangles, then add them to the group.
- Modify the vertices of the top and side to give them a 3-D look.
- Customize the Width, Height, and Pin formulas of the top and side shapes to control their resizing behavior.

As the following figure shows, vertex 2 of the top and vertex 3 of the side are skewed. The *y*-position of the top's vertex 2 is equal to that shape's height. The *x*-position of the side's vertex 3 is equal to that shape's width. Top height and side width are a constant value, 0.125 in. By adding this constant to the appropriate vertex formulas, we get a skewed shape.



Local coordinates for the component shapes of the 3-D box

To draw the 3-D box as a group:

1. Use the rectangle tool to draw the face, top, and side.

   Don't worry about drawing the top and side in perspective; just draw rectangles in approximately the right position.

2. Select just the face, and group it.
3. Select the group, choose Show ShapeSheet from the Window menu, and then set the formula for the LockCalcWH cell in the Protection section to 1.

   This step preserves the current alignment box. Otherwise, the group alignment box changes as you add the top and side shapes.

4. Select the group. From the Edit menu, choose Open Group to open the group editing window. Then select the top and the side and drag them into the group window to add them to the group.

5. In the group's ShapeSheet window, add the Scratch section, then enter the following formulas:

```
Scratch.X1          = 0.125 in.
Scratch.Y1          = 0.125 in.
```

   Scratch.X1 controls the width of the side. Scratch.Y1 controls the depth of the top.

6. Reference the constant values of the top and side shapes.

   To do this, in the group window, select the top shape, choose Show ShapeSheet, and add a Scratch section. Do the same for the side shape. In the top's and side's Scratch sections, enter these formulas:

```
Scratch.X1          = <group sheet id>!Scratch.X1
Scratch.Y1          = <group sheet id>!Scratch.Y1
```

   You must supply your group's ID in these formulas. For example, if the group's ID is Sheet.4, the formula for the X1 cell would be *Sheet.4!Scratch.X1*.

7. Define the skew for the vertices in the top and side shapes.

   To do this, you customize formulas in the Geometry section, as the following tables show.

Custom formulas in the Geometry section for the top

Row	X	Y
1 (Start)	= 0 in.	= 0 in.
2 (LineTo)	= Scratch.X1	= Height
3 (LineTo)	= Width + Scratch.X1	= Height
4 (LineTo)	= Width	= 0 in.
5 (LineTo)	= Geometry1.X1	= Geometry1.Y1

Enhancing group performance

To enhance performance, keep the total number of shapes in a group to a minimum, and use no more cell references between shapes than is necessary. For example, in step 6 of the procedure, you could refer to the group's Scratch section in the formulas for the shapes' vertices; however, performance is best if you keep references to cells in other shapes to a minimum. By adding a Scratch section to the top and side, you need only one reference to the group's row per section instead of several.

Another way to improve performance is to keep the number and level of nested groups to a minimum. Although you can add a group to a group, fewer groups generally result in faster performance. You can always use an Operations command to merge similar parts of a shape, then group the parts.

Custom formulas in the Geometry section for the side

Row	X	Y
1 (Start)	= 0 in.	= 0 in.
2 (LineTo)	= 0 in.	= Height
3 (LineTo)	= Width	= Height + Scratch.Y1
4 (LineTo)	= Width	= Scratch.Y1
5 (LineTo)	= Geometry1.X1	= Geometry1.Y1

8. Define the resizing behavior for the top and side shapes.

   To do this, you customize the Width, Height, and Pin formulas in the Shape Transform section. For the top, use these formulas:

```
Width     = <group sheet id>!Width
Height    = Scratch.Y1
PinX      = 0 in.
PinY      = <group sheet id>!Height
LocPinX   = GUARD(0 in.)
LocPinY   = GUARD(0 in.)
```

   In the Shape Transform section for the side, use these formulas:

```
Width     = Scratch.X1
Height    = <group sheet id>!Height
PinX      = <group sheet id>!Width
PinY      = GUARD(0 in.)
LocPinX   = GUARD(0 in.)
LocPinY   = 0 in.
```

More 3-D box formulas

One way to alter the 3-D box group is to use a control handle to set the depth of the perspective. To do this, you remove the width and height constants from Scratch.X1 and Scratch.Y1 of the group, and enter them instead in Controls.X1 and Controls.Y1 respectively of the Controls section. Then edit Scratch.X1 to reference Controls.X1, and edit Scratch.Y1 to reference Controls.Y1. As the control handle is dragged on the drawing page, the custom formulas in the top's and side's Width and Height cells will change accordingly. For details about control handles, see Chapter 4, "Enhancing shape behavior."

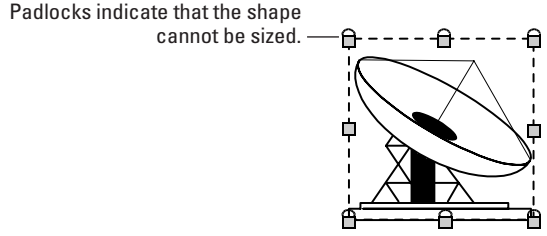
Protecting against unwanted changes

Most actions in the drawing page affect the ShapeSheet cells and so can affect the specialized behavior and custom formulas you have designed for a shape. You can set constraints on shape behavior, called *locks*, that prevent particular actions in the drawing page. You can also use specialized techniques to prevent your custom formulas from being overwritten by user or Visio actions.

For example, at Visio we created a grand piano shape in the home planning stencil. Pianos come in different sizes, but they are built only one way—the higher strings are always to the right as you face the keyboard. To protect this characteristic, the piano shape is locked against horizontal flipping. You can still rotate the piano—as you could if you were pushing it around the room—but you can't flip it.

Using locks to limit shape behavior

One of the simplest ways to protect your shapes is to set the lock cells in the Protection section of the ShapeSheet window. Setting locks prevents accidental changes to a shape. For example, if your shapes represent items with standard dimensions, such as building materials, you can lock their resizing handles, because users shouldn't be able to stretch the shapes in all directions. Setting some locks causes a padlock symbol to appear in place of some or all selection handles on the shape, indicating to the user that the feature cannot be edited.



Setting protection locks gives the user visual feedback.

To lock a feature, set the appropriate cell in the Protection section to a nonzero number. To unlock a feature, type 0 in the cell. You can also set some locks with the Protection command on the Format menu.

Setting locks doesn't affect the menu commands that are enabled and doesn't protect other ShapeSheet cells from change. For example, if you lock the width and height of a shape that is in a group and then scale the group, the width and height of the shape can change. Locking only prevents the user from scaling the shape with the mouse.

Protecting formulas

The only way to protect the formulas in individual ShapeSheet cells from change is to use the GUARD function. GUARD protects the entire formula in a cell; it cannot protect parts of formulas. Actions in the drawing window cannot overwrite ShapeSheet formulas protected by the GUARD function, except when a user applies a style. The GUARD function uses this syntax:

```
GUARD(expression)
```

When a shape is moved, resized, grouped, or ungrouped, Visio writes changes to ShapeSheet cells that can overwrite custom formulas. The cells most commonly affected by such actions are Width, Height, PinX, and PinY in the Shape Transform section. For example, to prevent a shape from being flipped, you can enter the formula:

```
FlipX = GUARD(0)
FlipY = GUARD(0)
```

The exception to the protection rule is the application of a style, which always overwrites formulas in the affected cells, even if they're protected by the GUARD function. Applying a style can overwrite the formulas of the Line Format, Fill Format, Text Block Format, Character Format, and Paragraph Format sections. For details, see "Protecting local shape formats" in Chapter 7, "Managing styles, formats, and colors."

A single action in the drawing window can affect several ShapeSheet cells. You must guard the formulas in each of these cells if you want to prevent unexpected changes to the shape. Of course, if a user deletes a ShapeSheet section, all the formulas in it, including guarded ones, will be deleted.

GUARD and protection locks

The GUARD function and protection locks protect your shapes in complementary ways. The GUARD function prevents formulas from changing, but it allows user actions. By contrast, setting locks in the Protection section prevents user actions without protecting cell formulas.

For example, if you use the formula *Width = GUARD(5 pica)*, users can stretch the selection handles, but the shape snaps back to its original width. If you use the formula *LockWidth = 1*, users cannot drag the shape's side selection handles.

For details about the syntax of the GUARD function, search online help for "guard function." For details about locks, search online help for "protection section."

Protecting the formatting of shapes in groups

When you locally format a group by choosing a command from the Format menu, you're also applying the format to all of the shapes in the group. Any local formatting you've applied to those shapes can be overwritten by the formatting applied to the group, and if you've used formulas to change the formatting of the component shapes, those formulas are also overwritten. To avoid this effect, you can:

- Protect specific formatting cells with the GUARD function.
- Lock a group against formatting changes.
- Selectively prohibit application of styles to some or all of the shapes in a master.

To protect specific ShapeSheet cells from changing when a user locally formats a shape, use the GUARD function. Although GUARD protects against local formatting, it can't protect a formula when a user applies a style to a group. The style overwrites the formulas in the formatting cell for the attributes included in the style definition. To apply a style to a group but keep your local formulas, use the Style command and check the Preserve Local Formatting option as you apply the style.

You can lock a group against formatting with the LockFormat cell in the Protection section of the group. This lock prevents a user from applying a style or local format to the group. When a user tries to do so, Visio displays a message indicating that the action isn't allowed.

When you lock a group against formatting, the shapes in the group can still be subselected and individually formatted unless you set the LockFormat cell for every shape in the group. You can also selectively lock against formatting in a group when you want to allow users to format some shapes but not others.

Styles versus local formatting

A shape that is formatted by applying a style may look no different from a shape that is locally formatted, but Visio treats them differently. To format a shape with a style, select the shape, then choose a style from the Text, Line, or Fill boxes on the toolbar. Or choose Style from the Format menu. To locally format a shape, select the shape, then choose any of the other formatting commands on the Format menu, such as Line, Fill, or Text, or click a button on the toolbar, such as Line Weight or Line Pattern.

When you apply a style to a shape that is locally formatted, the attributes defined in the style replace any corresponding local formatting. Locally formatted attributes that are not specified in the style are unaffected. For example, if a shape's line is locally formatted and you apply a text style that specifies only text formatting, the local formatting of the line remains intact, and only the text style changes.

For details, see Chapter 7, "Managing styles, formats, and colors."

4

Enhancing shape behavior

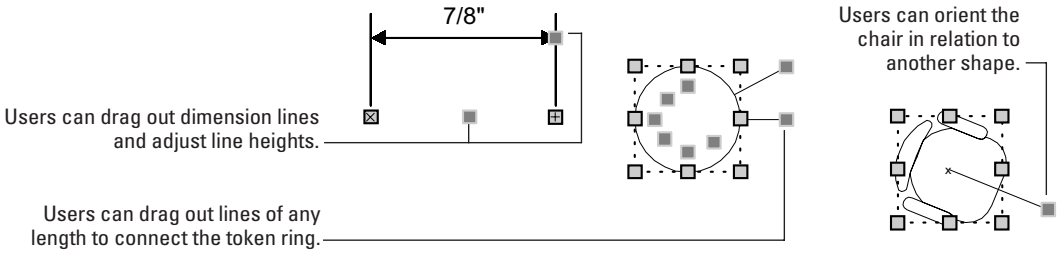
You can write any number of ShapeSheet formulas to control the appearance or position of a shape on a page, but there's more to shape behavior than geometry and location. A shape can provide information to users in the form of visual feedback, such as control handles with tooltips or custom commands on a shortcut menu. Moreover, users can associate information with a shape in the form of custom property data or layer assignments. These enhancements to shape behavior can make a shape better model the real-world object it represents.

This chapter describes techniques and formulas that you can use to add control handles, shortcut menu commands, custom properties, and layers to shapes and masters.

Topics in this chapter

Making shapes flexible with control handles

One way to control shape behavior, yet provide your users with greater flexibility, is to add *control handles* to a shape. Like selection handles, control handles appear as small green squares that users can select and move. A shape responds to changes in the control handle's position according to your formulas. The following figure shows different uses for control handles in a shape. The real strength of control handles is that they let you take advantage of user input when designing shape behavior.



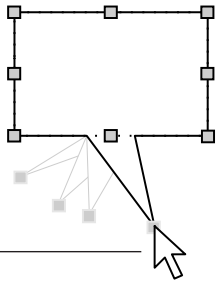
Visio masters with control handles

To add a control handle to a shape, you add a Controls section in the ShapeSheet window. The Controls section defines control handle attributes. After adding this section, you can write formulas that reference it to define the handle's behavior. In the Controls section you can also define a descriptive tooltip that appears when a user pauses the mouse over a control handle.

For ideas about using control handles, look at the shapes that come with Visio. Each shape with a control handle includes tooltip information describing the handle's behavior. For details about repositioning a text block with a control handle, see "Controlling the text block's position" in Chapter 6, "Designing text behavior."

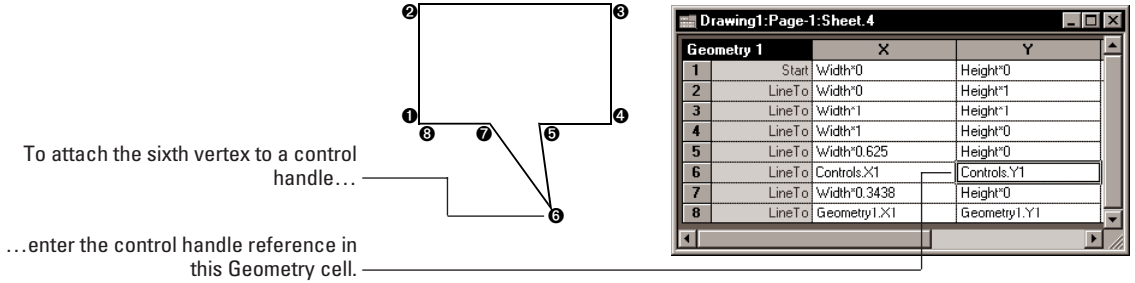
Adding and defining a control handle

You use the cells in the Controls section to define the control handle's location and behavior. For example, you can create a word balloon by adding a control handle, then associating it with a vertex of a rectangle, as the following figure shows.



Control handle defined for a vertex of a word balloon

When you add the Controls section to a shape, Visio creates a control handle with the coordinates *Width*0*, *Height*0*. You can move the control handle, but it doesn't do anything until you associate it with the shape's geometry in some way—typically with the vertex you want the handle to control. For example, the following figure shows how the vertex at the balloon's mouthpiece is associated with the control handle. The Geometry cell representing the appropriate vertex contains a cell reference to the Controls section.



The location of the control handle is defined by the formula in a Geometry cell.

To add a control handle to a shape:

1. Select the shape, then choose Show ShapeSheet from the Window menu.
2. If the Controls section is not already present, from the Insert menu, choose Section, and then check Controls. Click OK.

   Visio adds a Controls section with one row to the shape, and adds the control handle to the shape on the drawing page.
3. Put a cell reference to the handle (the Controls.X*n* and Controls.Y*n* cells) in the cell representing the vertex that the handle controls.

   In general, you enter a formula that refers to the *x*-coordinate of a control handle in an X cell of the Geometry section and a formula that refers to its *y*-coordinate in a Y cell. For example, for the word balloon you would enter:

   ```
   Geometry1.X6        = Controls.X1
   Geometry1.Y6        = Controls.Y1
   ```
4. In the X Behavior and Y Behavior cells, enter a constant from 0 to 9 to determine how the control handle resizes with the shape.

   For details, see "Setting a control handle's behavior" opposite. For a list of constants, search online help for "controls section."
5. In the X Dynamics and Y Dynamics cell, enter a formula to describe the origin of the control handle's anchor point.

   For details, see "Setting a control handle's anchor point" later in this section. For example, for the word balloon you could enter:

   ```
   X Dynamics          = Width/2
   Y Dynamics          = Height/2
   ```
6. In the Tips cell, enter a string for the control handle tooltip.

   Visio encloses the string in quotation marks. For example, for the word balloon you could enter:

   ```
   Tip     = "Reposition mouthpiece"
   ```

TIP As you drag a control handle, the status bar displays the handle's local coordinates.

Drawing the word balloon

You can draw the word ballon as a 2-D shape with an alignment box around the rectangle part only. To do this, use the line tool to draw the rectangle and mouthpiece, but draw the mouthpiece inverted—that is, pointing into the rectangle instead of out of it. This way, the alignment box encompasses the rectangle. To keep it that way as the mouthpiece moves, open the ShapeSheet window and in the Protections section, set LockW/H to TRUE (1). Now add the Controls section and formulas to associate the control handle with a vertex.

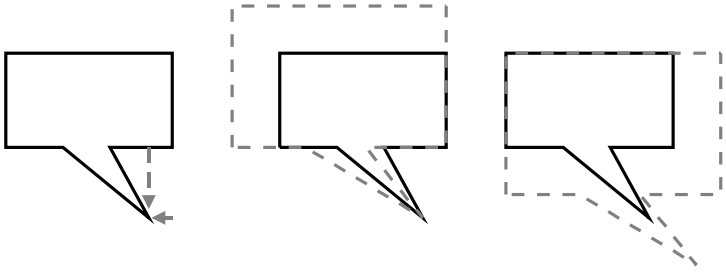
Setting a control handle's behavior

When a user stretches a shape that has a control handle, you can specify how the handle behaves when the shape is stretched: whether the control handle moves in proportion to the shape or stays in the same place relative to the shape. The values of the X and Y Behavior cells in the Controls section define a control handle's position and behavior, as the following table shows. The value of the X Behavior cell is independent of the value of the Y Behavior cell.

Settings for the X Behavior and Y Behavior cells

Value	Control handle behavior when shape is stretched
0 or 5	Moves in proportion with the shape when the shape is stretched. If 0, handle is visible; if 5, it is not visible.
1 or 6	Moves in proportion with the shape, but cannot be moved horizontally (X Behavior) or vertically (Y Behavior). If 1, handle is visible; if 6, it is not visible.
2 or 7	Offsets a constant distance from the shape's left side (X Behavior) or bottom (Y Behavior). If 2, handle is visible; if 7, it is not visible.
3 or 8	Offsets a constant distance from the center of the shape. If 3, handle is visible; if 8, it is not visible.
4 or 9	Offsets a constant distance from the shape's right side (X Behavior) or top (Y Behavior). If 4, handle is visible; if 9, it is not visible.

For example, the following figure shows a word balloon with a control handle whose X Behavior value is 4 and Y Behavior value is 2.



The control handle is offset a constant distance from the shape's right and bottom.

If the shape is stretched using the handles on the left or top, the control handle stays anchored…

…or if stretched using the bottom or right handles, the control handle moves to retain the offset.

Related control handle settings

To change display properties for control handles, you can set these values in the Miscellaneous section of the ShapeSheet window:

- Set the NoCtrlHandles cell to TRUE (1) to prevent control handles from appearing when a user selects a shape.
- Set the UpdateAlignBox cell to TRUE (1) to force Visio to recalculate a shape's alignment box whenever a user moves the control handle.

When a user stretches the word balloon, the mouthpiece stays anchored to the same point if the shape is stretched vertically and maintains its offset from the right side if the shape is stretched horizontally.

Setting a control handle's anchor point

When a user moves a control handle, a black line appears and stretches as the user drags the handle. This "rubber band" originates at the control handle's *anchor point*. The rubber band is a visual aid to help users determine where the control handle is being moved and what will happen to the shape as a result. You can set the anchor point at any position in relation to the shape using the X Dynamics and Y Dynamics cells.

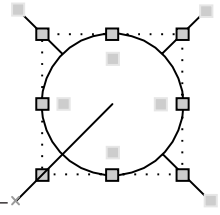
For example, to set a control handle's anchor point at the bottom of a shape, enter this formula:

```
Y Dynamics     = Height * 0
```

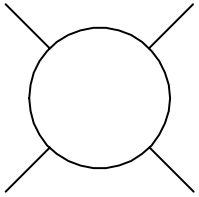
The location of the anchor point does not affect how the shape appears on the page, but only how the rubber band appears as the user moves the control handle.

To set a control handle's anchor point outside a shape, describe the anchor point with respect to the shape's width and height.

To help users work with your shape, set the anchor point so its position reflects its function, as the following figure shows.



As you move a control handle, the rubber band shows the relationship of the anchor point to the control handle's location.



A Token Ring shape has multiple control handles anchored to the same point.

Defining shortcut menu commands

When a user right-clicks a shape on the drawing page, a shortcut menu appears that includes commands that apply to the selection. You can define commands that appear on a shape's shortcut menu and on the Actions submenu of the Visio Shape menu. You use the Actions section in the ShapeSheet window to specify a new command name and define its action for a shape or page.

For example, you could define a menu command in the Actions section called Run Program that evaluates this formula:

```
Action    = CALLTHIS("my_prog")
```

When a user right-clicks your shape, the Run Program command appears on the shortcut menu. If the command is chosen, Visio evaluates the formula. In this case, Visio launches MY_PROG.EXE.

NOTE Action cells, like Event cells, are evaluated only when the action happens, not when you enter the formula.

To define an Action command for a shape or page:

1. Select a shape, and then choose Show ShapeSheet from the Window menu.

   Or with nothing selected, choose Show ShapeSheet from the Window menu to display the page's sheet.

2. If the Actions section is not already present, from the Insert menu, choose Section. In the Insert Section dialog box, check Actions, and then click OK.

3. In the Action cell, enter the formula that you want to be evaluated when the user chooses the Action command.

4. In the Menu cell, enter a command name as you want it to appear on the shortcut menu.

5. *Optional.* In the Prompt cell, type a descriptive prompt that appears in the status bar when the Action command is chosen on the menu.

To test the new command, right-click the shape or page to display its shortcut menu, and then choose the Action command you defined.

You can customize Visio menus and commands in other ways using Automation. For details, see Chapter 16, "Customizing the Visio user interface."

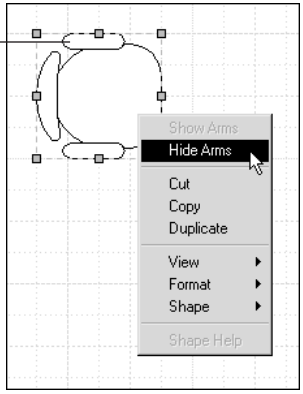
The Action command

To quickly define common actions for a shape or page, you can use the Action command on the ShapeSheet window's Edit menu. The command is dimmed unless you have inserted an Actions section and selected one of its cells. When you enter a value for the Menu and Prompt options in the dialog box, Visio updates the corresponding cells of the Actions section. When you choose an Action, such as Go To Page, Visio enters the appropriate formula in the Action cell.

Controlling shape geometry: an example

You can use shortcut menu commands to control shape geometry, so that users can choose a command to change the shape's appearance. For example, you can create a single shape that represents two states: on or off, open or closed, engaged or disengaged. To do this, you create a shape with multiple Geometry sections, called a *multishape*. In the Actions section of the multishape, you can define shortcut menu commands that control the visibility of the Geometry section that represents one state. To demonstrate, we'll create an office chair with arms that can be shown or hidden as the following figure shows.

Choosing the command displays the geometry of one of the multishape's component shapes.



You can define shortcut commands that appear when you right-click the multishape.

Handy SETF uses

You can use the SETF function in an Event or Action cell to toggle the value of another cell between two options or to increment values in another cell. Because the formula in an Event or Action cell is evaluated only when the event occurs, you can write a self-referential formula using the SETF function that doesn't cause a loop. For example, to toggle the value of cellA depending on the value of cellB, use the following syntax in an Event or Actions cell:

```
SETF("cellA",
IF(cellB=0,1,0))
```

To increment the value of *cell* by one, use this syntax:

```
SETF("cell", cell+1)
```

For details about the syntax of the SETF function, search online help for "SETF function."

To create a multishape, you use the Combine command to create one shape with multiple Geometry sections in its ShapeSheet interface. Each section defines the path of one of the original shapes. In the Actions section, you use the SETF function to set the value of a user-defined cell to true or false, then reference this value in the appropriate Geometry sections to indicate if the chair arms are visible.

To combine shapes into a multishape:

1. Create the shapes you want to use in your multishape.

   For example, to create a chair, draw a rectangle or oval for the seat, one for the chair back, and one for each arm.

2. Select the chair shapes. From the Shape menu, choose Operations, then Combine.

   Visio creates a single shape that contains one Geometry section for each original shape. The Geometry sections are numbered in the order in which you selected the shapes.

3. From the Window menu, choose Show ShapeSheet.
4. From the Insert menu, choose Sections. Check User-Defined Cells and Actions, and then click OK.
5. Type a name for the user-defined cell, such as User.State, and then enter the value *1*.

   The initial value is 1, or true, so that the chair arms are visible.
6. Select the first row in the Actions section, then choose Row from the Insert menu so that there are two rows altogether in the Actions section.
7. To create the command names and corresponding actions, enter these formulas:

```
Action[1]   = SETF("User.State",1)
Menu[1]     = "Show Arms"
Action[2]   = SETF("User.State",0)
Menu[2]     = "Hide Arms"
```

8. In the two Geometry sections that correspond to the arms of the chair, enter this formula:

```
Geometryn.NoShow    = NOT(User.State)
```

   For example, if the arms of the chair correspond to the Geometry3 and Geometry4 sections, you would enter:

```
Geometry3.NoShow    = NOT(User.State)
Geometry4.NoShow    = NOT(User.State)
```

How the formulas work. The Action cell formula sets the value of User.State to 1 (TRUE) when the Show Arms command is chosen or 0 (FALSE) when the Hide Arms command is chosen. The Menu cell defines these command names.

To hide and show paths, you enter formulas in the NoShow cell of the appropriate Geometry section that refer to the value of the User.State cell. The NoShow cell (which is labeled B1 but referred to as Geometry*n*.NoShow) controls whether the path defined by that Geometry section is shown or hidden. In this case, the arms are both shown or both hidden, so the same formula is used in the NoShow cells of the two corresponding Geometry sections.

Action menu names

To control the position of your Action command in the shortcut menu, you can use a prefix before the name you type in the Menu cell. To display your command at the bottom of the shortcut menu, use this syntax:

```
= "%Menu item"
```

To display a divider bar above the command, use this syntax:

```
= "_Menu item"
```

To create a keyboard shortcut for the command, place an ampersand (&) before the desired shortcut letter, as follows:

```
= "&Menu item"
```

The NOT function returns 1 if an expression is false and 0 if true. When a user chooses Hide Arms, User.State is set to 0 (false). The NOT function then returns 1 so that the value of the NoShow cell is true and the path for the corresponding component is hidden.

Dimming the command on the menu. To refine the shape, you can dim the command that no longer applies to the shape. For example, when a user chooses the Hide Arms command, the arms are hidden so that only the Show Arms command needs to be highlighted on the menu. To do this for the chair shape, you use a logical expression in the Disabled cell of the Actions section, such as the following:

```
Disabled[1]    = User.State=1
Disabled[2]    = User.State=0
```

The result of any formula in the Disabled cell is evaluated as either true or false. When the Show Arms command is chosen, the value of User.State is 1, so the expression in the Disabled cell evaluates to true, and the Show Arms command is dimmed. If the Hide Arms command is chosen, User.State is 0, so the expression in Disabled[2] cell is true.

Checking commands on the shortcut menu

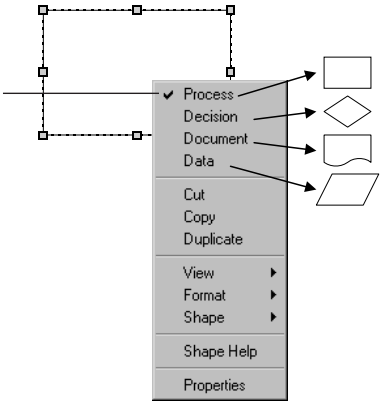
When you define several shortcut menu commands for a shape, you can show which one has been applied to the shape by placing a check mark beside it on the menu. To do this, you set the Checked cell of the Actions section to TRUE. You can use a logical expression to check and uncheck the command under the appropriate conditions, such as:

```
Checked      = User.State=1
```

In this case, when the value of the user-defined cell User.State is true (1), the Checked cell evaluates the formula as true, and Visio places a check mark beside the command name.

The Flowchart Shapes master in Visio uses the Checked cell to tell users which command is in effect. Like the chair shape in the previous example, the Flowchart Shapes master has multiple Geometry sections that are shown or hidden depending on which command on the shortcut menu has been chosen.

A user chooses to display the Process, Decision, Document, or Data shape geometry by checking the appropriate command on the shortcut menu.



You can place a check mark beside the commands you define in the Actions section.

For details, see the Flowchart Shapes master on the DVS SmartShapes template in the \DVS\SHAPE SOLUTIONS folder.

Hiding and showing commands

Instead of dimming a shortcut menu command that is not available, you can create mutually exclusive commands that appear only under the appropriate conditions. For example, in the chair shape described earlier, you could display only one command on the menu at a time: If the arms are visible, the command on the shortcut menu is Hide Arms. If the arms are hidden, the command is Show Arms.

In the chair shape, the User.State cell holds the value of the state of the shape: 1 (true, or show arms) or 0 (false, or hide arms). To create a command that changes on the shortcut menu, you need only one row in the Actions section. You write two logical expressions: one in the Menu cell to determine which command to display based on the value of User.State, and another in the Action cell to toggle the value of User.State:

```
Action    = SETF("User.State",NOT (User.State))
Menu      = IF(User.State,"Hide Arms","Show Arms")
```

The SETF function writes the result of the IF statement to the User.State cell. If the value of User.State is true (1), SETF sets it to 0; otherwise, the value of User.State is false (0), so SETF sets it to 1.

The formulas to show or hide shape geometry remain the same as in the chair example earlier:

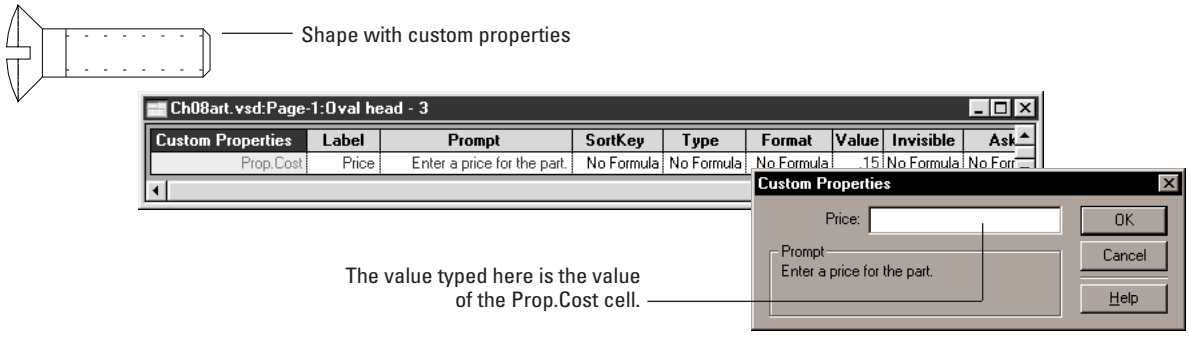
```
NoShow    = NOT(User.State)
```

Whenever a shape has only two states or attributes that represent an either/or situation, you can toggle the command names. If the shape has more than two states or menu commands, users will find it less confusing if you use the Checked or the Disabled cell to indicate which commands are available.

Working with custom properties

You can define a *custom property* to store string, numeric, Boolean, date or time, duration, currency, fixed list, or varable list data with a shape, group, master, or page. A custom property is stored as a ShapeSheet cell whose name and contents you define. You can view and modify custom properties through the Visio menu commands, as well as create reports from the information or refer to the values in other ShapeSheet cells to modify the shape's behavior.

For example, you can use custom properties to update an inventory control list. You can create a stencil containing masters that represent the parts in inventory. For each master, you can define the custom properties Name, Cost Per Unit, and Quantity. You can set the value of these properties when you create the shapes, or you can allow the shapes' users to enter the name, cost, and quantity for a given part, even prompting users to do so.



Custom properties for an inventory control list

To transfer the custom property values to an external source, such as an inventory control list, you could write an intermediary program to get the data and write it to a spreadsheet or database. You can also set the value of a custom property from an external source. For details about using Automation to retrieve and set custom property values, see Chapter 14, "Working with drawings and shapes."

Defining custom properties

You can define custom properties for a single shape or a page by adding the Custom Properties section to its ShapeSheet interface. If you're editing a stencil, a more efficient method is to define custom properties for the stencil's masters so that their instances also contain the properties. With the Custom Properties Editor, you can define custom properties for a master on a local or standalone stencil.

To add custom properties to a shape or page:

1. Select the shape you want, or click an empty portion of the drawing page, then choose Show ShapeSheet from the Window menu.
2. If the Custom Properties section is not already present, from the Insert menu, choose Sections. In the Insert Sections dialog box, check Custom Properties, and then click OK.
3. In the Custom Properties section, select the Row label Prop.Row_1, which appears in red text. In the formula bar, type a descriptive name.

   For example, type *Unit_Cost* to create the custom property Prop.Unit_Cost. The name that appears in the Row label is the cell name for the Value cell in that row. Use this name (for example, *Prop.Unit_Cost*) in cell references.
4. In the Label cell, type the label that appears to users in the Custom Properties dialog box for this property.

   For example, type *Cost Per Unit*. In the ShapeSheet cell, Visio automatically encloses the string in quotation marks.
5. In the Prompt cell, type descriptive or instructional text that appears to users in the Custom Properties dialog box when the property is selected.

   For example, type *Enter the cost per unit for the part*. Visio automatically encloses the string in quotation marks.

Data types in earlier versions of Visio

Versions of Visio products earlier than 4.5 do not support the custom property types 5 (date and time), 6 (duration), and 7 (currency). If you open a drawing that includes these data types in an earlier version of a Visio product, the custom property values are evaluated as strings.

6. In the Type and Format cells, enter a data type and format for your custom property's value.

   For details, see the following table.

7. *Optional.* Set the Invisible cell to a nonzero number (TRUE) to prevent the custom property from appearing in the Custom Properties dialog box. Set the Ask cell to a nonzero number (TRUE) to display the Custom Properties dialog box whenever an instance of the shape is created.

   To see the custom property you have defined, select the shape in the drawing page or cancel all selections, if you want to view the page's custom properties. Then choose Custom Properties from the Shape menu.

Custom property data types and formats

Type	ShapeSheet formula	Description
String	`Type = 0` `Format = "<Picture>"`	This is the default. Use a valid format picture* in the Format cell to format strings as number-unit pairs, dates, times, etc.
Fixed list	`Type = 1` `Format = "Item 1;Item 2"`	Displays the list items in a drop-down list box in the Custom Properties dialog box. Specify the list items in the Format cell. Users can select only one item from the list.
Number	`Type = 2` `Format = "<picture>"`	Use a format picture* in the Format cell to specify units of measure and other number formats.
Boolean	`Type = 3`	Displays FALSE and TRUE as items users can select from a drop-down combo box in the Custom Properties dialog box.
Variable list	`Type = 4` `Format = "Item 1;Item 2"`	Displays the list items in a drop-down combo box in the Custom Properties dialog box. Specify the list items in the Format cell. Users can select a list item or enter a new item.
Date or time	`Type = 5` `Format = "<picture>"`	Use a format picture* in the Format cell to specify days, months, years, hours, minutes, seconds, or other date formats; time formats; or combination date and time formats.
Duration	`Type = 6` `Format = "<picture>"`	Use a format picture* in the Format cell to specify elapsed time in hours, days, weeks, months, or other duration formats.
Currency	`Type = 7` `Format = "<picture>"`	Use a format picture* in the Format cell to specify currency formats.

* For example, Format = "##/10 UU" formats the number 10.92 cm as "10 9/10 CENTIMETERS" (specifying the use of "10" in the denominator and the uppercase, long form of the units). For details about valid format pictures, see "Formatting strings and text output" in Chapter 6,"Designing text behavior." Or search online Help for "format function."

To add custom properties to a master:

1. Open a standalone stencil as Original, or display the local stencil by choosing Show Master Shapes from the Window menu.
2. From the Shape menu, choose Custom Properties.

   Visio asks if you want to add properties to the masters. Click Yes.
3. Follow the instructions onscreen to identify the masters you want to edit, then click Next to add the custom properties.

   The editor adds the Custom Properties section to the selected master and sets the value of the Label, Prompt, and other cells based on your selections.

   To see the custom property you have defined, select the master in the stencil, then choose Custom Properties from the Shape menu.

Adding a Properties command to a shortcut menu

You can add a Properties command to a shape's shortcut menu that displays the Visio Custom Properties dialog box. To do this, you write a formula that uses the DOCMD function, which takes as its argument a Visio command number constant. These constants are declared in the Visio type library and prefixed with visCmd.

To add a Properties command that displays the Custom Properties dialog box, enter the following in a shape's Actions section:

```
Action    = DOCMD(1312)
Menu      = "Properties"
```

For best results, use DOCMD only for nondestructive commands that do not require that particular data be available to succeed. When the user chooses a command from a menu, Visio provides certain safeguards to make sure the command is appropriate. For example, a user cannot save a document if no document is open, or quit Visio with unsaved changes without first being prompted to save his or her work. The DOCMD function provides no such safeguards—Visio executes the command with whatever data happens to be in memory in the locations referenced by the command, whether that data is appropriate or not. This can produce indeterminate results at best, and damage files or cause data to be lost.

Protecting custom properties

You cannot use the GUARD function to protect the value in the Value cell of the Custom Properties section. You can, however, create a custom property that is not displayed in the Custom Properties dialog box. To hide a custom property, set its Invisible cell to TRUE (1). You can still work with the custom property in the ShapeSheet window or from a program.

Using custom properties with a database

After you have defined custom properties for a shape, you can link the data to a database. The Database Wizard can automate this process for you. It links the values of ShapeSheet cells in the Custom Properties section to a database created in an application compliant with the Open Database Connectivity (ODBC) standard. If you revise the database, you can refresh the values in the ShapeSheet cells to reflect the revisions. If you change the value of the custom properties in Visio, you can update the database to reflect the changes.

When it links a shape to a database, the Database Wizard adds the Custom Properties section and the User-Defined Cells section to the shape. The latter section is where the wizard stores information about the primary key for the database, the database fields that are linked to ShapeSheet cells, and the last valid data retrieved from the database.

To run this wizard, from the Tools menu, choose Macro, then Database, then Database Wizard. For details about options, click the More Info button in the wizard. Or search online help for "database wizard."

Assigning shapes and masters to layer

You can assign shapes and masters to *layers*, which are named categories that help users organize shapes in a drawing. When a user drags a master onto the drawing page, the instance inherits the layer information from the master. If the instance is assigned to a layer that doesn't exist on the page, Visio automatically creates the layer. When shapes include layer assignments, users can highlight shapes by layer in different colors while working in a drawing, print shapes by layer, and hide all shapes on a layer.

For example, if you're creating shapes for an office plan stencil, you can assign the wall, door, and window shapes to one layer, electrical outlet shapes to another layer, and furniture shapes to a third layer. You can also lock layers to prevent the shapes on the locked layer from being edited. A shape can belong to more than one layer.

Stencils and templates with layers

If you intend to save a document as a new stencil or template, you can save space by removing all layers from the drawing page that have no shapes assigned to them. To do this, check Remove Unreferenced Layers in the Layer Properties dialog box.

Layers belong to pages; every page has a list of layers associated with the page. Shapes appear on layers and can belong to more than one layer. By hiding or locking different layers on a page, you can control which shapes are visible or can be edited. You use the Layer Properties command on the View menu to control the behavior of each layer and also the behavior of the shapes associated with that layer.

To assign a master to a layer:

1. Double-click the master in the stencil to open it in the master drawing window.

   For details about editing stencils, see "Opening a stencil" in Chapter 2, "Tools for creating solutions."

2. Select the master, and then from the Format menu, choose Layer.

3. In the Layer dialog box, select the layer you want the shape to belong to, and then click OK.

   To assign a shape to more than one layer, press the Ctrl key to select multiple layers.

   If no layers currently exist, Visio displays the New Layer dialog box. Type a name for the new layer, and then click OK to display the Layer dialog box, where the new layer appears in the list.

When you assign a shape or a master to a layer, the Layer Membership section of the ShapeSheet window shows the layer assignment as an index to the list of layers on the page. The layer index corresponds to an entry in the Layer dialog box, which corresponds to a row in the Layer Membership section. The first layer created is layer 0, the second is layer 1, and so forth. Deleting layer 0 does not increment the other layer numbers. Layers are listed in alphabetical order in the dialog box. If a shape belongs to more than one layer, each layer index is separated by a semicolon.

For details about working with layers on the drawing page, search online help for "layering shapes" and "layers on pages."

Layers, pages, and backgrounds

In other graphics programs, the term layers often refers to the stacking order of objects on the page. In Visio products, layers organize shapes into categories called layers, while pages and backgrounds can be used like transparencies to stack the shapes that appear on them. For example, you can draw a title block on a background so that it appears on all foreground pages to which the background is assigned.

Because layers belong to the page, each page in a drawing file can have a different set of layers. Both foreground and background pages can have layers to organize the shapes that appear on them. A shape can belong to any and all layers on the current page. If you copy a shape with layer membership data to another page in the drawing, the associated layers are added to the destination page, if it doesn't already have them.

5

Making shapes connect: 1-D shapes and glue

Should a shape behave like a box or a line? When you're designing a shape, that's one of the first questions you need to ask. A shape that behaves like a box—that is, a 2-D shape—can be stretched vertically or horizontally. A shape that behaves like a line—a 1-D shape—can be stretched and rotated in one operation. You can use 1-D shapes to join other shapes together, and in this capacity they are often called *connectors*. The attribute of a connector that causes it to stay joined to another shape is called *glue*.

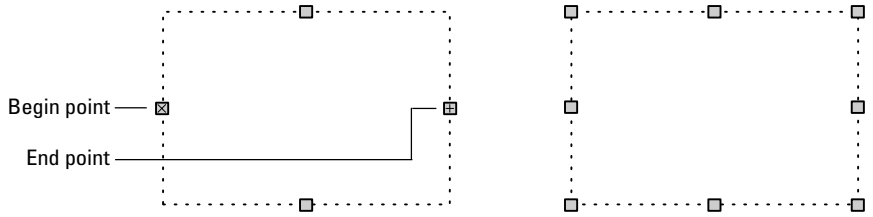
This chapter describes how to create different types of 1-D shapes. It explains the differences between 1-D and 2-D shapes, and how to work with the glue that holds them together.

Topics in this chapter

Understanding 1-D and 2-D shapes

When you want a shape for which the size or length of the line is less important than the connection it represents, create a 1-D shape. Because 1-D shapes are often used to connect other shapes, they are called connectors. For example, in a flowchart, circuit diagram, or mechanical illustration, 1-D shapes often connect other components. However, not all 1-D shapes are connectors. Some function as lines, such as callouts or dimension lines, or are simply easier to work with as 1-D shapes, such as a wedge of a pie chart.

Most shapes when you first draw them are 2-D. Their width-height boxes have eight handles for resizing. When you draw a single arc or line, however, the result is a 1-D shape that has handles for begin and end points and for height adjustment. Not only do 1-D and 2-D shapes look different, they act differently on the drawing page.



Position of selection handles on a 1-D shape

Position of selection handles on a 2-D shape

A shape that looks like a box can act like a line, because you can convert a 2-D shape to 1-D and vice versa. Converting a shape in this way dramatically changes its ShapeSheet interface.

When a user drags a 1-D shape onto the drawing page, its alignment box appears as a straight line, rather than as an outline of a box as for 2-D shapes. This can make the shape easier for users to align, as with a 1-D wall shape in a space plan.

Two of the 1-D shape's handles have a special purpose. The starting vertex of a 1-D shape is its *begin point*, and the handle that represents the end of the line formed by the shape is the *end point*.

Drawing 1-D shapes

The easiest way to create a 1-D shape is often to draw the shape roughly as a 2-D shape, convert it to 1-D, and then adjust the vertices and enter smart ShapeSheet formulas. You can save time and effort when you initially draw the shape by orienting it horizontally—that is, dragging left to right or right to left in the direction you want the line to go. Visio places 1-D endpoints on the left and right side of the shape you draw, so a horizontally drawn shape will be closer to what you want after it is converted to 1-D.

You can glue the begin or end point of a 1-D shape to a guide, guide point, connection point, shape vertex, or handle. If you glue one end, the other end stays anchored on the page, and the 1-D shape stretches as the glued end moves with the shape it is glued to.

Converting 1-D and 2-D shapes

A key difference between a 1-D and 2-D shape is whether its ShapeSheet interface includes the 1-D Endpoints section. If so, the shape is 1-D. Converting a 2-D shape to 1-D adds this section and default formulas. Converting a 1-D shape to 2-D removes it, regardless of any protection (including GUARD functions) that you may have set.

When you convert a 2-D shape to 1-D, the Alignment section is deleted, and the formulas in the Shape Transform section's Width, Angle, PinX, and PinY cells are replaced with default 1-D formulas. Converting a shape does not remove its connection points, but its connections to other shapes or guides are broken.

To convert a shape from 1-D to 2-D or 2-D to 1-D:

1. Select the shape.
2. From the Format menu, choose Behavior.
3. Under Interaction Style, select Line (1-Dimensional) to specify a 1-D shape. Select Box (2-Dimensional) to specify a 2-D shape.
4. Click OK.

   Visio modifies the shape and adjusts the alignment box according to the behavior chosen.

Functions that convert coordinates

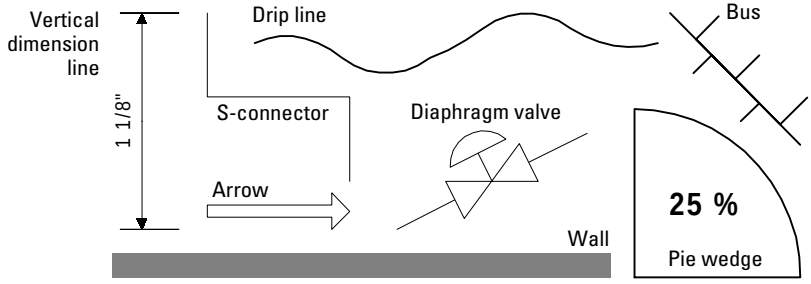
You can use the LOC and PAR functions with the PNT function to convert the coordinates of a point on one shape into the local coordinates of another shape, or into the coordinate system of the shape's parent.

When both of those points are in local coordinates, use the LOC function. For example, use the LOC function to express the vertex in one shape in terms of the vertex in another shape. For details, see the Block shape in the SOLUTIONS stencil in \VISIO\DVS\SHAPE SOLUTIONS. It uses the LOC function to express the position of another shape in local coordinates. The result of this conversion is used to position a control handle.

Use the PAR function when you want to express the endpoint or pin location of one shape, which is shown in parent coordinates, in terms of a vertex in another shape, which is shown in that shape's local coordinate system. For example, if you glue the endpoint of a connector to a connection point on a shape, Visio expresses the coordinates of the glued endpoint in terms of the connection point using the PAR function. To see this, display the ShapeSheet window for a glued connector and look at the formula in the EndX and EndY cells.

1-D shape gallery

The 1-D shapes shown in the following figure have custom formulas that create smart behavior. For example, the ShapeSheet formulas for the S-connector keep the connector right side up. As its endpoints are moved, the shape resizes in a way that keeps it upright by stretching its horizontal or vertical segments. Smart formulas for the valve shape give it height-based resizing behavior. As a user moves an endpoint, the line stretches, but the middle details remain the same size. If a user increases the shape's height, the middle details resize proportionately, but the line does not change.



Examples of 1-D shapes

The arrow shape shown in the figure can also be a 2-D shape. When should an arrow act like a line and when should it behave like a box? If the arrow is intended to be used in an up-down, left-right manner only, then making it 2-D can make horizontal and vertical positioning easier. In addition, 2-D shapes cannot be rotated without the use of the rotation tool, whereas it is very easy to change the angle of a 1-D shape accidentally by nudging one of its endpoints. However, to allow the arrow to connect other shapes through the Visio user interface (as opposed to an add-on), it must be 1-D.

Not all 1-D shapes require special formulas to be useful. Because a 1-D shape looks like a line as it is being dragged, it can be faster to position in a drawing. Consider using 1-D shapes whenever you want to create masters that your users will align precisely in a drawing. For example, a text callout or annotation shape is easier to position accurately if users can see where the line will point.

Creating routable and other 1-D connectors

If you are designing solutions for connected diagrams, you must decide whether to use the connector tools built into Visio or design your own connectors. Connected diagrams include everything from flowcharts to piping diagrams. When your users create such diagrams, you can rely on the automatic routing behavior that Visio includes with the dynamic connector tool (on the Standard toolbar) and the Connect Shapes command (on the Tools menu and Shape toolbar), provided that you design your shapes to work with them. In some cases, however, you may want to create your own 1-D connectors instead.

The dynamic connector tool and Connect Shapes command create *routable* connectors between *placeable* shapes. A routable connector is a 1-D shape that draws a path around other shapes rather than crossing over them on the drawing page. A placeable shape is a 2-D shape that the routable connector works with. Whether shapes are placeable and routable in a drawing determines how Visio reacts when changes occur, such as when shapes are added, deleted, resized and repositioned. In response to such changes, Visio automatically repositions shapes that are placeable and reroutes shapes that are routable.

Routable connectors can save users lots of time when they revise complex connected diagrams. In some cases, however, you may want a connector with more predictable behavior—that is, one that does not automatically reroute. For example, if your drawing type requires connecting lines that always form a 90-degree angle or that connect shapes with an arc or spline, you can create your own 1-D connector that is not routable.

Dynamic versus universal connector

By default, the shapes created by using the connector tool, the Connect Shapes command, or the Dynamic Connector shape are instances of the dynamic connector. The dynamic connector is routable, but users can manually edit its position as well. When the dynamic connector crosses over another routable connector on a drawing page, it can "jump" over the other line with a U-shaped bump.

The dynamic connector supercedes the functionality provided in previous versions of Visio products by the universal connector. You can still open drawings that contain the universal connector; the shape is not converted to a dynamic connector. To use the universal connector shape, open the Connectors stencil in the Visio Extras folder.

You can open a drawing containing the dynamic connector in version 4.0. Visio treats the connector like a 1-D shape and ignores its automatic routing capabilities. If you edit the connector in version 4.0, then open the drawing in version 5.0, the dynamic connector works as expected—its routing behavior is retained.

Creating routable connectors

You can create a routable connector from any 1-D line by setting its ObjType cell in the Miscellaneous section to 2. To control the path taken by a routable connector, you set its *behavior,* which corresponds to the value of the ObjBehavior cell of the Miscellaneous section. By default, the value of this cell is No Formula, which evaluates to 0, meaning the connector uses the behavior set for the page.

The Lay Out Shapes command (on the Tools menu) provides two routing behaviors for the page, Right Angle and Flowchart, but you can specify other behaviors. For example, you can create a routable connector that always creates a tree diagram in north-south orientation by setting its ObjBehavior cell to 7. To set this behavior as the page default, change the User.visRoutingStyle cell to 7 in the page's

User-Defined Cells section. For details about other settings for the ObjBehavior cell, search online help for "miscellaneous section."

The following table provides valid values for the ObjType cell.

Values for a shape's ObjType cell

Value	Meaning	Automation constant
0	Visio decides based on the drawing context	visLOFlagsVisDecides
1	Placeable	visLOFlagsPlaceable
2	Routable	visLOFlagsRoutable
4	Not placeable, not routable	visLOFlagsDont

When you create a new 2-D shape, by default Visio sets its ObjType to No Formula, which evalutes to 0, meaning that Visio will determine whether the shape can be placeable depending on its context. For example, if you draw a simple rectangle, the value of its ObjType cell is 0 by default. If you then use the Connect Shapes command or the dynamic connector tool to connect the rectangle to another shape, Visio decides that the rectangle can be placeable, and sets the rectangle's ObjType cell to 1 (placeable). You can also create a placeable 1-D shape (via the ShapeSheet window only), which can be useful when the shape is not a connector and will be used in a drawing with automatic layout. Setting a 2-D shape to routable, however, has no effect on its behavior.

When you create a template for a diagram that uses routable connectors and placeable shapes, you can customize the default values that Visio uses to route and place shapes. By specifying values with the Lay Out Shapes command on the Tools menu, you define the default values for the page. To set these defaults without affecting the layout on the current drawing page, be sure to check Set Layout Properties Only in the Layout Shapes dialog box. Users can edit shapes on the page to override the page settings; however, when users create or add placeable shapes, by default, the settings for the page will be used.

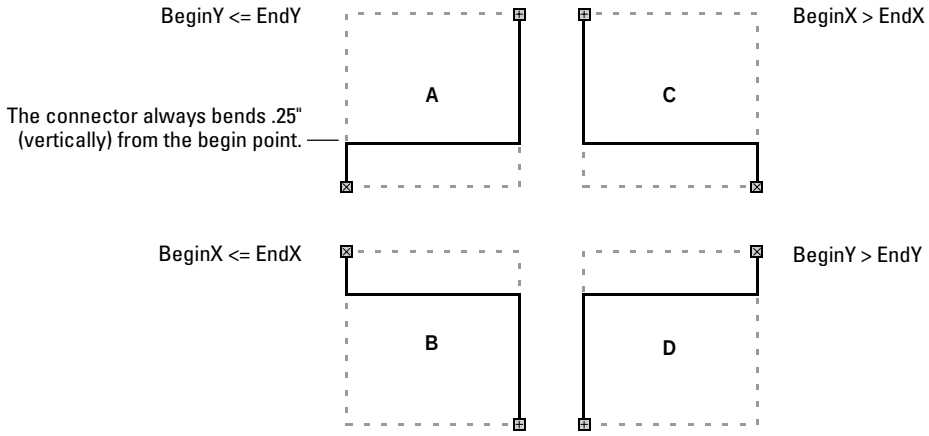
For details about creating diagrams that use routable connectors and placeable shapes, see "Working with connections" in *Using Visio Products* or search online help for "automatic layout."

Nonroutable 1-D connectors

If you are creating shapes that you specifically do not want to work with routable connectors, set their ObjType cell to 4. Connectors can glue to connection points on the shape, but in a diagram that contains placeable shapes and routable connectors, the nonplaceable shape will be ignored—that is, routing lines will behave as if the shape does not exist.

Creating an angled connector: an example

When your solution calls for a connector with behavior that you can control programmatically, you can create one that does not automatically route. With ShapeSheet formulas, you can control how a connector extends from its begin point to its end point. For example, the following figure shows an angled connector with two right-angle bends, which is useful for creating hierarchical diagrams such as organization charts. The custom formulas for this connector are included in this section as a demonstration of the type of formulas you need to control 1-D shapes. You can find other 1-D connectors in the Connectors stencil in the Visio Extras folder.



The four different ways an angled connector can bend when a user moves it.

With its two bends in the middle, the angled connector has two vertices that require custom formulas. To calculate the coordinates of the first vertex after the begin point, remember that its *x*-coordinate is the same as that of the begin point. The *y*-coordinate is 0.25 in. if the shape is drawn from the bottom up. If it is drawn from the top down, its *y*-coordinate is calculated as:

```
= Height – 0.25 inches
```

The *x*-coordinate for the next vertex is the same as the *x*-coordinate for the last LineTo row, which specifies the shape's end point and so is always Width or 0. Its *y*-coordinate is the same as the preceding vertex.

Adding a connector to a shape

You can use the SmartShape Wizard to add a connector to an existing shape. The result is a group consisting of the original shape and a line with a control handle that can glue to another shape. For example, you could add a connector to a text-only shape, then glue the line to a part you want to annotate in a drawing.

By using the wizard, you can create built-in connectors with a variety of connecting behavior, such as top to bottom, side to side, and so on. To start the wizard, from theTools menu, choose Macro, choose Visio Extras, and then choose SmartShape Wizard.

To create an angled connector:

1. Select the line tool and draw a straight, 1-D line from left to right.
2. From the Window menu, choose Show ShapeSheet.
3. Type the following formulas in the Shape Transform section:

```
Width     = GUARD(ABS(EndX - BeginX))
Height    = GUARD(ABS(EndY - BeginY))
Angle     = GUARD(0 deg.)
```

4. From the Insert menu, choose Section, and then check User-Defined Cells.
5. In the User-Defined Cells section, type a name for the cell, such as *yOffset,* and then type *0.25 in.* in the Value cell.
6. Select the last row in the Geometry section, and then from the Insert menu, choose Row After. Repeat to add two rows total.

   Each row corresponds to a vertex of the shape.
7. Type the formulas shown in the following table.

Custom formulas in the Geometry section

Row	X	Y
1 Start	`= IF(BeginX <= EndX,0,Width)`	`= IF(BeginY <= EndY,0,Height)`
2 LineTo	`= Geometry1.X1`	`= IF(BeginY <= EndY, User.yOffset, Height - User.yOffset)`
3 LineTo	`= Geometry1.X4`	`= Geometry1.Y2`
4 LineTo	`= IF(BeginX <= EndX,Width,0)`	`= IF(BeginY <= EndY,Height,0)`

8. In the Protection section, set the LockHeight cell and LockVtxEdit cell to 1.

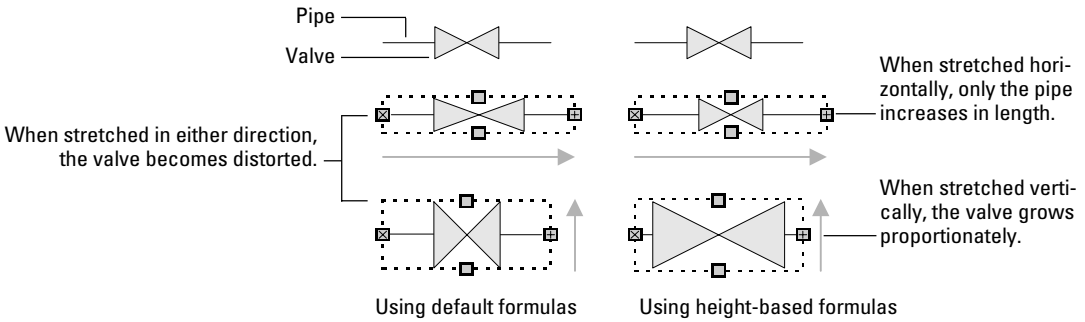
   Setting LockVtxEdit protects the geometry formulas by preventing users from editing the shape vertices. Setting LockHeight protects the height formula and removes the top and bottom handles, which aren't needed for a connector.

Adding a control handle

If you want users to be able to change the position of the bend in the angled connector by moving a control handle, you can link User.YOffset to a control handle and lock the handle's *x*-position so that it moves only in the *y* direction. For an example of this behavior, see the Bottom To Top Variable connector in the Flowchart - Basic stencil. For details about control handles, see Chapter 4, "Enhancing shape behavior."

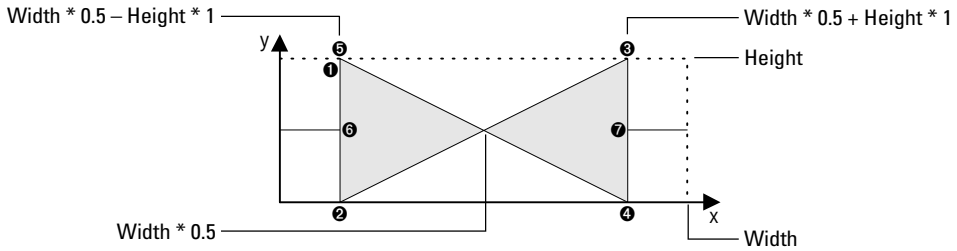
Creating a height-based 1-D shape: another example

Some shapes, such as the 1-D pipe and valve shape in the following figure, can stretch between two points to connect other shapes. You can create this type of 1-D shape as a single shape with multiple geometry components that have different resizing behaviors. In a 1-D shape, the endpoints control the shape's width. In the pipe and valve shape, when a user drags an endpoint, only the line component stretches. When a user drags a top or bottom handle, only the valve component resizes, and it does so in a way that maintains its aspect ratio.



You use a height-based formula to define the width of the valve component in terms of the shape's height. To create this connector as a single shape, you draw two line segments at either end of a valve shape, then use the Combine command.

To maintain the valve's proportions when the shape is resized, define the *x*-coordinates of the valve's vertices in relation to the center and height of the shape, as the following figure shows. Doing so also serves to keep the valve centered in the width-height box. This approach requires that you set formulas in the X cell of the Geometry section.



Height-based formula for a 1-D shape with multiple geometry components

To control the valve portion of the shape, open the ShapeSheet window for the combined shape and type the formulas shown below in the Geometry1 section. To create this shape in a way that ensures your Geometry rows match the figures and tables shown here, see the following sidebar, "Creating a 1-D shape by combining multiple shapes."

Custom formulas in the Geometry1 section

Row	X	Y
❶ Start	`= Width*0.5 – Height*1`	`= Height*1`
❷ LineTo	`= Geometry1.X1`	`= Height*0`
❸ LineTo	`= Width*0.5 + Height*1`	`= Height*1`
❹ LineTo	`= Geometry1.X3`	`= Height*0`
❺ LineTo	`= Geometry1.X1`	`= Geometry1.Y1`

To control the point where the left pipe segment meets the valve (vertex 6 in the preceding figure), type this formula:

```
Geometry2.X2          = Geometry1.X1
```

To control the point where the right pipe segment meets the valve (vertex 7 in the preceding figure), type this formula:

```
Geometry3.X1          = Geometry1.X3
```

Creating a 1-D shape by combining multiple shapes

When you draw 1-D shapes such as the pipe and valve shape, you often draw several shapes, and then either group or combine them. Using the Combine command results in a more efficient shape. If the component shapes all have the same formatting, you don't need to group them, which adds a group sheet. However, you need to make sure that the endpoints of the resulting 1-D shape are in the right place.

Visio always places the begin point on the left end of a 1-D shape and the end point on the right. If you draw a shape top to bottom, and then convert it to 1-D, the endpoints may not be where you want them. So draw the component parts from left to right. You also need to select them in the right order before using the Combine command so that the resulting shape looks the way you expect.

For example, to create the pipe and valve shape, select the line tool and draw a straight line from left to right to form the left segment of the pipe. Use the line tool to draw the valve, starting at point 1 as shown in the preceding figure. Use the line tool to draw the right segment of the pipe from left to right.

Now select the shapes in this order: first the valve, then the left line, and then the right line. From the Shape menu, choose Operations, and then choose Combine. This results in a single shape with three Geometry sections numbered in the same order as the table in this section. You shouldn't add any custom formulas to the component shapes before you combine them, because the Combine command removes them anyway. To make the resulting shape 1-D, from the Format menu, choose Behavior. Check Line (1-Dimensional), then click OK.

Controlling how shapes connect

The behavior that allows part of a shape to stay connected to another shape is called *glue*. You can specify the part of a shape to which another shape can be glued by defining a *connection point*. You can glue the endpoints of a 1-D shape to a guide, guide point, shape vertex, or selection handle. Visio automatically creates a connection point when you glue a 1-D shape to a 2-D shape's vertices or handles. When an endpoint of a 1-D shape is glued to a 2-D shape, you can move the 2-D shape and the glued endpoint stays attached, stretching the 1-D shape as the unglued endpoint stays anchored.

You can define different types of glue behavior. When the endpoint of a 1-D shape remains fixed to a particular connection point, it is said to use *static* glue. If the 1-D shape's endpoint "walks" from connection point to connection point to improve the visibility of the connection as the other shape moves, it is said to use *dynamic* glue. This is how routable connectors glue placeable shapes. You can think of dynamic glue as shape-to-shape glue: It connects two shapes between the shortest route, simplifying a drawing. Static glue is point-to-point glue: The connection is always between the same two points, no matter how the shapes move.

Although 1-D shapes are usually used to connect 2-D shapes, in some cases you can glue 2-D shapes to other shapes. When working with 2-D shapes, you can glue:

- An entire side of a shape to a guide or a guide point.
- An edge of the alignment box to a guide.
- A selection handle to a guide point.
- A control handle to a connection point.

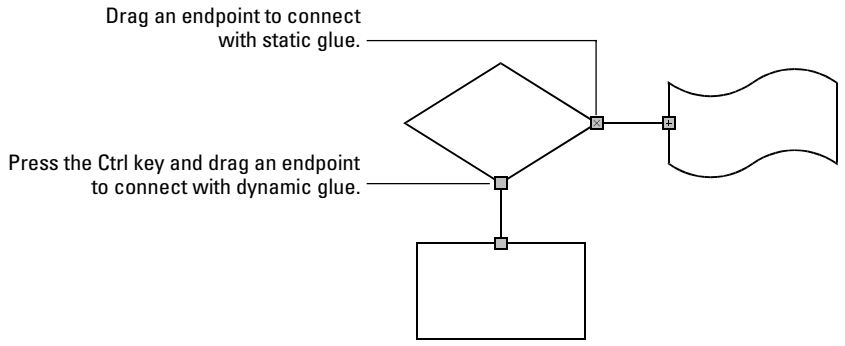
Defining a shape's glue behavior

You can choose the type of glue behavior a 1-D connector shape uses, static or dynamic. The default behavior for a shape you draw or any shape that is not placeable or routable is static glue.

Setting a template's glue behavior

When you are designing a template, you can choose which glue behaviors to enable in your template's drawing pages. From the Tools menu, choose Snap & Glue to specify the parts of a shape to which another shape can be glued on that page. Most Visio templates enable users to glue 1-D shapes only to guides, guide points, and connection points. In the Snap & Glue dialog box, you can check the Shape Handles and Shape Vertices options so that users can glue 1-D shapes to these points as well. Visio automatically creates a connection point at the glued handle or vertex.

Although you can set the default gluing behavior for your template's drawing pages, your users can change that default by choosing other options in the Snap & Glue dialog box.



A connector defined to use dynamic glue can create a connection with static or dynamic glue.

When you create a connector, you can set it to use dynamic glue so that its endpoints can move from one connection point to another as a user moves the shapes the connector is glued to. Visio redraws the connector so it connects the shapes at their two closest connection points. However, users must press the Ctrl key as they drag the connector to any shape other than a placeable shape in order to activate the dynamic glue. If they don't press Ctrl, the connector uses static glue. If the connector's endpoint is glued with dynamic glue, its selection handle is solid red. If glued with static glue, the selection handle displays the default begin point ( ) or end point ( ) symbols in dark red.

To define glue for a connector:

1. Select the shape, then from the Window menu, choose Show ShapeSheet.
2. If the Glue Info section is not displayed, from the View menu, choose Sections. In the Sections dialog box, check Glue Info, and then click OK.
3. In the Glue Info section, type *0* in the GlueType cell to specify static glue, or type *3* to specify dynamic glue.

By default, dynamic glue connects via the shortest route between two connection points or midshape selection handles. You can set a preference so that a shape with dynamic glue walks to a side, top, or bottom connection point when the glued endpoint is moved. To do this, set the WalkPreference cell. For details about WalkPreference settings, search online help for "WalkPreference." Routable connectors ignore the setting of the WalkPreference cell; their routing behavior is controlled by the value of the ObjBehavior cell.

Dynamic glue formulas

When a user glues a 1-D connector with dynamic glue to another shape, Visio generates a formula that refers to the EventXFMod cell of the other shape. When that shape is changed, Visio recalculates any formula that refers to its EventXFMod cell, including the formula in the BegTrigger and EndTrigger cells. These two cells contain formulas generated for a 1-D shape by Visio when the 1-D shape is glued to other shapes. Other ShapeSheet formulas for the 1-D connector refer to the BegTrigger and EndTrigger cells and move the begin or end point of the connector or alter its shape as needed.

Adding connection points

When you design a shape, you indicate the location where it can be glued by adding connection points to it. As you create masters, consider which points users will most likely need to glue another shape to, and avoid creating additional points, because they can make a shape respond less efficiently. Visio automatically creates a connection point at the vertex or selection handle of a shape when a connector is glued at that position, so you must manually add connection points only when you need one in a nonstandard location.

You create a connection point using the connection point tool on the toolbar or by adding the Connection Points section in the ShapeSheet window.

To create a connection point:

1. Select the shape.
2. Select the connection point tool, hold down the Ctrl key, and then click where you want to add a connection point.

NOTE The connection point tool always adds a connection point to the *selected* shape. Before using this tool, you must always select the shape to which you want to add the connection point.

When you add a connection point, Visio adds the Connection Points section to the ShapeSheet window with a row describing the point's *x*- and *y*-local coordinates. By changing the formulas for a connection point's coordinates, you can control how the location of the connection point changes when a shape is resized.

You can also store data in extra cells of the Connection Points section. To do this, right-click a row, then choose Change Row Type. The unitless cells, A, B, C, and D, become available for you to use for scratch formulas associated with your connection points.

Connection point names. You can rename the Connections Points row to create a more meaningful reference for the value contained in the X or Y cell of the same row. The cell name you enter must be unique within the section. When you create a name for one cell in this section, Visio names all the cells in the section with the default name, Connections.Row_*n*. If no rows in the section are named, the name cell is blank.

For example, to rename the cell for the first row, type *Custom* in the formula bar to create the cell name Connections.Custom. Visio creates the name Connections.Row_2 for the cell in the second row. To refer to the X cell of the first row, use Connections.Custom.X or Connections.X1. To refer to the Y cell of the first row, use Connections.Custom.Y1 or Connections.Y1. To refer to the X cell of the second row, use Connections.Row_2.X or Connections.X2, and to refer to its Y cell, use Connections.Row_2.Y or Connections.Y2.

Connection points in a group. If your master is a group, all the connection points should be added to the group, because only the connection points on the topmost group are visible in the drawing window. Unless you expect users to ungroup your shape, you should delete any connection points in the member shapes; they simply take up space.

If you want a connection point to refer to a shape in the group, you can either calculate the appropriate location in the referring formula, or use a reference to another shape within the group to refer to that shape's geomery. If you refer to anothershape, you may need to adjust for that shape's local coordinate system.

Connection points on a routable connector. You can add connection points to any shape, including a routable connector. As you move the connector, the connection points maintain their position along the path of the connector with respect to the endpoints. For example, if you place a connection point in the middle of a straight connector, which is then routed around shapes, the connection point remains in the middle as measured along the path of the connector. Because of this behavior, it's easiest to precisely position connection points when you add them to a straight connector.

Using named connection points

Named connection point rows are not compatible with versions of Visio earlier than 5.0. When saving a Visio 5.0 drawing file to an older format, references to named connection point rows are converted to indexed references, and the row names are lost.

6

Designing text behavior

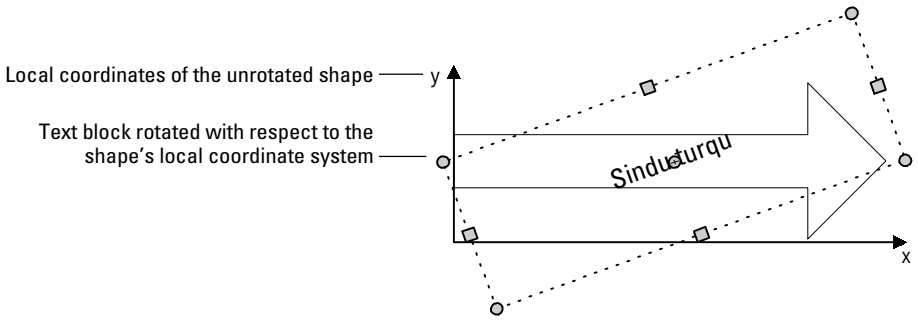
By default, users can add text to any Visio shape. When you design shapes, it's important to consider the position and appearance of the text block attached to a shape—even if it's a shape you think no one will ever add text to. Should the text rotate with the shape? Should the text resize with the shape? Should the shape be allowed to have text at all?

This chapter describes the mechanics of changing a shape's text block and provides guidelines for making a shape's text look good and behave in appropriate ways. It also describes the SmartShape Wizard, a useful shortcut to creating custom text behavior.

Topics in this chapter

About text in shapes

A shape's text has a coordinate system defined by an origin and axes relative to the shape's local coordinate system. This coordinate system is called the *text block*, and its size, location, and rotation relative to the shape are determined by the cells in the Text Transform section in the ShapeSheet window. When you create a shape, by default its text block is exactly the same size as the shape's width-height box: It has the same width and height and has zero rotation in relation to the shape. The default text block pin is in the center.



The local coordinate systems of a shape and its text block

Some of the questions that you should consider when you design the text behavior for a master are:

- Where should the text block be located? This question is especially important if the shape is a group.
- Should the text block be limited to a minimum or maximum size?
- Should the text use an opaque background?
- How should the text block grow as more text is added?
- What should happen to the text block when the shape is sized, rotated, or flipped?
- Should the master have multiple text blocks?
- Should the shape's text determine the size of the shape?
- Should the user be prevented from adding or changing the text in a shape?

The variety of possible text behaviors is endless, but in practice only a limited number prove useful. After all, the goal is to produce good-looking, readable text. Because smarter text behavior usually involves larger, slower, and more complex formulas, you must balance the text

block's sophistication with the expected uses for the shape. There is no single, simple answer, but consistency is important: Similar shapes should have similar text behavior.

Defining a text block in the ShapeSheet window

The Text Transform section defines a text block within the shape's local coordinate system, just as the Shape Transform section positions a shape within its group or page. To save space in the ShapeSheet window, Visio doesn't insert the Text Transform section unless you've changed a shape's text block attributes or used the text block tool to move the text block. However, you can add this section as you need it.

To add the Text Transform section to a shape:

1. Select the shape, then choose Show ShapeSheet from the Window menu.
2. From the Insert menu, choose Section.
3. In the Insert Section dialog box, check Text Transform, and then click OK.

   Visio adds a Text Transform section beneath the Shape Transform section with the values shown in the following table.

Text Transform section default values for a new shape

Cell	Formula	Cell	Formula
TxtPinX	`= Width * 0.5`	TxtLocPinX	`= TxtWidth * 0.5`
TxtPinY	`= Height * 0.5`	TxtLocPinY	`= TxtHeight * 0.5`
TxtWidth	`= Width * 1`	TxtAngle	`= 0 deg.`
TxtHeight	`= Height * 1`		

Text in a master

Should you include default text in a master to indicate where it can appear? This is a good idea in either of these situations:

- There are multiple text blocks in the master, and the user needs a cue to subselect different parts for text entry.
- The text is a standard phrase that is a natural part of the object (for example, the word "Stop" on a stop sign).

In other situations, it may be better not to include default text in the master:

- The user has to delete the text if it is not wanted.
- At whole page view, it is difficult to tell which shapes have the text you want and which still display the default placeholder text.
- It may require translation in other countries.

NOTE If the shape is a group, custom formulas that refer to the Width and Height cells may need to be modified to access the group's values rather than those of a component shape.

Viewing text attributes in the ShapeSheet window

When a user applies a text style to a shape or uses a formatting command on the Format menu, Visio updates the shape's ShapeSheet interface. Options in the Text dialog box correspond to cells in the Character, Paragraph, and Text Block Format sections. To view these sections, from the View menu, choose Sections, and then check Character Format, Paragraph Format, and Text Block Format.

The row numbers displayed in these sections reflect the number of characters that utilize the formatting defined in that row, as the following figure shows. For example, in a Character section with the row numbers 18, 16, and 1, the first 18 characters in the text block have the format described in the first row. The next 16 characters have the format described in the second row, and so on.

Character	Font	Color	Style	Case	Pos.	Size
1	4	0	1	0	0	8.0000 pt.
1	0	0	0	0	0	8.0000 pt.
1	65	0	3	0	0	10.0000 pt.
2	41	0	2	0	0	10.0000 pt.

The Character section for a shape with several different font formats

In general, if you write custom formulas in the Character and Paragraph sections, be sure you consider user actions that could overwrite your work. For example, if a user locally formats characters in a text block, Visio adds a new row to describe the formatting of those characters. When a user cuts text, Visio deletes the affected rows. If you want to write a custom formula in a cell of the Character section, copy the formula into that cell in each row of the section. That way, as Visio adds and deletes rows, at least one copy of the formula is likely to remain intact.

Shapes without text

If you create a shape that doesn't accommodate text, such as a very tiny shape without room for it, you can accept the default behavior and assume your users won't add text to the shape. Or you can add formulas to position and size the text block, in case users type in it. But perhaps the simplest thing to do is protect the shape against text entry by setting its LockTextEdit cell in the Protection section. You might also want to lock masters against text editing when they include specific text you don't want the user to accidentally or easily change. You can hide text altogether by setting the HideText cell to TRUE in the Miscellaneous section. You can still use the text tool to type in the shape: The text is visible as you edit, but it won't show in the shape when you're done editing.

Protecting text formulas

When you create custom text formulas for a shape, you can protect your formulas so that user actions cannot overwrite them. Many common user actions on the drawing page—formatting a font, setting margins, applying a text style—affect the values of the Text Transform, Text Block Format, Character, and Paragraph sections. If you write formulas to customize these text attributes, you can:

- Protect the formula using the GUARD function.
- Prevent users from making changes using a protection lock.

Use GUARD to protect formulas in cells that control the position or location of the text block. For example, protect formulas that customize text width and text height so that resizing a shape won't overwrite your formulas.

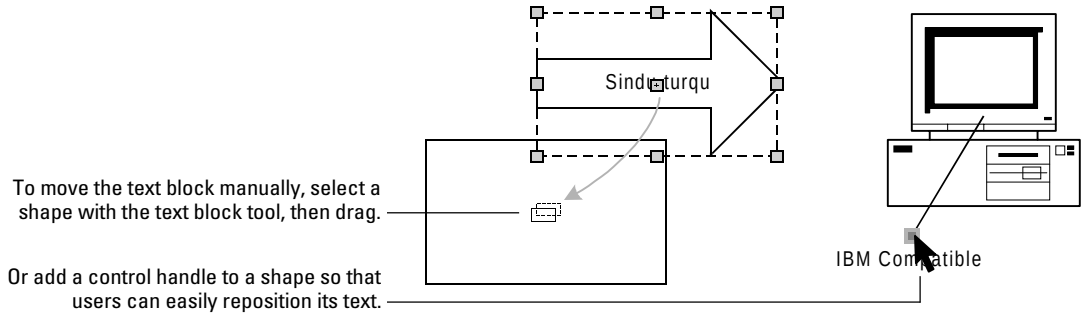
When you use GUARD to protect a formula in a cell that controls text formatting, users cannot locally format the text. However, when a user applies a text style, the formula is overwritten anyway. For this reason, avoid placing custom formulas in ShapeSheet cells of the Character, Paragraph, and Text Block Format sections, and instead use the Scratch and Text Transform sections. For details about how applying styles affects the ShapeSheet window, see Chapter 7, "Managing styles, formats, and colors."

Use a protection lock to prevent users from formatting a shape or typing in it altogether. Set the LockFormat cell in the Protection section to 1 to keep users from applying any formatting or styles. Set the LockTextEdit cell in the Protection section to 1 to keep users from typing in a shape (but to allow them to apply a text style). It's best to use LockTextEdit only in cases where entering text would damage the shape or cause it to behave in unexpected ways (as can happen with very small shapes).

Of course, the more protection you use, the less your users can modify a shape. What if a user simply wants to make a final adjustment to a shape's text and then print the document? You want to be able to give users enough flexibility to accomplish their tasks while preserving customized shape formulas.

Controlling the text block's position

When you are developing shapes, it often makes sense to move the text block to more easily accommodate readable text. For example, in many Visio shapes, the text block appears below the shape so that typing in it doesn't obscure the shape. You can easily move a shape's text block manually by using the text block tool. If you want to provide the users of your shapes with a more obvious method of adjusting text position, you can add a control handle that moves the text block.



Changing the position of a shape's text block

A quick way to add a control handle that moves the text block is to use the SmartShape Wizard, which provides options for setting the text block position, then defines a control handle for the block's pin.

To use the SmartShape Wizard to add a control handle:

1. Select a shape, then from the Tools menu, choose Macro, then Visio Extras, then SmartShape Wizard.
2. Under SmartShape Options, choose Customize Shape's Text, then click Change Option.
3. In the Text Position screen, choose the initial position you want for the text block.

   The Add Control Handle To Shape option is already checked.
4. Follow the instructions onscreen to finish the wizard.

Resizing the text block

You can quickly resize a shape's text block by dragging a text block selection handle. To display the selection handles for a text block, choose the text block tool, or choose the text tool and then press F2.

If you are designing shapes to distribute to other users, make sure that the text block is big enough for users to select easily. For example, the default size of a 1-D shape's text block may be too small for a user to easily adjust with the text block tool.

The wizard defines the control handle for the text pin by adding these formulas to the Text Transform section:

```
TxtPinX   = Controls.X1
TxtPinY   = Controls.Y1
```

A Controls section is added to the ShapeSheet window containing the formulas that define the control handle's position and its behavior as it or the shape moves or stretches. The formulas may vary depending on the position you set in the wizard for the text block. For example, if you centered the text block below the shape, the wizard adds these formulas to the Controls section:

```
X1          = Width * 0.5 + TxtWidth * 0
Y1          = Height * 0 + TxtHeight * -0.5
X Dynamics  = Width/2
Y Dynamics  = Height/2
X Behavior  = (Controls.X1 > Width/2) * 2 + 2
Y Behavior  = (Controls.Y1 > Height/2) * 2 + 2
```

The X1 and Y1 cells specify the position of the control handle in relation to the shape's local coordinates and the text block's coordinate system. The control handle appears in the center of the text block.

The X Behavior and Y Behavior cells define the behavior of the control handle after it is moved or after the shape is resized. The formulas in the X Dynamics and Y Dynamics cells set the position of the control handle's anchor point (the origin of the "rubber band") at the center of the shape.

For details about control handles, see "Making shapes flexible with control handles" in Chapter 4, "Enhancing shape behavior."

Resizing shapes with text

Your shapes should look good after users edit the text or resize the shape. You can control text behavior and appearance with formulas that correlate shape geometry and text. The quickest way to add common text formulas is to use the SmartShape Wizard. This section describes how to:

- Control the size of a shape's text block as a user types in it.
- Base a shape's size on either the amount or value of its text.
- Proportionately resize a shape's font as the shape is resized.

Controlling text block size

When you use the SmartShape Wizard to customize text, it assumes that you want to offset the text block from the shape, and so it adds formulas to control text block size. These formulas set the initial size of the text block and then ensure that the text block encompasses added text. To offset a text block, you need to know its boundaries. Without these formulas, a user can type outside the boundaries of the text block.

To control the text block size, the wizard uses the MAX function to define the maximum allowable size and the TEXTWIDTH and TEXTHEIGHT functions, which evaluate the width and height of the composed text in a shape. The wizard adds these formulas to the Text Transform section:

```
TxtWidth  = MAX(TEXTWIDTH(TheText), 8 * Char.Size)
TxtHeight = TEXTHEIGHT(TheText, TxtWidth)
```

The width of the text block is set to whichever value is greater: the longest text line terminated by a carriage return, or eight times the font size (which ensures that the text block is at least wide enough to hold a word or two.) If the text block contains text formatted with more than one font size, this formula returns the size of the first font used in the text block. The TxtHeight formula returns the height of the shape's composed text where no text line exceeds TxtWidth.

You can also set a text block to a minimum size by using the MIN function. For example, this formula ensures that when a shape is resized, its text block doesn't stretch wider than 4 inches or smaller than 0.5 inches:

```
TxtWidth  = MIN(4 in., MAX(0.5 in., Width))
```

Basing shape size on the amount of text

You can create a shape whose size depends on the amount of text it contains. If you want a shape that is just big enough to fit the text typed into it, such as a word balloon or text callout shape, use the TEXTWIDTH and TEXTHEIGHT functions as part of the formulas for the shape's width and height.

For example, the following formula in the Shape Transform section limits a shape's width to the length of the text lines it contains plus a small margin:

```
Width     = GUARD(TEXTWIDTH(theText) + 0.5 in.)
```

The function returns the width of all the text in the shape (theText). The shape's width is limited to that value plus 0.5 inch. The GUARD function prevents the user from stretching the shape's width with selection handles, which would cause new values to overwrite the formula in the Width cell. You can also set the LockWidth cell in the Protection section to prevent users from stretching the shape.

TEXTWIDTH and TEXTHEIGHT

The TEXTWIDTH and TEXTHEIGHT functions cause Visio to recompose the shape's text with each keystroke. To minimize the impact on performance, include a minimum-size test in your formula so the shape grows only after the text reaches a given width or height. Beyond that width or height, Visio still must recompose the text with each keystroke.

For example, you can create a 2-inch by 1-inch box that grows in height to accommodate text. To offset potential performance problems, the box doesn't resize until the text height reaches 1 inch. To create this behavior, add these formulas to the Shape Transform section:

```
Height   = GUARD(MAX(1 in.,
TEXTHEIGHT(TheText,Width)))
Width    = 2 in.
```

Basing shape size on text value

You can create a shape whose size is controlled by the value of the text it contains. For example, in a bar chart, you can ensure that the size of a bar depends on the value it represents. With the EVALTEXT function, you can create simple charting or other shapes into which users type a value that determines the shape's size. To associate a shape's width with its text value, put the following formula in the Shape Transform section:

```
Width     = GUARD(EVALTEXT(TheText))
```

The EVALTEXT function evaluates the text in the shape as if it were a formula and returns the result. For example, if you type *10 cm,* the shape's width changes to 10 centimeters. If there is no text or the text cannot be evaluated—for example, a nonnumeric value is typed—Width is zero. You can further refine the shape by resizing it only in the direction of growth, such as for a bar that grows to the right. To do this, use the rotation tool to move the shape's pin to the stationary end.

Changing the font size as a shape is resized

By default, when a user resizes a shape, its geometry and text block change but the font size doesn't. You can either use the SmartShape Wizard or write your own formulas to make font size a function of shape geometry.

Using the SmartShape Wizard. You can use the SmartShape Wizard to make font size a function of a shape's size. When a user resizes the shape, its text increases in proportion to the value of its height.

To use the SmartShape Wizard to resize text:

1. From the Tools menu, choose Macro, then Visio Extras, then SmartShape Wizard.
2. Under SmartShape Options, choose Customize Shape's Text, then click Change Option.

Font size in scaled drawings

If a shape is to be used in scaled drawings, you must take the drawing scale into account when you make font size a function of shape height. For example, the height of a desk in a space plan might be 1 m instead of 3 cm.

The font-sizing formulas adjust only the character size. Attributes such as text indents and line spacing do not scale as the shape is resized unless you use similar formulas in the cells that control those attributes.

3. Click Next until the Text Size screen is displayed, then choose Font Size Changes With Shape.
4. Follow the instructions onscreen to finish the wizard.

The wizard sets the font size to a proportion of shape height by adding the following formula to the Character section:

```
Char.Size   = 1 * Height * 0.1333
```

Writing custom resizing formulas. If you want a shape's size and its font size to resize proportionately, you can use this general formula:

```
(Height/<original height>) * (<original font size>)
```

To improve shape performance, store the proportional formula in a user-defined cell. For example, assume the original shape height is 3 cm and the original font size is 10 pt. Add the User-Defined Cells section in the ShapeSheet window, then add these formulas:

```
User.FontSize = Height/3cm * 10pt
Char.Size     = User.FontSize
```

To ensure that the font size is always readable, you can limit the range of acceptable sizes. For example, to limit font size to between 4 and 128 points, use the MIN and MAX functions with the proportional formula:

```
User.FontSize   = MIN(128pt,
                  MAX(4pt, Height/3cm * 10pt))
```

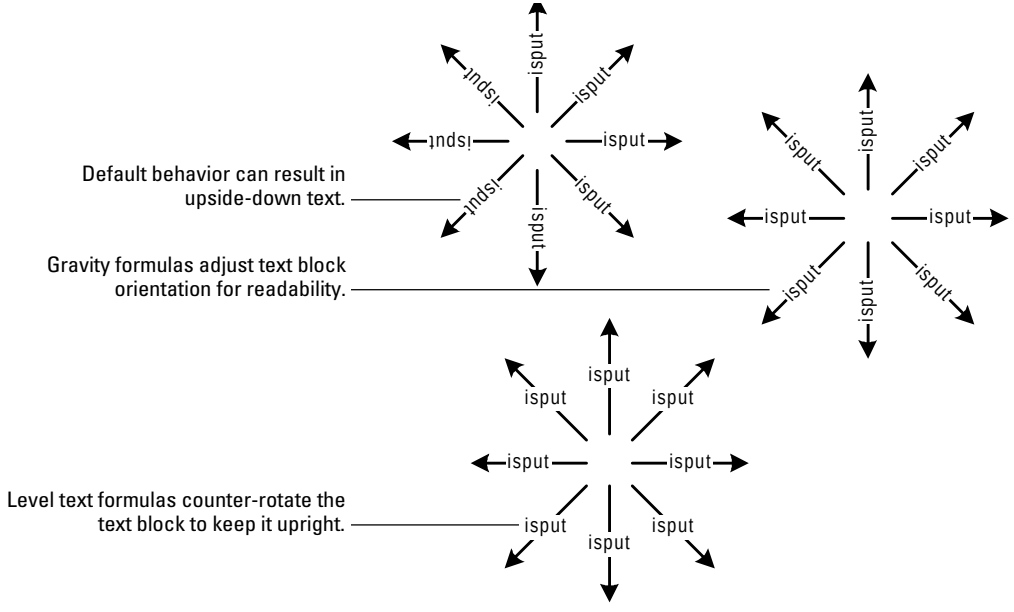
Be sure to use minimum and maximum font sizes that are supported by the expected printers and drivers. If the Character section for a shape contains more than one row, the Size cells in subsequent rows should use similar formulas.

Controlling text rotation

You can control the appearance of rotated text so that users don't have to read upside-down text. By default, when a shape is rotated, the text block rotates, too—a readability problem for shapes rotated between 90 and 270 degrees. If you are designing shapes for use in drawings where readability is an issue, you can customize text rotation behavior using one of the following methods:

- To prevent upside-down text as a shape is rotated, use the GRAVITY function, which orients the letter baseline toward the bottom or right edge of the page.
- To prevent text from ever rotating, use a counter-rotation formula to keep the text block level with respect to the bottom of the page as a shape is rotated.

You can use the SmartShape Wizard to create either text behavior. In addition, by using the wizard you can choose whether the gravity or level text block is centered over the shape or offset from it. For example, a text pointer like those shown in the following figures is formatted with a solid color background and remains centered on the shape. By contrast, if you were designing street shapes for a map, you might want to offset the street names from the lines that represent the streets.



Text can rotate with a shape, or not.

To use the ShapeSheet Wizard to create gravity or level text:

1. Select a shape, then from the Tools menu, choose Macro, then Visio Extras, then SmartShape Wizard.
2. Under SmartShape Options, choose Customize Shape's Text, then click Change Option.
3. In the Text Position screen, choose an offset or centered position for the text block.

   If you do not want a control handle added to the text block for positioning, be sure to check Do Not Add Control Handle.
4. Click Next until the Text Rotation screen is displayed, then choose Level Text or Gravity Text.
5. Follow the instructions onscreen to finish the wizard.

Gravity formulas

When you choose gravity behavior, the wizard adds this formula to the Text Transform section:

```
TxtAngle  = GRAVITY(Angle, -60deg., 120deg.)
```

In the formula, the two angles specify the range for which the default text orientation is rotated 180 degrees. When the shape is rotated between –60 and 120 degrees, the text would be oriented toward the top or left and so must be flipped. Using this formula, the text is upright for most angles of rotation.

If you also offset the text block from the shape, the wizard adds formulas to the TxtPinX and TxtPinY cells to shift the text block pin based on the shape's size and the amount of text.

Counter-rotation formulas for level text

If you used the wizard to create level text, the following formula is added to the Text Transform section to counter-rotate the text block as the shape is rotated:

```
TxtAngle = IF(BITXOR(FlipX, FlipY), Angle, –Angle)
```

Rotating a rotated text block

If you rotate a shape's text block (such that TxtAngle > 0 degrees), then rotate the shape, the amount of the shape's rotation is added to the value of TxtAngle.

This formula uses –Angle if the shape has been flipped in both dimensions or has not been flipped at all (if FlipX and FlipY are either both 1 or both 0). It keeps the original angle if the shape has been flipped in only one dimension (if either FlipX or FlipY is 1). Visio writes only the values 0 or 1 (for FALSE and TRUE) into the FlipX and FlipY cells, so you can safely assume these are the only values present. However, the following TxtAngle formula works in any case:

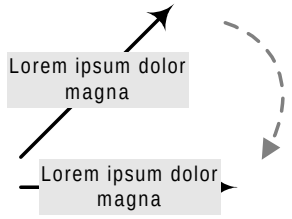
```
TxtAngle = IF(FlipX, -1, 1) * IF(FlipY, -1, 1)
           * (-Angle)
```

If the shape will never be flipped, you can use a simpler formula to counter-rotate the text block:

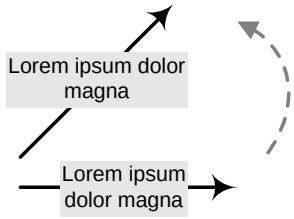
```
TxtAngle = -Angle
```

Constraining text block size: some examples

With the SmartShape Wizard, you can create level text behavior for a variety of common cases, but you may need greater control. When you're designing level text for a small shapes, the shape can become obscured by the text if a user types a lot of text or rotates the shape to certain angles. You can constrain the width of the text block to accommodate shapes using the methods described below.



Centered, level text can obscure the shape when rotated, and by default constrains the text block width.



Smart formulas widen the text block if the shape is rotated out of the way.

Constraining the width of a level text block

With some shapes, such as 1-D arrows or short shapes, counter-rotating text to keep it level isn't enough. As the shape rotates, the level text may obscure portions of the shape, as in the preceding figure. This is especially true when the text block is centered horizontally and vertically on the shape and has an opaque background. You can write formulas that keep the text block level and adjust its width as necessary when a user rotates the shape or adds text.

When you use the counter-rotation formula described earlier in this chapter, the text block stays level as the shape rotates. The default Text Transform formulas constrain text block width to shape width, which may not be useful or attractive if the shape is rotated and stretched. To constrain the text block width to the shape width only if the shape is within 15 degrees of horizontal, use the following formulas. In this example, when the shape is rotated beyond 15 degrees, the text block is set to a fixed width (2.5 in.).

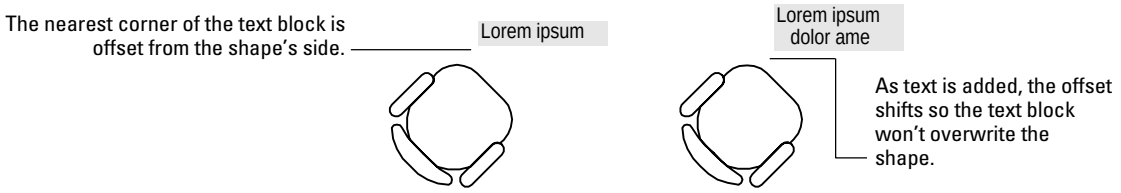
```
TxtWidth   = MAX(0.5 in., IF(Scratch.B1, 2.5 in.,
             Width – 0.25 in.))
TxtHeight  = 0.25 in.
TxtAngle   = IF(BITXOR(FlipX, FlipY), Angle, –Angle)
Scratch.A1 = DEG(MODULUS(Angle, 180 deg.))
Scratch.B1 = AND(Scratch.A1 >= 15 deg.,
             Scratch.A1 <= 165 deg.)
```

The formula in the TxtWidth cell above keeps the text block at least 0.5 inches wide for readability. If the shape is rotated beyond the limit, text block width is set to 2.5 inches; otherwise, it is set to the shape's width minus 0.25 inch to prevent the text from obscuring the shape. The formula in the B1 cell of the Scratch section performs the rotation test, returning 0 (FALSE) if the text block width is constrained by the shape width, or 1 (TRUE) if the text width is unconstrained. The formula in the A1 cell yields a shape angle normalized to a value from 0 degrees to 180 degrees to determine deflection from horizontal.

These formulas work most of the time, but they fail for short shapes that are close to the horizontal limit and have wide text. A more sophisticated solution would take the width of the shape and the composed width and depth of the text into account, but this solution would affect performance.

Controlling the width of an offset level text block

You can use the SmartShape Wizard to customize a shape's text block so that it remains level and is also offset from the shape. For example, in a space plan, you may want to move and rotate furniture but keep the labels right-side-up as viewed on the page, as the following figure shows. However, depending on the alignment of the text block, the shape's rotation, and the amount of text, the text block can obscure the shape. You can write formulas so that the position of the offset text block is automatically adjusted if the shape rotates or if text is added.



Level text offset from a shape

In the TxtAngle cell, the counter-rotation formula levels the text. The offset is calculated by requiring that, in the shape's local coordinate system, the left side of the text block be to the right of the edge of the shape. The following figure shows that the offset is the sum of line 1 and line 2. Line 1 is the leg of a right triangle whose hypotenuse equals TxtHeight/2, so its length is calculated using this formula:

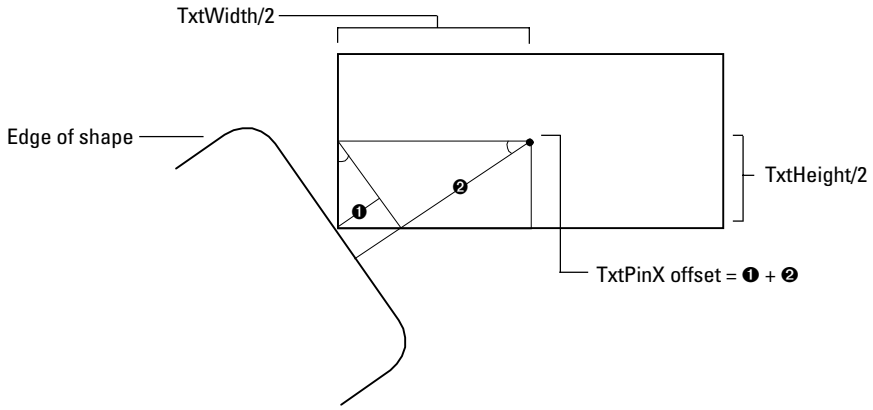
```
Line 1 = (TxtHeight/2) * ABS(SIN(Angle))
```

Line 2 is a leg of a right triangle whose hypotenuse equals TxtWidth/2, so its length is calculated using this formula:

```
Line 2 = (TxtWidth/2) * ABS(COS(Angle))
```

The offset is always a positive value, even when the shape is rotated at a negative angle, because we use the ABS function to return the absolute value for lines 1 and 2. The resulting formula looks like this:

```
TxtPinX = Width + (TxtWidth * ABS(COS(Angle))
          + TxtHeight * ABS(SIN(Angle))) /2
```



Calculating the text block offset

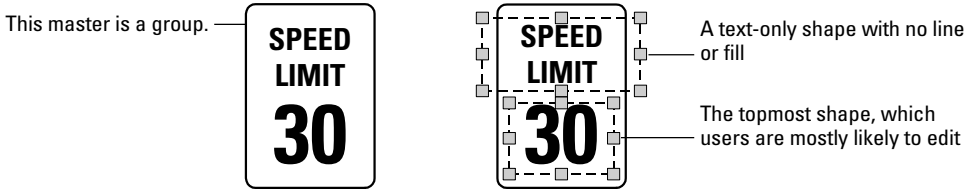
No matter how the shape is rotated, the text block stays offset from an imaginary boundary running along the shape's side. Calculating the offset this way means we don't need additional formulas to keep the text block from overwriting the shape as it rotates. In addition, custom formulas calculate the width and height of the text block based on the size of the text it contains. Following are the resulting formulas.

```
TxtWidth  = MAX(8 * Char.Size, TEXTWIDTH(theText))
TxtHeight = TEXTHEIGHT(theText, TxtWidth)
TxtAngle  = IF(BITXOR(FlipX, FlipY), Angle, -Angle)
TxtPinX   = Width + (TxtWidth * ABS(COS(Angle)) +
            TxtHeight * ABS(SIN(Angle)))/2
TxtPinY   = Height/2
```

Controlling text in a group

Only shapes have text blocks; groups do not. For groups composed of more than one shape, you need to decide which shape's text block receives the insertion point when a user selects the group and starts typing. By default, the topmost shape in a group receives the text. To add text to any other shape in the group, a user must subselect the shape.

The simplest way to designate the shape that will accept text is to move that shape to the front of the group with the Bring To Front command on the Shape menu. For readability, you can create a new shape with no line or fill for this purpose, as the following figure shows. This is particularly helpful if the shapes in your group have dark fill.



Use a group for a master that requires multiple text blocks with different styles.

The road sign shape was created as a group so that users could easily edit only the number portion of the sign without editing the "Speed Limit" label. In addition, by grouping you can use different text styles for the two grouped shapes. The master includes default text, such as "Speed Limit 30," which shows users where the shape can be edited. In such a shape, you can use the master's prompt to tell users to subselect the other text blocks for editing.

If more than one shape in a group can accept text, text block formulas become complex because they must access transform data in the ShapeSheets cells of both the enclosing group and the shape containing the text. It's best to isolate as many of the formulas as possible in the shape that has the text and use simpler formulas for text block positioning.

NOTE Verify that the group you design can accept text. If the topmost shape in a group is itself a group, when a user presses the F2 key to edit the shape's text, nothing happens, and the user must open the group and find a shape that contains a text block.

Displaying and formatting formula results

You can display the results of a ShapeSheet formula, such as in a text field, and format the output appropriately. When you choose the Text Field command from the Edit menu and select Custom Formula, Custom Properties, or User-Defined Properties, the text field created is really the value from the evaluated formula in a cell converted to text. You can use the same techniques to develop custom text fields as those you use in the ShapeSheet window, and display the formatted results in the shape itself.

When you create a custom formula for a text field using the Text Field command, the formula appears in the shape's Text Fields section. Visio displays the ShapeSheet formulas in the order they were inserted in the text, not necessarily the order in which they appear in the text. Deleting a text field in the drawing window does not automatically delete the corresponding text field row in the ShapeSheet window.

The following sections provide examples for using custom formulas in text fields.

Displaying a shape's width in different units

You can use text fields to show a shape's current width in inches, centimeters, points, or other units. To do this, you can use the FORMATEX function to specify the units you want to display for the result. The FORMATEX function takes this syntax:

```
FORMATEX(expression,"picture",["input-unit"],
["output-unit"])
```

This function returns the result of *expression* evaluted in *input unit* as a string formatted according to *picture* expressed in *output unit*. The format *picture* is a code that indicates how the result should be formatted. If you specify the optional *input unit* and *output unit*, use a numerical value or a valid spelled out or abbreviated unit of measure (in, in., inch, and so on). If you don't specify *input unit*, the units of the expression are not converted. If you don't specify *output unit*, the units of the result are used.

Units in earlier versions of Visio

The FORMATEX function is not available in versions of Visio prior to 4.5. If you are developing shapes for earlier versions of Visio, you can approximate the behavior of this function by specifying the units you want to display for the result as the first number-unit pair in a formula. For example, if you are inserting a text field to display a shape's width, you can enter this custom formula to ensure that Width is displayed in meters:

```
= 0 m + Width
```

You can use the same technique to coerce the units displayed with the result of any formula, not just one in a text field.

To use the FORMATEX function to display the shape's width in a text field:

1. Select a shape with the text tool.
2. From the Insert menu, choose Field.
3. In the Category section, choose Custom Formula.
4. In the Custom Formula box, enter an expression using the FORMATEX function, specifying the desired format picture, and input and output units.

   For example, if Width is in inches and you want to display it in centimeters, enter:

   ```
   = FORMATEX(Width,"0.00 u","in.","cm")
   ```

5. Click OK.

   Visio formats the value of Width using two decimal places, abbreviates the units, and converts it to centimeters. For example, if Width is 1.875 in., Visio displays 4.76 cm.

For details about valid format pictures that you can use, search online help for "format picture."

Displaying normalized angular values

You can design a shape that displays the current angle of rotation in its text box. For example, shapes representing lines of bearing on a nautical chart or slope indicators in a property line diagram display the current angle. By default, Visio returns angular values between –180 and +180 degrees. You can use the ANG360() function to convert the value of the shape's angle to a value between 0 and 360 degrees (or between 0 and 2π radians), then display the value in the shape.

To display the value of a normalized angle in a text field:

1. Drag a shape onto the drawing page, then select it.
2. From the Insert menu, choose Field.
3. In the Category section, choose Custom Formula.
4. In the Custom Formula box, enter:

   ```
   = ANG360(Angle)
   ```

5. In the Format section, choose Degrees.

Formatting strings and text output

When you display strings, such as formula results in a text field or custom property values, you can specify a format for the output. Text output can be formatted as a number-unit pair, string, date, time, duration, or currency. Visio recognizes a set of *format pictures* that format the text as you want it to appear. For example, the format picture "0 #/10 uu" formats the number-unit pair 10.9cm as "10 8/9 centimeters".

Format pictures appear in the list of formats when you choose Fields from the Insert menu, as arguments to the FORMAT and FORMATEX functions, and as formulas you can use in the Format cell of the Custom Properties section. For details about all the format pictures that you can use, including date, time, duration, currency, and scientific notations, search online help for "format function."

Using the FORMAT function

In any formula that resolves to a string, including custom text field formulas, you can use the FORMAT function to format the output. The FORMAT function uses the following syntax:

```
FORMAT(expression, "picture")
```

The result of *expression* is formatted according to the style specified by *picture*, which is enclosed in quotation marks. The function returns a string of the formatted output. The format picture must be compatible with the type of expression used, and you cannot mix expression types. For example, if you combine the formatting of a date and a number by using the number and date format pictures together ("#.## mmddyy"), Visio ignores the "mmddyy" portion and tries to evaluate the expression using the first part ("#.##") of the format picture.

To use the FORMAT function in a text field, specify a custom formula as described in the previous section, "Displaying and formatting formula results." In the Custom Formula box, include the FORMAT function in your formula.

The following table provides examples for formatting common number-unit pairs.

Custom text formats for number-unit pairs

Syntax	Display output
`FORMAT( 0ft. 11.53in., "0.## U")`	0 FT. 11.53 IN.
`FORMAT( 260.632 cm, "0.## u")`	260.63 cm.
`FORMAT( 0 ft. 11.53 in. , "# #/# u")`	11 5/9 in.
`FORMAT( 260.632 cm, "0 #/# uu")`	260 5/8 centimeters
`FORMAT( 260.632 cm, "0 #/5 uu")`	260 3/5 centimeters
`FORMAT( 0ft. 11.53in., "0.000 u")`	0 ft. 11.530 in.

Displaying formatted custom properties

You can format the displayed value of a custom property so that it appears the way you want in the Custom Properties dialog box. To do this, you use a format picture in the Format cell. In addition, you can display the value of a custom property in a text field. By using the Field command on the Insert menu, you can specify a custom property and a format picture for the value.

For example, a project timeline shape can have a custom property called Cost that measures the cost of a process. To format "1200" as currency, you can use the following format picture in the Custom Properties section:

```
Format = "$###,###.00"
```

In the U.S. English version of Microsoft Windows, the value is displayed in the Custom Properties dialog box as "$1,200.00". Under the German version of Windows, it appears as "DM1.200,00". Visio uses the current Regional Settings in the Windows Control Panel to determine the currency symbol, decimal character, and thousands separator to display.

In the Field dialog box, you can specify a custom property under Category. Visio displays a list of appropriate format pictures based on the custom property's data type.

If you intend to perform calculations with custom properties, you can define a data type other than string for the property value, such as number, currency, date or time, and duration. For details, see "Working with custom properties" in Chapter 4, "Enhancing shape behavior."

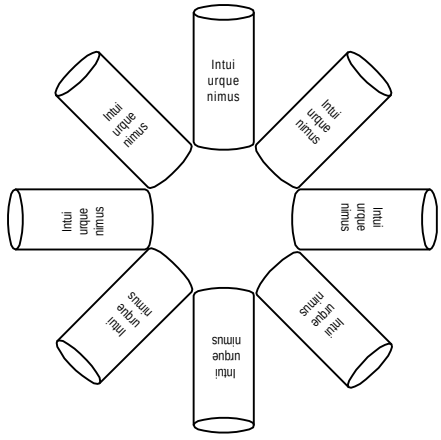
Testing text block formulas

The best way to test your text block formulas for a given situation is to try them. The following are procedures that we use at Visio to test the positioning and resizing of a shape's text block. To ensure that the position of the text block remains correct as a user manipulates the shape, you need to test all combinations of flipping, rotating, and reversing ends.

To test a shape's text block positioning:

1. Create an instance of the master you want to test, and then type some text in it.
2. Duplicate the instance seven times. Rotate each instance by increments of 45 degrees. Arrange the eight instances in a rosette. Group the instances for easier handling.

   This is a test set, and illustrates how the shape normally behaves under various rotations.



3. Duplicate the test set two times (for 2-D objects) or five times (for 1-D objects), and arrange as rows with three columns.
4. If testing a 1-D shape, select the three groups in the bottom row and choose Reverse Ends from the Shape menu.
5. Select the group(s) in the middle column and choose Flip Vertical from the Shape menu.

6. Select the group(s) in the right column and choose Flip Horizontal from the Shape menu.
7. Print the results and examine them in detail. Fix any problems and test again as needed.

Next you should test your shape's ability to handle text. To do this, you replace the test text in every shape, then check the results.

To test how a shape resizes as text is added:

1. Use the text tool to select one of the shapes, and then type new text.

   Type enough text to stretch the text block in a manner appropriate to the intended use of the shape.
2. Press Ctrl+A to select all the shapes.
3. Press F4 to repeat the new text in all the selected shapes.
4. Print the results and examine them. Fix any problems and test until you get the results you want.
5. As a final test, resize each group. Try both moderate and extreme sizes.

   Do they work the way you expected? Does the text still look good? Can you at least read it? If not, maybe you should specify a minimum text width. For details, see "Constraining text block size: some examples" earlier in this chapter.

7

Managing styles, formats, and colors

As a shape developer, you apply styles to the shapes you draw to ensure consistency. You also define the styles and custom options, such as custom fill patterns, that will appear in the templates you create for your users. Styles in Visio work a little differently from styles you may have used in other software, such as word-processing or spreadsheet programs. You can define and apply separate styles for text, lines, and fill, or styles that apply all of these attributes at once.

This chapter explains how to apply and create styles when you're working with shapes and provides guidelines for designing the styles that appear in your templates. It also explains how to change formatting attributes of the masters you work with and protect the styles of the masters you create. This chapter also describes how to create custom line patterns, fill patterns, and line ends that your users can apply just like any Visio format.

Topics in this chapter

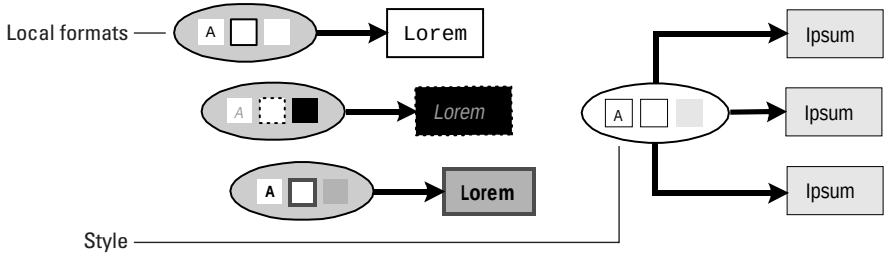
Working with styles in the drawing page

Styles are named collections of formatting attributes that you can apply to a shape. In Visio, a single style can define text, line, and fill attributes, so applying a style is an efficient way to promote consistency in your shapes.

When you apply a style to a shape, you are formatting the following attributes:

- For text, the font type, size, style (such as bold or italic), and color; text block alignment, margins, and background color; paragraph alignment, indents, and spacing; and tab spacing
- For lines, the line weight, color, pattern, cap, arrowhead style, and corner style
- For fills, the pattern and the foreground and background colors for a shape's interior (its *fill*) and for its shadow, if there is one

You can apply a style to a shape, or you can apply *local formatting* using the commands on the Format menu to achieve the same effect. If many of your shapes have the same format, styles are a more efficient use of computer resources than local formatting. A style definition is stored in only one place in a Visio document, and several shapes can refer to it. With local formatting, all the formatting instructions are stored separately for each shape. Shapes formatted using styles respond faster than locally formatted shapes when they are created, moved, scaled, and rotated.



Local formatting attributes are stored with each shape.

When you apply a style to a shape, the style definition stores the formatting information in one place.

In the documents you create, you can separately define styles for text, line, and fill attributes. Visio organizes styles on the toolbar by text, line, and fill attributes so that users of your templates can quickly apply the style they want. For ease of use, most styles in templates supplied by Visio affect only one set of attributes. For example, the Times Centered style changes only text attributes.

A single style can apply a combination of attributes. For example, one style can define a particular fill color and font. The style will appear on the toolbar under each attribute; in this case, under Fill and Text. When a user applies such a style from the toolbar, Visio asks if it should apply all the related styles at the same time or only the specific set of attributes the user selected.

When you apply a style to a shape that is locally formatted, the attributes defined in the style replace any corresponding local formatting. Locally formatted attributes that are not specified in the style are unaffected. For example, if a shape's line is locally formatted and you apply a text style that specifies only text formatting, the local formatting of the line remains intact, and only the text style changes. For details, see "Protecting local shape formats" later in this chapter.

Setting default styles for a drawing

When you are drawing a number of shapes, you can ensure consistency by specifying the styles that you use most as the document's default styles. Visio applies the default text, line, and fill styles currently set for a drawing page when you draw using any of the tools on the toolbar. You can also set default styles for a template's drawing page to help its users draw consistently or according to particular standards.

To change the default styles used by a drawing page:

1. Make sure nothing is selected and that the drawing page window is active, then choose Style from the Format menu.
2. In the Text Style, Line Style, and Fill Style boxes, select the new default styles you want, then click OK.

   The new default styles affect any shapes you subsequently draw with the drawing tools. Instances of masters dragged onto the drawing page are not affected—they inherit their styles from the master. The new default styles remain in effect for a drawing page until you change them again.

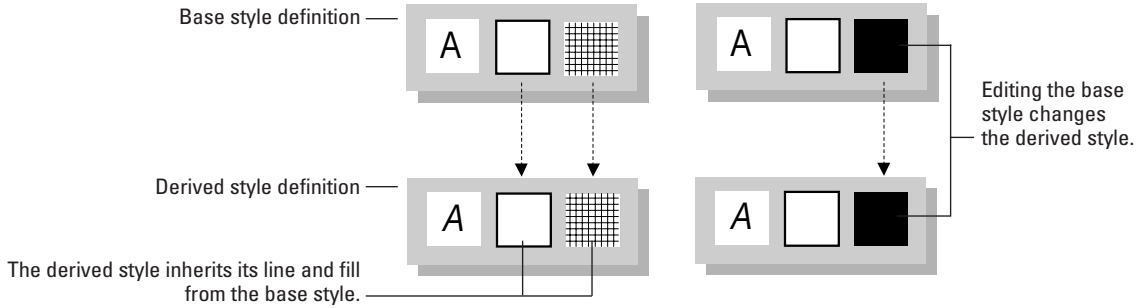
Applying a style

To apply a style to a shape, select the shape, then choose a style from the Text, Line, or Fill style lists on the toolbar. Or select the shape, then choose Style from the Format menu. Select a style from one or more of the boxes for the attributes you want to change, then click Apply. For details about applying styles and formatting, search online help for "styles."

Creating a new style

You can create a new style to include in your template or to quickly and consistently format several shapes. The styles you define in your templates appear to the user in the Text, Line, and Fill style lists on the toolbar and in the Style and Define Style dialog boxes.

You can create a new style from scratch or base it on an existing one. The advantage of creating new styles based on existing ones is that you develop a hierarchy of styles in which changing one style affects all of the styles that are based upon it, as the following figure shows. You must be careful, though, not to inadvertently edit a series of styles—and all the shapes formatted with those styles—when you mean to edit only one.



Deriving a new style from an existing, "base" style

When you create a new style, Visio fills in appropriate settings depending on whether the style is based on an existing one:

- A new style that you base on another style inherits the base style's attributes, and the new style name appears in the Fill, Text, or Line style lists as appropriate.
- If you are creating a style from scratch, Visio defines the default settings for the attributes you check under Includes in the Define Styles dialog box.

To create a style, choose Define Styles from the Format menu. The Based On option controls whether the style is based on another. After you have defined a style, you can apply it to shapes on a drawing page. For details about creating a style, search online help for "styles."

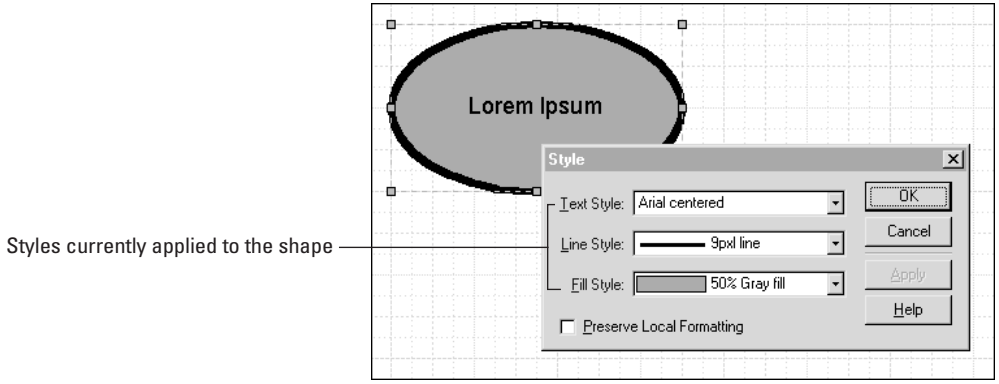
Copying styles between documents

If you have defined a style for one drawing file and want to use it in another, you can copy the style. To do this, drag a shape formatted with the style into the file lacking the style. (Or copy and paste the shape.) Then delete the shape. The style definition remains in the file. Be careful, though, that the destination file doesn't contain a style with the same name. If it does, when you copy the shape, the style definition doesn't get copied: The existing definition takes precedence.

Modifying the formats of shapes and masters

Whether you're dropping masters onto a drawing page, designing shapes for your own stencils, or creating a new template, using styles is an efficient way to format shapes. Visio offers several techniques for applying and editing styles. The technique you use depends on whether you want to reformat all shapes that use a particular style, reformat the master itself and so all subsequent instances of it, or change the instances currently on a drawing page, as follows:

- To change the appearance of all instances of a master on the drawing page as well as those you add later, you can edit the drawing file's styles.
- To change the appearance of a master, you can reformat it by applying different styles in the standalone stencil.
- To quickly reformat only the instances of a master on the drawing page, you can edit the copy of the master on the local stencil.



To determine the style currently used for a shape, select the shape, then choose Style from the Format menu.

Creating a style by example

When you add custom attributes to a shape, you can create a new style based on the example of the existing shape. That way, you can apply those custom attributes to other shapes as well. For example, say you have created a line and entered a ShapeSheet formula that evaluates to 3 mm in the LineWeight cell. If you are drawing many 3 mm lines, it's more efficient to create a style that you can reuse.

To create a style, select the example of the existing shape; for example, the line. Choose Define Styles from the Format menu. Choose New Style, then type a name for the style, such as "3 mm line." Visio fills in the attributes for the new style based on your selection—in this case, the line width is already set to 3 mm. Click Apply to close the dialog box and save the new style, which you can now apply to other shapes.

Editing a style to reformat shapes

You can edit a style to change the appearance of all shapes in a drawing page that use the style. To do this, use the Define Styles command on the Format menu to revise the text, line, or fill attributes of an existing style. All shapes formatted with the edited style are changed.

For example, say you're working with the Flowchart stencil, but you want text to appear in 10-point, italic, Times Roman type. Shapes from this stencil are formatted with the text style "Flowchart Normal." You can use the Define Styles command to change the style definition for "Flowchart Normal" to format text in the font you want. The new definition affects all shapes on the drawing page to which the style is applied as well as any new shapes you add that are formatted with that style.

A new style definition is saved only with the current drawing file. The standalone stencil and its masters are not changed. (The stencil file has its own style definitions.)

Reformatting masters in a standalone stencil

You can reformat masters in a standalone stencil by choosing new styles, and thus reformatting any instances subsequently created from those masters. Unlike editing a style to reformat the shapes that use it, this procedure changes the definition of the master in a stencil and saves the changes to the stencil. Use this procedure to edit masters in stencils you use in many different drawings.

To reformat a master with different styles:

1. Open the stencil file. Make sure Original is selected in the Open dialog box.
2. In the stencil window, double-click the master you want to edit to open it in the master drawing window.
3. Select the shape, or subselect the shape you want if the master is a group, then reformat the shape as you want it to appear.

   For example, choose Style from the Format menu, choose a text, line, or fill style to apply, then click OK.

4. In the master drawing window, click the Close box.

   When Visio prompts you to update the master, click Yes.

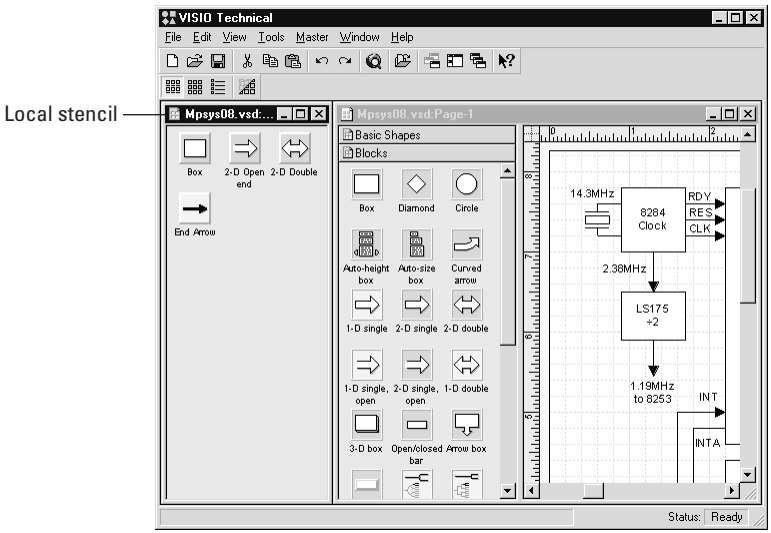
5. Make sure the stencil window is active, then choose Save from the File menu.

   The edited master is saved in the stencil. If you need to revert to the previous version of the master, you can edit it again to reformat it using the original styles. Or if it is a Visio stencil, you can reinstall the original from your Visio CD.

Reformatting all instances of a master

You can quickly reformat all instances of a master without changing either the master or its style definition. When you want to reformat instances, edit the copy of a master on the local stencil.

To do this, choose Show Master Shapes from the Window menu to open the local stencil, then double-click the master of the instances you want to affect. In the master drawing window, make the changes you want, then close the window and, when prompted, save your changes to see the effects on the drawing page.



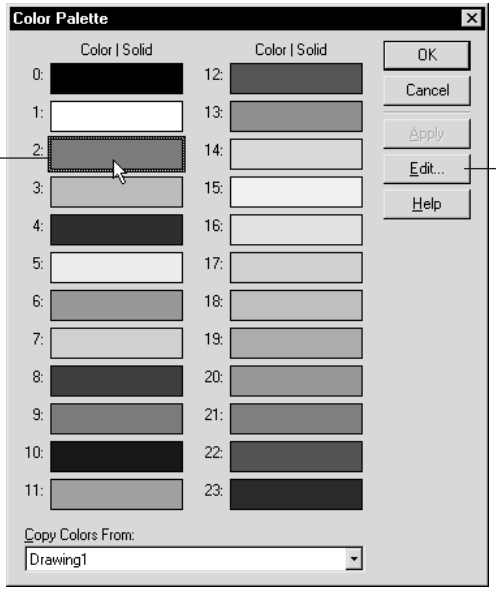
By editing the copy of a master on the local stencil, you edit all of its instances in the drawing page.

Managing color in styles, shapes, and files

When you are designing masters, you need to consider how the color of the master will look when used on different user systems. You can apply color to a shape using either the Visio color palette or a custom color that you define. The method you choose affects how the shape appears if used in another document. You can apply color to a shape using the following methods:

- By applying a color from the Visio color palette, you choose an *index* of one of the palette's colors. Visio records only the index to the color palette, not the color itself.
- By defining an RGB (red, green, blue) or HSL (hue, saturation, luminosity) value, either in the Color dialog box or as a ShapeSheet formula, you apply a *custom* color to a shape.

The color palette appears in the Color Palette dialog box, shown in the following illustration, as well as in the drop-down list of colors in the Fill, Line, Font, Text Block, and other dialog boxes. For any document that uses the default Visio palette, a color index refers to the same color: 0 is black, 1 is white, 2 is red, and so on.



To edit the color palette, select a color…

…then click Edit to display the Color dialog box, where you can pick a different color for that index.

Choose the Color Palette command from the Tools menu to display a document's color palette, which you can edit.

However, users can choose the color they want to appear at any index by editing the color palette. If they do, any shape mapped to that index can change color. For example, if you apply a fill color to a master by clicking red in the palette, Visio records the shape's fill color as 2. If a user creates an instance of the red master in a document in which the second index in the color palette has been edited, the shape's fill color will change to whatever color appears at index 2.

Most users do not edit a document's color palette, so a shifting color is not likely to be an issue. But you can ensure that a shape's color never changes, regardless of a document's color palette, by using a custom RGB or HSL color. To specify a custom color as a ShapeSheet formula, use either the RGB or HSL function.

Standardizing color palettes across documents

When you're designing stencils that you intend to open with a template, you should use the same color palette in all documents. If the color palettes do not match, the colors defined by an index in a master's styles can change when an instance is dragged into a document that has a different color value at that index. To standardize the color palette used in documents that are intended to open together, such as stencils and templates, you can copy the color palette used in one file to another.

If you edit the color palette in a stencil file, you can copy the colors to the drawing page in the same template.

To copy a stencil's color palette to a template:

1. Open the template file.
2. From the Tools menu, choose Color Palette.
3. Under Copy Colors From, select the stencil whose color palette you want to copy to the template file, then click OK.

   Be sure to save the document.

Specifying color as a ShapeSheet formula

You can define shape color using a function that specifies an RGB or HSL value. For example, to ensure that a stop-sign shape is always red, you can enter the following formula in the Fill Format section:

```
FillForegnd = RGB(255,0,0)
```

The RGB function's three arguments specify the red, green, and blue components of the color. Each can have a value from 0 to 255, inclusive. To specify the color using an HSL value, you could instead use the formula HSL(0,240,120) in the FillForegnd cell. For details about function syntax, see online help.

Rather than specifying color constants as the argument to these functions, you can use the RED, GREEN, and BLUE or HUE, SAT, and LUM functions to return the value of a color constant from the document's color palette or from another ShapeSheet cell. For example, in the stop-sign example above, RED(FillForegnd) returns 255, the value of the red component in the fill color. You can use the RGB and HSL functions together with the other color functions to define a color based on another cell's color in the same or a different shape. This is particularly useful in a group containing shapes of related but not identical colors. You can define the grouped shape's colors in terms of one shape's color in the group. For example, if the topmost shape in a group is Sheet.1, you could enter the following in Sheet.2:

```
FillForegnd    = RGB(RED(Sheet.1!FillForegnd),0,0)
```

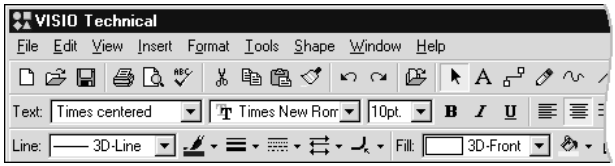
If a user applies a new color to the group, the topmost shape changes color, but Sheet.2 changes only the proportion of red in its fill color.

When you specify a custom color using the RGB or HSL functions, the color is added to the bottom of the color list in the Fill, Line, Font, and other dialog boxes in which you can assign color. If you create a master from a shape to which a custom color has been assigned, then drop an instance of it in another Visio document, the custom color is added to that document's color lists as well.

A custom color is saved only with the shape to which it has been applied. If you delete a shape with a custom color, and then save and close the document, the next time you open the document the custom color is no longer included in the color list of the different dialog boxes.

Using styles in stencils and templates

When you're designing stencils and templates for others to use, your styles should be consistent and easy to use. Users may perceive styles as the only formatting options available, so it's often better to include a larger number of styles in your templates than is strictly necessary.



Users frequently notice the toolbar's style lists before they discover the formatting commands on the Format menu.

The following sections provide tips for making sure your style definitions work consistently in the masters, stencils, and templates you create for your users.

Keeping styles consistent across files

When you create a stencil that will be used with a template, the style definitions should be the same in both the stencil and template files. When a user creates an instance of a master, the instance inherits the master's styles, which Visio applies as follows:

- If a style of the same name does not already exist in the drawing file, it is copied from the stencil file and added to the drawing file.
- If a style of the same name already exists in the drawing file, the existing style is used.

Visio style guidelines

At Visio, we develop styles according to these guidelines:

Text styles use TrueType fonts that ship with Windows. We limited our font choices to those we knew everyone using Windows 95, Windows 3.1, or Windows NT would have. However, if you know that users will have other fonts (especially fonts created for specialized markets, such as cartographic symbols), you can safely use those fonts in your text styles.

Fill and line styles use colors supported by a standard 16-color VGA monitor. We limited our color choices to those available on the most limited graphics system our users might have.

All styles are based on Normal, rather than on each other. When you have a hierarchy of styles based on each other, changing one style affects all of the styles that are based upon it. We thought this behavior might confuse inexperienced users, so our styles are not based on other styles. However, you may want to take advantage of this powerful feature of Visio in your solutions.

Most styles apply only one formatting attribute (fill, line, or text), or all three. Multiple-attribute styles can be confusing to inexperienced users. You might find, however, that your users always use one fill, one line, and one text style for a specific shape you're designing. This is the perfect opportunity to develop a style containing all three formatting types. Visio displays a message box to alert the user when such a style is applied from the toolbar.

If the style's definition in the drawing file differs from the definition in the stencil file, Visio uses the drawing's definition, and the shape's appearance in the drawing is different from that of the master. This behavior is sometimes referred to as the "home team wins" rule, because the style on the drawing page "wins" over the formatting attributes of a style with the same name in a master. It's not easy to ensure that styles are consistent. You can inspect each style definition, but this is tedious. One technique is to save a copy of the stencil file (.VSS) as a template file (.VST), delete all the masters in the template file, and save the workspace file (.VSW) to get identical styles and colors.

If you plan to save the drawing page as a stencil or template, you'll save file space by deleting any styles that are not used by your shapes. To do this, use the Define Styles command on the Format menu. Another method is to open a new drawing file that contains only the default styles, then drag the shapes formatted with the styles you want to copy into the new file. For details about cleaning up stencils and templates, see Chapter 9, "Packaging stencils and templates."

Using naming conventions for styles

The styles you create for your stencils and templates will be easier to use if you consistently follow a naming convention. Explicit style names, such as "Quarter-Inch Black Line" or "8pt Arial Left," are more expressive and understandable than abbreviated names, such as "Line2," or "T8L." Styles appear in alphabetical order in the toolbar lists and in the Style and Define Styles dialog boxes.

Good naming conventions keep related styles together in the lists, making it easier for users to find the styles they need. Line, fill, and text styles with similar attributes should have similar names. For example, if you name a 1-pixel-wide line style "1 Pixel Line," you should name a 3-pixel-wide line "3 Pixel Line," rather than "Line3." At Visio, we use one of two different conventions for naming styles, depending on how we expect the style to be used:

- Styles specific to a shape or stencil are named according to the shape (or shapes) they're applied to, such as Flow Connector Text.
- General-purpose styles are named according to their formatting attributes, such as Black Line or Arial Centered.

Deleted styles

What happens if a shape in the drawing or on the local stencil uses a style that you delete? If the deleted style was based on another style, the shape assumes the base style. If the style wasn't based on another, the shape assumes the No Style style, a default Visio style that cannot be deleted from a document.

TIP To make a style appear at the top of the style list in Visio, preface the style's name with a character that has a low ASCII value, such as a hyphen (-). For example, "- Standard Line" or "- Corporate Blue."

Protecting local shape formats

Applying a style can change the formulas in the Line Format, Fill Format, Text Block Format, Character, and Paragraph sections for a shape. Any local (custom) formulas in the related ShapeSheet cells can be overwritten. For example, you might write a custom formula in the Size cell of the Character section to dynamically change font size of your master based on text block height. If a user applies a different text style to the shape, the custom formula is overwritten.

As a Visio developer, you can protect a shape from both formatting and style changes by setting the LockFormat cell to 1 in the Protection section. If you protect a group in this manner, you automatically protect the shapes and other groups within it from inheriting formatting; however, users can subselect shapes in the group that are not explicitly locked and change their formatting. For details about protecting formatting in a group, see Chapter 3, "Controlling shape size and position."

You can use the GUARD function to prevent ShapeSheet formulas from changing when a user applies local formatting to a shape, but it doesn't prevent a style from being applied. The ShapeSheet cells reflect formatting values; they do not record the names of styles applied to a shape, so you can't guard against the application of styles. For example, if you protect the FillForegnd cell with GUARD, users cannot use the Fill command to edit the shape, but they can apply a fill style, which will overwrite the formula in the FillForegnd cell.

Use the LockFormat cell and GUARD function with care. When a shape is locked against formatting, Visio automatically displays a message when a user tries to format the shape. The GUARD function works without any notification or user messages. Either behavior may confuse or annoy users who want to format a protected shape. As you develop shapes, you must find the appropriate balance between limiting shape behavior and increasing user flexibility in your solution.

Creating custom patterns

You can create additional fill patterns, line patterns, and line ends. For ease of discussion, these styles are collectively termed *custom patterns* and appear to users as options in the Fill and Line dialog boxes. To design the custom pattern, you create a master that represents one instance of the pattern, such as a dot that, when applied as fill, looks like a complete pattern, such as polka dots. A *master pattern* is a special type of master that appears to end users only as an additional fill pattern, line pattern, or line end.

When you create a master pattern, you set its properties to specify:

- The master pattern name.
- The *type* of custom pattern: fill pattern, line pattern, or line end.
- The pattern's *behavior*—how the custom pattern is applied to a shape and how it changes as the shape is stretched or formatted.
- The custom pattern's use in scaled or unscaled drawings.

A custom pattern is always saved as a master pattern on a stencil. To distinguish master patterns from master shapes, the icon for a master pattern appears only if a stencil is opened as a copy or original. When a stencil containing master patterns is opened, the master pattern names appear in alphabetical order at the bottom of the appropriate list of options in the Fill or Line dialog box. Users can then apply the custom pattern as they would any standard pattern.

When a user applies a custom pattern, Visio copies its master pattern to the document's local stencil. The custom pattern then remains available in the active document, even if the standalone stencil containing the original master pattern is closed. If a user does not use a particular custom pattern during the current working session, it no longer appears in the Fill and Line dialog boxes after the stencil is closed. If a user copies a shape formatted with a custom pattern to another document, the usual inheritance rules apply: The master pattern is copied to the new document's local stencil, unless the new document already contains a master pattern of the same name, in which case the local master of that name is applied to the shape.

Many of the techniques that you use to develop master shapes also apply to developing master patterns. For example, you—and your users—will have more predictable results if you use a single shape or a group in your master (shape or pattern). You can always combine multiple geometries to create a single shape using the commands on

Custom patterns as formulas

When a user applies a custom pattern to a shape, Visio records the choice by inserting the USE function in the FillPattern, LinePattern, BeginArrow, or EndArrow cell. For example, if a user applies a custom line end called Star to the begin point of a line, the BeginArrow cell of the line will contain the formula USE("Star").

the Operations submenu of the Shape menu. In addition, you should create a master pattern as a single instance of the minimum design required to repeat as intended.

NOTE Do not use text or a bitmap in a master pattern. Neither would appear when the pattern is applied to a shape.

To create a custom pattern:

1. Open a new stencil, or open an existing stencil as a copy or original.
2. From the Master menu, choose New Master.
3. In the Master Name box, type a name for the custom pattern as you want it to appear in the Fill and Line dialog boxes.
4. Under Master Type, choose Line Pattern, Line End, or Fill Pattern.
5. In the Behavior box, choose an option to specify how the pattern is applied to a shape.

   For details about custom pattern behavior, see the following sections.
6. Check Scaled if the custom pattern models an object with real-world dimensions.

   For example, if you're creating a fill pattern of 4-inch-square kitchen tiles, check Scaled to preserve the pattern dimensions when it's applied to a shape on a scaled drawing page.
7. Click OK to add a new, empty master to the stencil.
8. Double-click the master to open the master drawing window, where you can draw the custom pattern you want.

   If you want users to be able to change the color of a pattern or line end after it's applied to a shape, design the master pattern in black and white, as described in the following sections.
9. After you create the pattern, close the master drawing window, and save your changes to the stencil.

The master icon for the custom pattern appears in a stencil open as a copy or original. If you open the stencil as read-only, however, the icon does not appear. When a custom pattern is applied to a shape, its master icon appears on the document's local stencil.

About a pattern's alignment box

Visio applies custom fill patterns, line patterns, and line ends based on the size of their alignment box. To ensure that your pattern or line end works as expected, you can design an alignment box with dimensions that differ from the size of the pattern itself.

For details about creating a custom-size alignment box, see "Adjusting a shape's alignment box" in Chapter 8, "Scaling, snapping, and aligning."

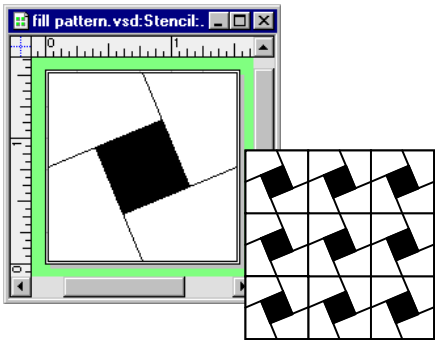
Developing custom fill patterns

You can design custom fill patterns that fill a 2-D shape in one of three ways, depending on the behavior you choose in the Properties dialog box for the master. The most common type of fill pattern behavior is *tiled*, where instances of the pattern are repeated to fill the shape from the lower-left corner outward, as shown in the following figure.

Tiles the pattern from the lower-left corner of a 2-D shape

Centers the pin of one pattern instance on a shape's pin

Stretches one pattern instance to fill an entire shape



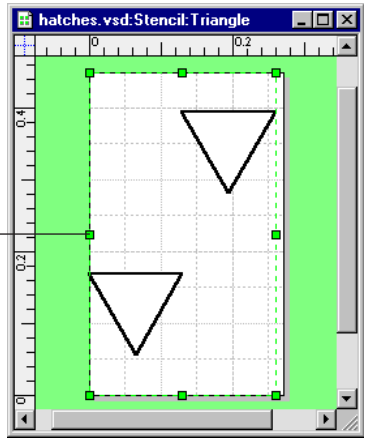
To create a patterned floor tile, instances of the pattern are repeated to fill the shape from the lower-left corner outward.

You can also create a centered or stretched fill pattern. In a *centered* pattern, a single instance of the pattern fills the shape. Visio aligns the pattern's pin with the shape's pin. In a *stretched* pattern, a single instance of the pattern is stretched horizontally and vertically to fill a shape. Visio disregards the position of the pattern's pin. As you resize the shape, the pattern resizes, too, unlike the built-in patterns.

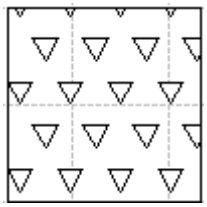
Fill pattern colors. If you design your fill pattern in black and white, users can set the pattern color when they apply it to a shape as they can any Visio pattern. White areas (line or fill) in your pattern inherit the foreground fill color of the shape to which the pattern is applied; black areas (line or fill) in your pattern inherit the shape's background fill color. If your pattern contains any colors other than black and white, the pattern retains those colors when applied to a shape.

Designing tiled patterns. The most common fill pattern behavior is tiling, in which the pattern is tiled by the edges of its alignment box. You can get different tiling effects by creating a pattern with a larger or smaller alignment box, as the following figure shows, or by placing the pattern off-center within its alignment box. For details about creating a custom-size alignment box, see "Adjusting a shape's alignment box" in Chapter 8, "Scaling, snapping, and aligning."

The master pattern includes two offset triangle shapes in a large alignment box.

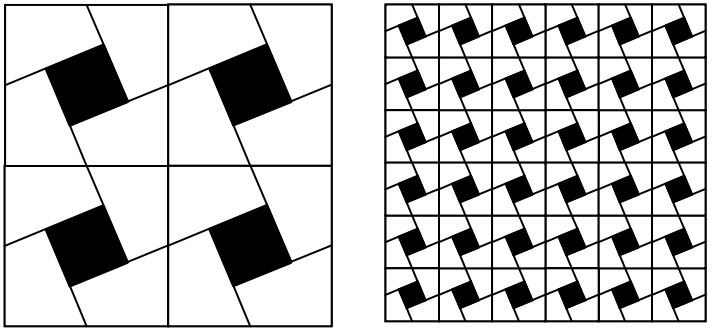


A tiled fill pattern with a custom alignment box.



The pattern fills the shape from the lower-left corner.

When your tiled pattern represents a real-world object, check Scaled in the Properties dialog box for the master. For example, a 1-ft by 1-ft ceramic floor tile is always the same size, regardless of the drawing scale in which it is used. The default fill pattern behavior is unscaled, which means the pattern behaves like the built-in Visio line patterns: They always print at the same size, regardless of drawing scale.



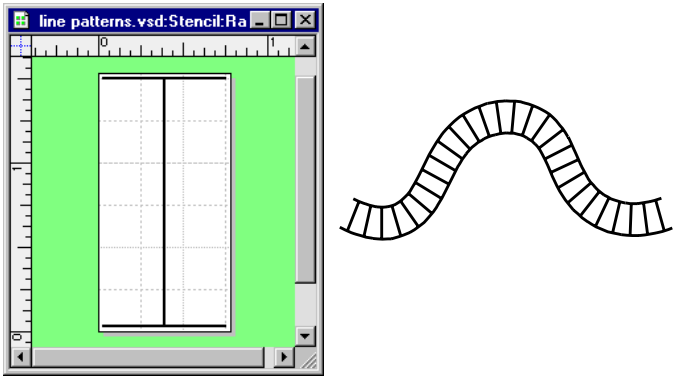
On a drawing page that uses an architectural scale, an unscaled pattern looks the same as on a page with no scale, but a scaled pattern retains its dimensions.

Developing custom line patterns

By applying a custom line pattern, a user can reformat a line as railroad track, a garden path of stepping-stones, or any other line pattern. When you design a line pattern, consider how the pattern repeats along the length of the line and around curves and corners. Consider also whether the pattern should be resized when the line weight changes. These considerations—the pattern's behavior—determine the manner in which Visio applies the pattern to a line and can dramatically affect the line's appearance.

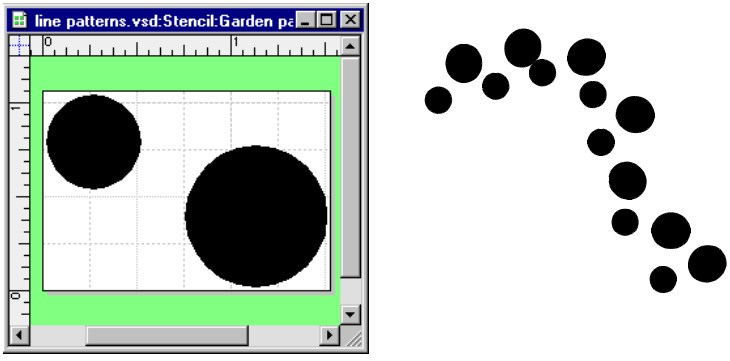
You choose a Behavior option in the Properties dialog box for the master to control how a line pattern is applied to a line. You can design line patterns to behave in one of four ways, as the following illustrations show.

Bends instances of the pattern to fit a curved line



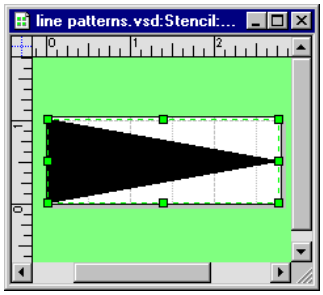
To create a railroad track, each instance of the pattern is bent to fit around curves as it repeats along the length of the line.

Repeats instances of the pattern to fit a line without bending around curves



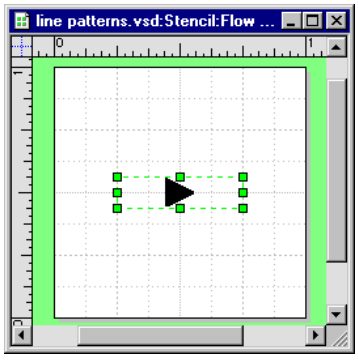
To create a garden path, each instance of the pattern is positioned and rotated as it is repeated along the length of a line. The pattern instances don't bend.

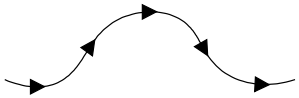
Stretches a single instance of the pattern along the length of a line



To create a tapered line, a single instance of the pattern is stretched along the entire length of a spline.

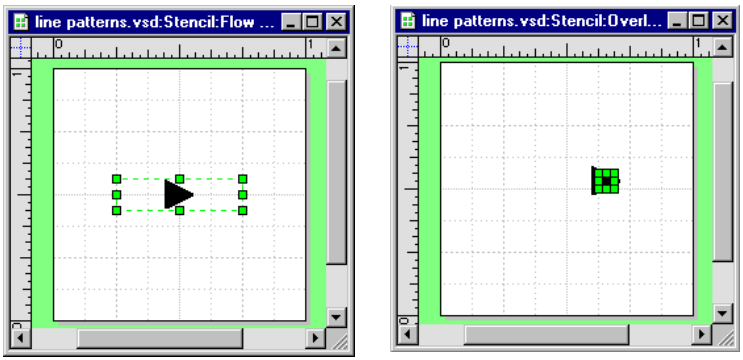
Repeats instances of the pattern on top of a line for a "string of beads" effect





To create a flow line, the pattern is repeated on top of the line, fitting whole instances of the pattern between corners. The alignment box is larger than the arrowhead to control the spacing between instances of the pattern.

Customizing the alignment box and pin. To design an effective line pattern, you must consider the size of the alignment box and pin position as well as the shape of the pattern. In fitting a pattern to a line, Visio aligns the pattern's pin to the line and repeats or stretches the pattern by the edges of its alignment box. If the alignment box is larger than the pattern, Visio leaves spaces between pattern instances as it repeats the pattern on the line. This is how you would create a stripe that repeats at precise intervals. If the alignment box is smaller than the pattern, you'll get an overlapping effect when the pattern is applied. For details about creating a custom-size alignment box, see "Adjusting a shape's alignment box" in Chapter 8, "Scaling, snapping, and aligning."



By changing a line pattern's alignment box, you can control how instances of the pattern repeat along a line.

Line with Flow Arrow pattern applied

Line with Overlap Arrow pattern applied

Scaled versus unscaled line patterns. If you design an unscaled line pattern (that is, the Scaled option is unchecked in the Properties dialog box for the master), when a user applies the line pattern, Visio resizes its alignment box until its height equals the line weight. Scaled line patterns keep their dimensions regardless of the drawing scale or the line weight.

Color in line patterns. When you design a line pattern, apply black to the areas (line or fill) that you want users to be able to change by choosing a new color in the Line dialog box. Apply white or any other color to the areas you don't want users to be able to change. Set the fill of your line pattern to None if you want the fill area to be transparent when applied to a line.

Developing custom line ends

Keeps the line end straight with respect to the line

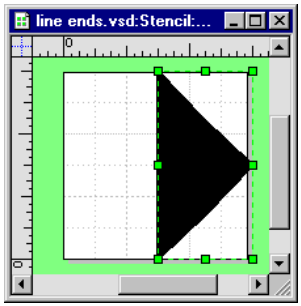
Keeps the line end upright with respect to the page

A custom line end is the simplest type of custom pattern to create—it's simply a shape that can attach to the endpoint of a line. When you design a line end, you determine whether it can adjust to the direction of the line to which it's attached and whether it resizes as the line weight changes. You can design a line end to:

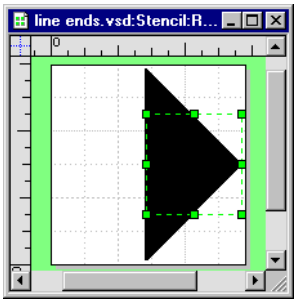
- Orient itself with respect to the line. If you move the line, the line end adjusts to point in the same direction.
- Orient itself with respect to the page. If you move the line, the line end remains upright as viewed on the page.

Visio attaches the pin of the line end to the endpoint of a line. If the line end behavior is to orient with respect to the line, Visio trims the line between its endpoint and the bounding box of the line end for a seamless look. Otherwise, the line is not trimmed. As you design a line end, consider where to place the pin to achieve the right effect. For example, to design an arrowhead, you would draw a triangle, then move the pin to the pointing tip.

NOTE A line end must point to the right; otherwise, it won't be applied properly.



The Simple Arrowhead is a right triangle with black fill.



The Refined Arrowhead is a group with an alignment box that is slightly narrower than the triangle. The pin was moved to the arrowhead's point.

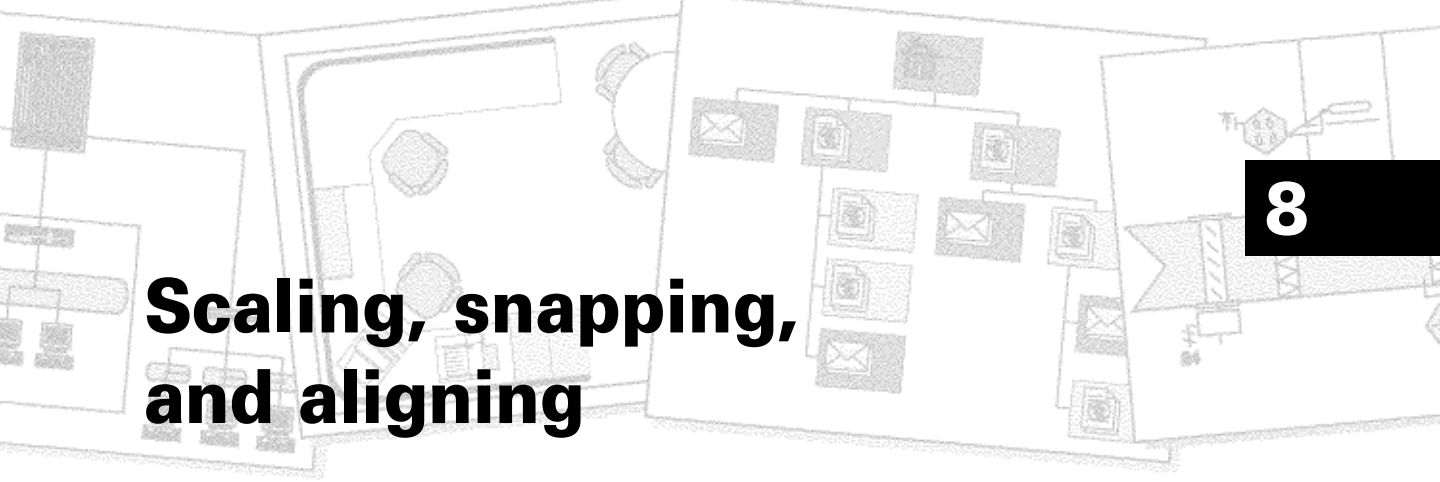
The Simple Arrowhead line end applied to a 36-pixel line.

The Refined Arrowhead line end applied to a 36-pixel line.

TIP To move a shape's pin, select the shape, choose Size & Position from the Shape menu, then click in the Position box grid. Or select the shape with the Rotation tool, then drag the pin to a different position.

Another consideration in designing a line end is whether its size should be affected by the line weight of the line to which it is applied. If you design an unscaled line end (that is, leave Scale unchecked in the Properties dialog box for the master), Visio will set the height of the line end's alignment box to equal the line weight as long as the user sets Size to Medium (the default) in the Line dialog box. However, on a 1-pixel line, the line end may not be visible. To ensure that your line end works at any line weight, you can customize its alignment box. If a user sets Size to something other than Medium, the line end resizes in the same way any line end resizes. For details about creating a custom-size alignment box, see "Adjusting a shape's alignment box" in Chapter 8, "Scaling, snapping, and aligning."

If your line end represents an object with real-world dimensions, such as a fitting at the end of a pipe, check Scaled in the Properties dialog box for the master. The Size and Weight settings in the Line dialog box will have no effect on the size of a scaled line end.



8

Scaling, snapping, and aligning

When the drawings your users create represent real-world objects, they need shapes and templates that draw to scale. You can design masters that size appropriately when users drag them into a drawing page with a scale, such as 1/4 inch = 1 foot. If you design the template as well, you can ensure that the scale of the drawing page works with the scale of the masters you provide, and thereby simplify a complicated drawing task for your users.

This chapter explains how to choose an appropriate scale for drawings and shapes that need to be scaled. In addition, it describes the effect of rotated pages on shapes and guides, and how to choose the appropriate-size drawing grid and create shapes that align precisely to the chosen grid.

Topics in this chapter

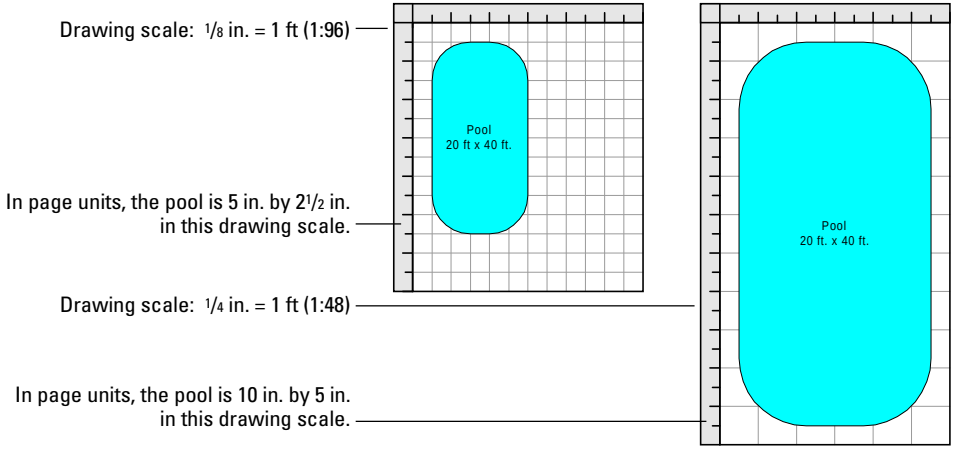
Choosing an appropriate drawing scale

Any drawing that depicts physical objects that are too small or too large to be drawn easily, or are larger than the paper size, must be scaled to fit on the page. For example, in an architectural rendering of a house, 1/4 inch on the drawing page might represent 1 foot of the actual house. Schematic diagrams such as flowcharts and organization charts depict abstract objects, so these types of drawings are unscaled, and shapes appear at their actual size.

In Visio, *drawing units* are sizes in the real world. In the house example above, 1 foot is the drawing unit. *Page units* are sizes on the printed page—1/4 inch in the house example. The ratio of page units to drawing units is the *drawing scale*.

ShapeSheet cells that describe object size or position—that is, most cells—are expressed in drawing units. Cells that represent measurements on the printed page, such as text format and indents, are shown in page units. If the drawing scale is changed, all ShapeSheet cells that are expressed in drawing units remain constant, but the shape is redrawn to the new scale.

In the the following figure, the swimming pool is 40 feet long and 20 feet wide, drawn using a 1-point line, and labeled using 8-point type. With a drawing scale of 1/4 inch = 1 foot (1:48), the picture of the pool is drawn 10 inches long by 5 inches wide. If you change the drawing scale to 1/8 inch = 1 foot (1:96), the pool is still 40 feet long and 20 feet wide; however, the picture of the pool is now only 5 inches by 2 1/2 inches. Regardless of the scale, the line size remains 1 point and the font size 8 points.



In drawing units, the pool is 40 ft by 20 ft regardless of the drawing scale.

To choose the appropriate drawing scale to include in a template, consider the following:

- The expected size of the drawing in drawing units
- The paper size users will print their drawings on
- The industry or drawing conventions that apply to the drawing type users create with your template, such as margins or title blocks

For example, a user can print a house plan on an $8\frac{1}{2}$-inch by 11-inch sheet of paper in landscape orientation. If the drawing scale is $\frac{1}{4}$ inch = 1 foot, the drawing page represents 34 feet by 44 feet (assuming no margins). This may not be large enough to accommodate the house and its landscape design. Instead you might choose a larger scale, such as $\frac{1}{8}$ inch = 1 foot or 1 inch = 10 feet.

You can also use elapsed time rather than elapsed distance for a page scale by setting the drawing units to hours, days, weeks, months, and so on. For example, you can use elapsed weeks (abbreviated "ew.") as the drawing units for the diagram of a project timeline.

To set the drawing scale for a page:

1. From the File menu, choose Page Setup, then click the Page Properties tab.
2. In the Measurement Units box, choose the drawing units you want, then click the Page Size tab.
3. Under Page Size, choose the orientation and size of paper the drawing will be printed on.

   The values in the Page Size tab show you the drawing unit measurements of your page according to the selected scale and paper size.
4. On the Drawing Scale tab, choose the type of scale you want.

   For details about options, click the Help button.

TIP To ensure that a master you create matches the drawing scale for a template's page, edit the master and repeat the preceding procedure in the master drawing window. For details, see "Setting the master's scale" later in this chapter.

Page units in ShapeSheet cells

Most ShapeSheet cells that reflect size or position represent drawing units. Cells that reflect page units include paragraph properties and text margins. A complete list of the cells that represent page units is shown below:

- Text Block section: TopMargin, BottomMargin, LeftMargin, RightMargin
- Paragraph section: IndFirst, IndLeft, IndRight, SpLine, SpBefore, SpAfter
- Line Format section: LineWeight
- Character section: Size
- Page Properties section: ShdwOffsetX, ShdwOffsetY, PageScale

Choosing a scale for masters

Masters can be scaled, as well as drawing pages. A shape's appearance on the drawing page depends on the master's scale and the drawing page's scale. If either a shape or the page is scaled and the other is unscaled or has a very different scale, the shape can behave in unexpected ways when the user drags it onto the page. If users aren't aware of scaling differences, they may become frustrated when they try to use shapes on a page with an incompatible scale.

Although you can't prevent users from creating a new drawing of any scale and dragging your shapes into it, you can make sure that the drawing pages you provide with your templates have drawing scales that match those used in your masters. You can also create masters that work in as many different drawing scales as possible.

Determining an appropriate master scale

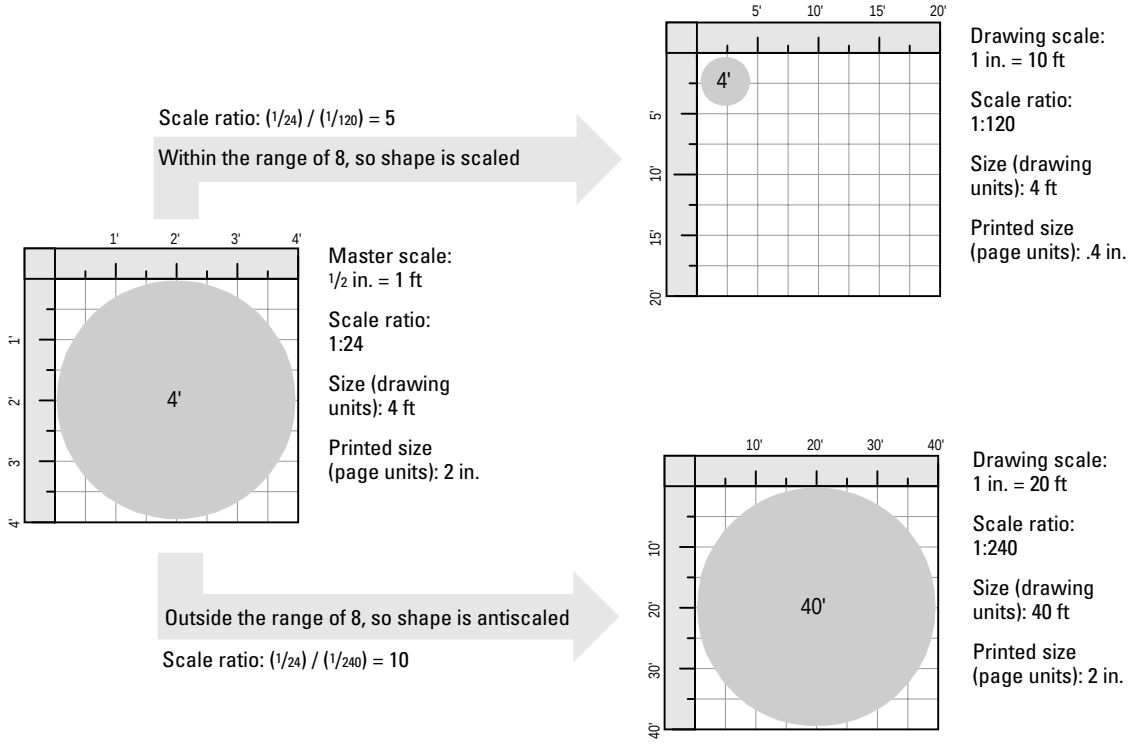
It is always best if the drawing scale of a master matches the drawing scale of the page it is dropped on. This is not always possible, however, so within certain limits Visio handles differences of scale by making sure that the shape as drawn is the same size *in drawing units* as the master. This limit is known as the "range of eight." If the scale of the shape does not differ from that of the drawing page by more than a factor of eight—that is, if the drawing scale of the master is no more than eight times greater or smaller than the drawing scale of the page—Visio calculates the shape's size in drawing units and scales it appropriately on the drawing page. This behavior prevents a shape from becoming so large that it obscures the drawing page or so small that you can't see it.

You can create a master of a table that can be used in space planning templates that vary in scale from 1/2 inch = 1 foot (drawing scale of 1:24) to 1 inch = 10 feet (drawing scale of 1:120). In the figure on the next page, when a 48-inch table shape is dragged into a drawing whose scale doesn't differ by more than a factor of eight, the table is properly scaled. The Shape Transform section shows its width is still 48 inches.

If the difference in scales exceeds a factor of eight, Visio *antiscales* the shape: The shape is drawn in the same size *in page units* as the size of the master. The user can resize the shape once it is dropped. For example, in the figure on the next page, when the table shape is dragged into a drawing whose scale is outside the range of eight, the shape appears at the same size in page units as the master (2 inches), but Visio recalculates its width using the drawing scale of the page.

Changing the range of eight

You can change the range of eight—it's a setting in the VISIO.INI file. To set the range to a different factor, change the value for the AutoScaleConversionRatio setting in VISIO.INI. Only the version of Visio running on your computer is affected. If you plan to distribute your shapes, you must still design them with the range of eight in mind, or change the AutoScaleConversionRatio setting for all users.



How shapes are redrawn at different scales according to the range of eight

Visio applies the range of eight rule only to width and height values. Constants in formulas are not adjusted. So, for example, typing the following formula in a cell of the Geometry section may cause unexpected results:

```
Width – 1 ft
```

Because Visio changes the shape's width, the Width reference will be scaled, but 1 foot will remain 1 foot in drawing units, so the shape may still look strange even after it has been correctly scaled.

To take advantage of the range of eight in designing your masters, try one of these tips:

- Set the scale of a master in between the largest and smallest scales in which the master is likely to be used. This way, the master works with the greatest range of drawing scales within the range of eight. This "middle scale" can be calculated as the square root of the largest drawing scale ratio times the smallest drawing scale ratio.
- Set the master scale to an extreme scale so that the shape always antiscales when dropped on the page. For example, use a scale such as 1000 inches = 1 inch, which is well outside the range of eight.

Setting the master's scale

In general, you should set the scale of a master equal to the scale of the drawing page the master will be used with. By default, a master uses the scale of the drawing page on which it was created, before the shape was dragged into a stencil. Or if you use the New Master command to create a master directly on the stencil, by default the master is unscaled. To set a master's scale, use the Master Setup command, which is available only when you are working in the master drawing window.

To set the scale for a master:

1. Select a master in your stencil, then choose Edit Master from the Master menu.

   NOTE To edit a master, the original stencil file must be opened. If the stencil is opened as read-only, you cannot edit its masters.

2. From the File menu, choose Page Setup, then click the Drawing Scale tab.

3. Under Drawing Scale, choose the scale you want.

   Choose Architectural, Civil Engineering, or Mechanical Engineering to choose from among the built-in industry-standard scales for these professions. Choose Metric to set a standard metric page scale ratio. Choose Custom Scale to enter a different scale.

Creating antiscaled shapes that are never scaled

You can create antiscaled masters—shapes that are the same size in page units for all drawing scales. For example, a title block in an architectural drawing or a legend in a map should remain the same size no matter what scale is used in the drawing. Visio has two page formulas that allow you to determine the scale: ThePage!PageScale and ThePage!DrawingScale. You can write an antiscaling formula that uses the ratio of these two values to convert a value expressed in page units to its equivalent in drawing units.

To convert a page unit value into the equivalent drawing unit value, multiply by this ratio:

```
ThePage!DrawingScale / ThePage!PageScale
```

If you write a custom formula for a master using this ratio, users can drag the shape into any drawing scale, and the shape's scale does not change. For example, to create a shape that is always 5 cm wide on paper, enter this formula in the Shape Transform section:

```
Width  = GUARD(5 cm * (ThePage!DrawingScale /
         ThePage!PageScale))
```

If you want users to be able to resize the shape, do not use the GUARD function. When a user creates an instance from this master on a page that has a scale of 1-cm = 1 m, the Width formula is reevaluated for the destination page's scale:

```
= 5 cm * (1 m / 1 cm)
= 5 cm * 100
= 500 cm
```

When the shape is printed or displayed onscreen at actual size, Visio scales the 500-cm shape to 5 cm.

If you are creating a number of antiscaled masters, you may find it more efficient to store the antiscaling formula in a user-defined cell of the page sheet, such as User.Antiscale. That way, you can quickly edit the antiscaling formula by changing values in only one place, the page sheet. The formula of any antiscaled master becomes:

```
Width  = 5 cm * ThePage!User.AntiScale
```

Working with rotated pages

You can enable page rotation so that a user can rotate a page in a drawing window. This option is enabled by default in Visio Technical 5.0. For other Visio products, you can enable it by choosing Options from the Tools menu, clicking the Advanced tab, then checking the Enable Page Rotation option. When this option is enabled, users can rotate the view of a page by clicking the Rotation Tool button on the Standard toolbar, then dragging a page corner in the drawing window.

When you open a window and display a page, it is rotated according to the current value of its Angle cell. This cell is not visible in the ShapeSheet window, but you can set it through Automation methods. These three events can affect the value of the Angle cell:

- Dragging the page corner with the rotation tool in a previously opened window
- Setting the Angle cell to a new value through Automation
- Disabling the Enable Page Rotation option, which resets the Angle cell to 0 for every page currently displayed

You can display a page in more than one drawing window and display a different angle of rotation for each window.

When you rotate a page, you affect only how a drawing window displays the page. Shapes on a page are expressed in the same coordinates regardless of the page's rotation. In addition, rotating a page doesn't affect the page's appearance when printed or the appearance of the rulers and grid in the window. The rulers and grid are always displayed perpendicular to the window frame, not to the page, even if the page is rotated. In a rotated page, shapes snap and align with respect to the rulers' and grid's orientation, not the page's.

Although a page and a master both have Transform cells, they do not display a Transform section in their ShapeSheet windows. You can access a page's Transform cells through Automation, but Visio reserves a master's Transform cells for internal use, and you cannot access them programmatically.

The only Transform cell that affects the behavior of a page in Visio is its Angle cell, which you can access through the **PageSheet** property. For details about changing the value of a cell through Automation, see "Working with formulas" in Chapter 14, "Working with drawings and shapes."

Designing a grid

In Visio, by default the drawing page displays a grid. If you design the scale of your shapes and drawing pages with the grid in mind, your users can quickly snap a drawing into place.

This section contains instructions for designing the grid to complement your masters and templates, and tips for creating shapes that work within the grid. Not all shapes need to snap to a grid, and not all templates require a customized grid. For most technical drawings, however, the grid is a useful tool that you should consider when designing your masters. To hide the grid, uncheck Grid on the View menu.

Setting a template's grid

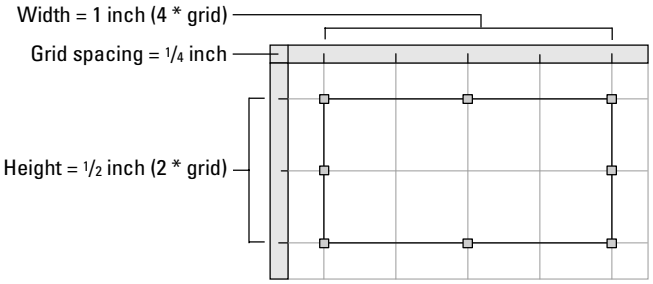
When you set up the drawing page in a template, you can decide whether the grid is variable or fixed. With a *variable* grid, the grid increments change as you zoom in and out. A *fixed* grid displays the same increments at every magnification. With either type, you can set how finely the grid and rulers are subdivided. In any view, users should be able to easily snap to a grid that works with the grid spacing used for the masters.

To set the grid spacing for a template, choose Ruler & Grid from the Tools menu. The settings in the Ruler & Grid dialog box are stored in the page sheet in the Ruler & Grid section. The variable grid settings are stored in the XGridDensity and YGridDensity cells. The fixed grid settings are stored in the XGridSpacing and YGridSpacing cells. To add this section when you are viewing a page sheet, from the Insert menu, choose Sections, and then check Ruler & Grid.

Creating a master that works with the grid

If you design masters so their dimensions are multiples of an underlying grid spacing, users can take advantage of the Visio snap-to-grid feature to drag shapes into precise positions quickly. When snapping to grid lines is enabled, the edges of a 2-D shape's alignment box snap to visible grid lines, showing the user the exact position of the shape. In addition, when a user drags a master from the stencil, the instance is easily aligned on grid lines when dropped.

For a 2-D shape, the snap-to-grid action is most useful if both the width and the height of the shape are multiples of the spacing of the currently displayed grid, as the following figure shows. If this is not the case, opposite edges of the object snap separately, the dragging behavior of the shape is jerky, and users must pay attention to whether the left or right edge snaps to the grid.



Designing a shape with width and height as integers of the grid spacing

To ensure that shapes snap to the correct position on the grid, masters should use the same units of measure as the drawing page. When you set up the drawing page for your templates, specify the same units of measure as those used for all the masters to be used with that template. You use the Measurement Units option on the Page Properties tab in the Page Setup dialog box for the master drawing window to set units of measure for a master. To set the units for a template, use the Measurement Units option on the Page Properties tab in the Page Setup dialog box.

If you want something other than a shape's edge to snap, you can adjust the alignment box. For details about customizing the alignment box, see "Creating shapes that snap to the grid" later in this chapter.

TIP If you are designing two masters that are likely to be connected, position their connection points so that when the masters are both snapped to the grid and appear to be aligned, the connector will travel a straight path between the two closest connection points. For details about connection points, see Chapter 5, "Making shapes connect: 1-D shapes and glue."

Using formulas to hold grid information

To create masters based on a grid that you may change, you can store the basic grid spacing used for a shape as a formula in a shape or page sheet. For example, you may want to adapt a template and stencil designed for a 1/4-inch grid for use with a different unit of measure, such as centimeters. You can store the grid spacing in a Scratch cell, then define shape width and height in terms of the value of the Scratch cell. Visio doesn't otherwise store the basic grid spacing with a master, so by writing a custom formula you can easily edit the masters in a stencil to work with different grids.

For example, the formulas create a shape for a 1/4-inch grid.

```
Scratch.A1  = 0.25 in.
Width       = 6 * Scratch.A1
Height      = 4 * Scratch.A1
```

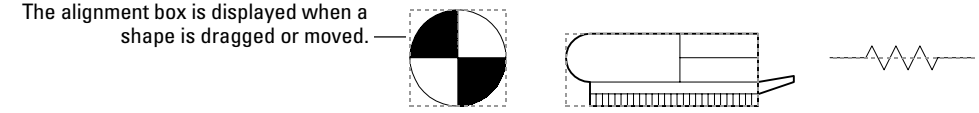
Because the A cells in the Scratch section are unitless cells, you can specify any unit you want in the formula. The value is in drawing units, just like the cells of the Shape Transform section. To make the shape work in a grid based on centimeters, simply edit the value of the Scratch.A1 cell and specify *1 cm* instead of *0.25 in.*

If your template's drawing page uses a fixed grid, you can define the shape formulas in terms of the grid spacing stored in the page's sheet. Instead of storing the grid spacing as a Scratch variable, the width and height formulas refer to the grid information in the page:

```
Width       = 6 * ThePage!XGridSpacing
Height      = 4 * ThePage!YGridSpacing
```

Creating shapes that snap to the grid

When a user drags a shape into the drawing window, Visio snaps the shape's selection rectangle, or alignment box, to the nearest grid line. All shapes have an alignment box, which by default is the same size as the shape's width-height box. If a shape is asymmetrical or composed of odd-sized components, users may find it harder to predict its alignment and snapping behavior. Or you may want parts other than the outer edges of the shape to snap to the grid. You can customize a shape's alignment box to clarify its intended use.

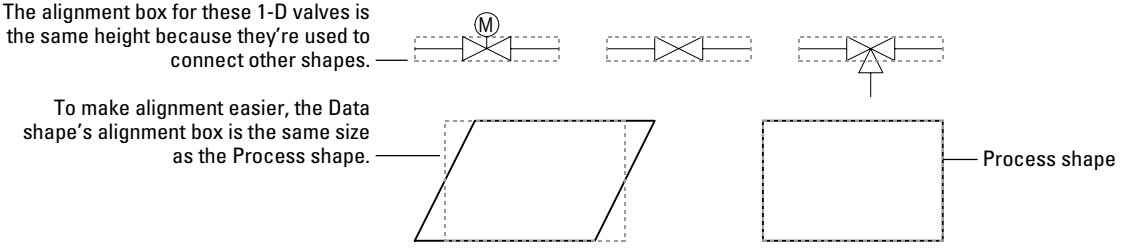


An alignment box may be larger or smaller than the shape it represents.

If a shape is rotated at an angle that is not a multiple of 90 degrees, the alignment box is the smallest upright rectangle that contains all of the paths of the shape as if their line thickness were set to zero.

Adjusting a shape's alignment box

You can customize the size of an alignment box for a shape. For example, you can design a series of different shapes with the same-size alignment box so that they snap and align correctly, as the following figure shows. To do this, you draw the alignment box first, and then prevent Visio from changing it as you create and edit the shape's geometry.



Masters with customized alignment boxes

To define an alignment box that differs from the width-height box:

1. Draw your shape.
2. Select the shape, then from the Window menu, choose Show ShapeSheet.
3. In the Protection section, set the formula for the LockCalcWH cell to 1.

   This setting preserves the current alignment box so that it won't change as you define the shape's geometry.
4. Use the pencil, line, arc, or rectangle tool to add to and modify the shape's geometry.

This custom alignment box is what you'll see as long as the shape's Angle cell is a multiple of 90 degrees.

Enclosing a shape in a larger alignment box

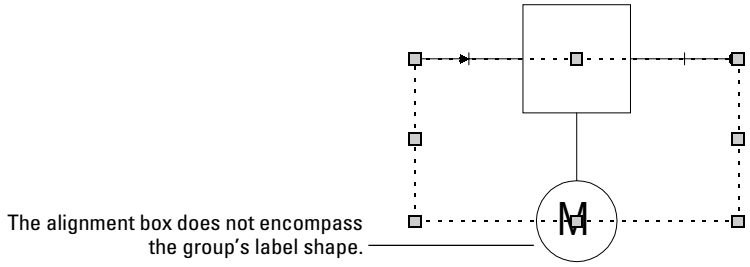
You can enclose a shape in an alignment box that's larger than the width-height box. This can make the shape easier for users to snap to the grid. For example, the symbol for an electrical outlet is a rectangular shape enclosed in a larger, square alignment box to make it easier to position the shape.

To enclose a shape in a larger alignment box:

1. Draw the shape.
2. Draw another shape that is the size you want for the larger alignment box.
3. Select the two shapes, and then press Ctrl+G to group them.
4. Select the group, choose Open Group from the Edit menu, and then delete the alignment box shape from the group.

Customizing a group's alignment box

You can customize the size of a group's alignment box to make your master easier for users to snap and align. When a master is a group of one or more shapes, the group supplies the alignment box. For some shapes, the default group alignment box would not align the shape appropriately. In the following figure, the shape is a group with a custom alignment box.



A custom alignment box that is smaller than the group

To create a group with a custom-size alignment box:

1. Construct the separate shapes that will make up the master. Don't customize formulas for these shapes yet.
2. Use the rectangle tool to create a shape the size and position of the desired alignment box.

3. Select the alignment box shape and group it.
4. Select the group, and then from the Edit menu choose Open Group to open it in the group window.
5. Select all the shapes you want to add to the group and drag them into the group window.
6. Delete the temporary alignment box shape.
7. Add custom formulas to the shapes as desired.

Changing the alignment box for 1-D shapes

By default, a 1-D shape's endpoints are centered horizontally in its alignment box. By moving the begin point and end point within the shape's local coordinate space, you can change the alignment box and make it easier for users to align your shape. For example, the following figure shows a 1-D wall shape with endpoints at the wall's edge, rather than its center. When users drag the shape, the line of the alignment box follows the edge used to connect the wall.

A customized alignment box for a 1-D wall shape

To move the alignment box for a 1-D shape:

1. With the rectangle tool, draw the shape.
2. Select the shape, and then from the Format menu, choose Behavior. Check Line (1-Dimensional), then click OK.
3. From the Window menu, choose Show ShapeSheet.
4. In the Shape Transform section, type *0 in.* in the LocPinY cell.

   Moving the *y*-position of the local pin aligns the endpoints with the shape's edge.

TIP You can hide the alignment box of a 1-D shape such as a connector if displaying it would interfere with the shape's function. Choose Behavior from the Format menu, then uncheck Show Alignment Box. Or set the NoAlignBox property to TRUE in the Miscellaneous section of the ShapeSheet window.

Updating an alignment box

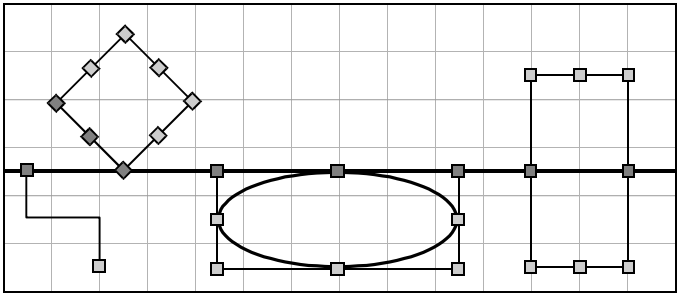
A shape's alignment box will no longer coincide with its width-height box after you edit its vertices or, in a group, after you resize a shape, or add a shape to or delete one from the group. To explicitly realign the alignment box with the width-height box, choose Update Alignment Box from the Tools menu. If you define a control handle at a shape vertex, moving the control handle also changes the shape's geometry so that the alignment box no longer coincides with the width-height box. In this case, you can set the UpdateAlignBox cell in the Miscellaneous section to TRUE so that the alignment box always resizes as the control handle is moved.

Aligning shapes to guides and guide points

When you design a template, you can help your users work more efficiently by including guides or guide points on the drawing page. *Guides* are the nonprinting lines on the drawing page used for alignment, as the following figure shows. A *guide point* is the crossbar-shaped guide dragged from the intersection of the two rulers. Users can then quickly align and move shapes by gluing them to a guide or guide point—when a guide is moved, all shapes glued to it also move.

There are advantages to using guides instead of the grid to align shapes. The grid is always displayed in even intervals. If you want to align shapes to an uneven grid, drag guides to the required positions. (You may then want to disable snapping to grid by choosing Snap & Glue from the Tools menu.) You can also rotate a guide by choosing Size & Position from the Shape menu, but not the grid (which always appears horizontal and vertical with respect to the window, not the page). In addition, a guide or guide point has a ShapeSheet interface, unlike a variable grid, which means you can write formulas to control a guide's position or to automate the alignment of shapes to a guide.

TIP To change the orientation of a guide on a drawing page, right-click the guide, then choose View Horizontal or View Vertical.



You can glue a point, a side, or the middle of a two-dimensional shape to a guide.

Creating guides and guide points

To create a guide, with the mouse, point to either the horizontal or vertical ruler. The pointer changes to a two-headed arrow. Drag to where you want the guide on the drawing page. To create a guide point, drag from the intersection of the two rulers.

To select a guide on the drawing page, click the guide with the pointer tool. The guide turns green. You can then move it, delete it, choose Size & Position from the Shape menu to rotate it, or display it in the ShapeSheet window.

To turn off the display of guides for a document, from the View menu, uncheck Guides. You can also disable snapping to guides. For details, search online help for "snapping shapes into place."

Guides in a rotated page

When you create a guide, it is parallel to the ruler you dragged it from. Rulers are always vertical and horizontal with respect to the window, not the page, so if you create a guide in a rotated page, the guide does not necessarily appear to be rotated with respect to the page you place it on. To specify a guide's angle of rotation, choose Size & Position from the Shape menu.

In a guide, the Guide Info section records the point around which a guide rotates in the PinX and PinY cells and the angle of rotation. (Earlier versions of Visio products do not include these cells but display the GuidePosX, GuidePosY, and Type cells.) A shape that is glued to a guide has an Alignment section, which refers to the guide with a formula that includes the INTERSECTX or INTERSECTY function. (These functions are not available in Visio products previous to version 5.0.)

You can use formulas to place a guide precisely on the page. For example, in a drawing on standard A5 paper at a 1:500 scale, the page width represents 74 meters. You can position the guide with respect to page width with a formula such as:

```
PinX    =ThePage!PageWidth-5m
```

Grouping guides with shapes

You can use guides or guide points to align shapes and groups as you develop masters. For example, you can group a shape and a guide. When you double-click the group, it opens in the group window, where the guide appears. This makes it easy to add shapes to the group.

If a shape is glued to a guide and you add the shape (but not the guide) to a group, Visio breaks the shape's connection to the guide. The reverse is also true: If you add a guide to a group, but don't also add the shapes that are glued to it, Visio breaks the shapes' connections to that guide. If you include both the guide and the shapes that are glued to it in the group, Visio maintains the connections.

9

Packaging stencils and templates

Masters, stencils, and templates make up the package that a graphic solution comes in. Not every solution requires all three, but your solution may if it includes many new or customized shapes and you plan to distribute them to users. In addition, you can include your own help files to assist your users. Before you distribute your masters, stencils, and templates to others, it's important to test them thoroughly. Only by testing can you ensure that every component of your Visio solution is easy for users to understand and use.

This chapter explains how to put the finishing touches on shapes, stencils, and templates. It also describes how to add shape help and includes detailed lists for testing your work based on the method used by the Visio quality assurance staff.

Topics in this chapter

Packaging a shape solution

If you are taking the time to develop your own shapes, you probably plan to reuse them or distribute them in stencils and templates for others to use. The goal of good shape design is to create shapes that work the way users expect them to. Like any creative work, developing shapes is an iterative process that benefits from experimentation and review.

To ensure a professional shape solution, consider following this design process:

1. Make notes about a shape's intended function. What requirements must it satisfy? How must it behave in order to meet those requirements? If the shape will be one of a collection in a stencil, how must it behave to be consistent with other shapes?
2. Draw a prototype of the shape and format it to look the way you want, and then experiment with the shape using the Visio drawing tools. How does the shape behave when you move it? Size it? Rotate it? Group it with other shapes? What happens when you lock parts of the shape? Which behaviors do you want to change?
3. Identify the ShapeSheet cells that influence the behavior you want to change. Which cells need custom formulas, and which cells should the formulas refer to?
4. Create one formula at a time and check its effect on the shape's behavior. Keep notes as you go, either on paper or in text blocks on the drawing that contains your prototype shape. If you're trying different alternatives, you may want to copy the shape each time you try something new and keep the copies so you can return to an earlier version if you need to.
5. Write shape help, so your users will understand the shape's intended function.
6. Test the shape for usability by giving it to coworkers to see if the shape meets their expectations as well as your own.

When you know exactly what you want the shape to look like, how you want it to behave, and what formulas you need to accomplish what you want, re-create the shape from the beginning. This may seem like unnecessary work, but it's the best way to ensure that no obsolete formulas remain in ShapeSheet cells and that the shape itself is drawn and formatted cleanly.

Adding help to masters

You can provide online help that displays general guidelines for using the masters in a stencil or the subtleties of a shape's behavior. This section assumes that you are familiar with the techniques and terminology used in creating Windows online help files. For details, see the documentation that comes with the Microsoft Platform Software Development Kit (SDK) for Windows 95 and Windows NT, versions 3.51 and 4.0.

Associating help with a master

You can associate help with any shape in a drawing, but typically you'll associate help with masters in a stencil. A user displays shape help by choosing the Shape Help command from the Help menu or from the shape's or master's shortcut menu.

Visio locates a shape help topic using the context ID number that is specified in the .HPJ file used to compile the .HLP file. To associate a particular help topic with a shape, you must provide the context ID number for that topic.

To associate help with a master on a stencil:

1. Open the stencil as an original, so you can edit its masters.

   For details about opening stencils, see "Creating masters and stencils" in Chapter 2, "Tools for creating solutions."

2. Double-click a master to open its drawing window, then select the shape.
3. From the Format menu, choose Special.
4. In the Help box, use the following syntax to enter the help file name and keyword:

   ```
   filename.hlp!#n
   ```

   Filename.hlp is the name of your help file, and *n* is the context ID number defined for the topic you want to associate with this shape. For example, *SHAPE.HLP!#63*.

   If you want to display the contents topic of your help file, do not specify a context ID number. Use the syntax:

   ```
   filename.hlp
   ```

5. Click OK.

Tips for writing shape help

Well-written shape help can give your users the key to working successfully with your shapes. At Visio, shape help explains how to use a shape and, for unusual shapes, why you use them.

If you provide help for one shape, you should provide it for all the shapes on the same stencil so that when users choose the Shape Help command, they get a consistent response. However, you don't have to provide a unique help topic for each shape. You can display unique topics for the more complicated shapes and display a general help topic for the rest of the shapes on a stencil.

Here are some tips for developing more effective shape help:

- Keep word count to fewer than 75 words if possible. Shape help is intended to be quick, to-the-point instruction.
- If necessary, use graphics to illustrate what a shape does, rather than more words to describe it.
- Be consistent: Use the same tone, formatting, and look in your help topics. Using a consistent approach makes writing go more quickly and clarifies your intent to users.

When a user chooses the Shape Help command, the indicated topic appears in a pop-up window that is not linked to a parent help system. If you do not define a shape help topic for a shape, the Shape Help command is dimmed on the menu.

NOTE Pressing F1 always displays the Visio online help, not a particular shape topic.

Installing the shape help file

For Visio to find your help file, you must place it in the correct folder. By default, Visio first looks for a shape help file in the default folder for help files (usually the \HELP folder). You can change the default folder by changing the HelpPath setting in the VISIO.INI file. This setting determines the path that appears in the File Paths dialog box, which you can edit for a document by choosing Options from the Tools menu, then clicking the File Paths tab.

If Visio doesn't find the help file you specify in the \HELP folder, it looks in the folder that contains the Visio program files. If Visio cannot find the help file in either folder, it displays the contents topic of the Visio online help.

Testing shape help

Make sure your shape help is as thoughtfully designed as the shape itself. Test the help and its jumps for consistency and accuracy.

To test shape help:

1. Right-click a master on the stencil, or create an instance of the shape, then right-click the instance. Choose Shape Help and check that the correct help topic appears.
2. Create another instance of the shape, point to the instance, and click the right mouse button. Choose Shape Help from the pop-up menu and check to be sure that the correct help topic appears.
3. Test all jumps to make sure they display the correct topics.
4. Check each topic for spelling, grammar, consistency, and accuracy of its content.

Developing solutions for different systems

What works on your system may not work as well on someone else's. Not all Windows installations are exactly alike. You can design more usable shapes, stencils, and templates for others to use if you know your users' hardware configurations. Even if you create shapes only for your own use, knowing the characteristics of your computer environment will save time by helping you create shapes that work the first time.

On any given system, the speed of the processor, the amount of memory, and the availability of a math coprocessor affects the usability of your stencils and templates. Shapes with many complex formulas recalculate and redraw more slowly than simple shapes, and they take up more disk space. Be sure to test your stencils on all the systems your users may have, including portable computers.

Enhancing shape performance

You can design shapes that perform more efficiently on different systems by using some of the following techniques:

- Combine, rather than group, parts of a shape.
- If you must use a group, keep the number and level of groups to a minimum to reduce the number of ShapeSheet interfaces, too many of which can slow performance and add to the file size.
- Group or combine component shapes that make up one master.
- Convert shapes with complex graphic detail, such as clip art, to a Windows metafile (.WMF). This conversion can reduce the file size and make the shapes appear faster.
- Convert objects from other applications from which you want to create masters into Visio shapes. Their performance as masters will be more reliable.
- Format shapes using styles rather than local formatting so the shapes take up less disk space and respond faster to user actions. Also, if possible, use fewer colors and fill patterns.
- Minimize the number of interdependencies between ShapeSheet formulas you write.

Designing for different video systems

When designing your stencils and templates for distribution, take into account the color capabilities and resolutions of different video systems. If you design for the system with the lowest resolution and fewest colors, chances are that your layouts and shapes will appear even better on more sophisticated systems. However, a stencil designed for higher resolution or more colors probably won't look as good on a less sophisticated system.

The color capabilities of a video system may determine how you use color in your shapes. For example, some video systems have difficulty displaying dithered colors or colored patterned lines, which Visio uses to differentiate different types of shapes.

If you develop stencils and templates on a system with higher resolution, you have more screen area in which to arrange icons in stencils and more space for master prompts. In addition, Visio can display more buttons on the toolbar at higher screen resolutions. Because toolbar buttons are measured in pixels, a button appears smaller at a higher resolution and more buttons can fit on the toolbar. The VISIO.INI file includes settings that can help if you are designing stencils for systems other than standard VGA. For details, see VISINI.TXT in the \DVS folder.

Designing shapes that print well

To design shapes that your users can print, you need to know the capabilities and limitations of their output devices. Do your users have a vector-based device, such as a pen plotter, or a raster-based one, such as a laser printer? Does the output device support monochrome, grayscale, or color? What is its resolution? Do you have to support more than one output device?

Some output devices can't print all of the fills that Visio provides, and others have difficulty printing shapes with multiple Geometry sections. You should test your shapes by printing them on the output device you expect your users to have to make sure the lines and fills look the way you want.

Testing masters

You should test all the masters on a stencil together for consistency, and then test each master individually. After performing the following tests, spend a few minutes to construct the kind of diagram or chart the shapes are intended to produce. This is the best way to evaluate their interaction, accuracy, and usefulness and to discover limitations or missing elements.

Checking the consistency of masters

You need to ensure that a stencil contains all the masters it should, that the names and formats are understandable, and that the icons appear in a predictable order on the stencil. If you have a written specification for master standards, be sure to check each shape against the specification.

To check the consistency of masters on a stencil, open the stencil file as Original, and then verify the following:

- The expected number of masters are on the stencil. Verify this number against the specification, if you have one. If the stencil is later modified and you test it again, you will know whether masters have been added or removed.
- The master name and prompt have correct spelling, punctuation, capitalization, grammar, content, and spacing, and they are consistent with other shapes.

- No trailing spaces exist in the master name that would cause highlighting to extend farther than necessary when the icon is selected. To check, choose Select All from the Edit menu.
- Names are aligned in the same way for each master on the stencil.
- Icons are arranged logically, aligned consistently, and appear in order from left to right, top to bottom.
- Icons are set to the correct size. Normal is the most commonly used setting.
- Each icon is a meaningful representation of its master. Visually inspect each icon for clarity, and compare the icon to the master itself. To check, select the icon and then choose Update Icon from the Master menu. You'll see a miniature of the master on the icon. Then choose Undo to see the icon again.
- Icons with a custom graphic are set to update manually. To check, choose Properties from the Master menu.

Checking the master in the master drawing window

To test a shape in the master drawing window, open the stencil file as Original. In the stencil window, double-click a master icon to open it in the master drawing window, and then verify the following:

- The shape uses the appropriate scale. To check this, choose Drawing Page from the Edit menu, then choose Master Properties.
- The shape is 1-D or 2-D, as appropriate. To check, choose Behavior from the Format menu.
- The information about the master that appears in the Special dialog box is correct. For example, the Data fields are filled out, and the shape is linked to shape-specific help. To check, choose Special from the Format menu.
- The appropriate Protection options are set. If you aren't sure, display the master's ShapeSheet window or the Protection dialog box to verify the Protection settings.
- Connection points are visible.
- The shape is the expected size. This is important if the shape must work with other shapes or if the name or prompt indicates that the shape is a specific size.

Testing the master scale

Because a stencil and a drawing page are opened with each template you provide, you should test each shape on all of the different page scales that it is intended to work with. It's also helpful to test a shape on a page with a very different scale.

To test a shape in a drawing of the same scale:

1. From the File menu, choose Open.
2. Under File Name, select a template file containing a stencil with masters to test and a drawing page that uses the same scale as the masters.
3. Under Open, select Read Only, and then click OK.
4. Drag a master onto the drawing page to create the instance to test.
5. Verify the following:

- The shape is aligned appropriately within its alignment box as you drag it. To see this, pause during dragging until you can see a representation of the shape.
- The shape and the alignment box snap to the grid.
- The prompt and shape-specific help provide useful information about the shape.
- The instance snaps to other shapes and to the grid or guides as expected.
- The shape's text box appears in the correct place, and text you type in it wraps and aligns appropriately.
- The shape and its text act as you expect when you apply a fill style.
- The shape and its text act as you expect when you resize the shape vertically, horizontally, and proportionately. This test is particularly important if you have programmed the shape to resize in a unique way—for example, in only one direction.
- The shape and its text act as you expect when you rotate the shape using the Rotate Left and Rotate Right commands and the rotation tool.
- The shape and its text act as you expect when you reverse ends and flip the shape vertically and horizontally.

- The Special dialog box contains appropriate information. To check, choose Special from the Format menu.
- The shape behaves as expected when connected to other shapes. For example, a connector shape uses the appropriate glue type.
- The shape acts as you expect when you double-click it. You may also want to check the setting by choosing the Double-Click command from the Format menu.
- The shape and its text act as you expect when you ungroup the shape. If the master is not a group, the Ungroup command is dimmed.
- The shape looks the way you expect it to when you print it on both PCL and PostScript printers. Some fill patterns affect performance on PostScript printers.
- If the shape has customized actions on its shortcut menu, they work as intended. To check, right-click the shape, then choose the Action command.
- If the shape has custom properties, they appear as expected. To check, choose Custom Properties from the Shape menu.
- The shape can be deleted.

To test a shape in a drawing of a different scale:

1. Create a new drawing page with a much different scale than the shape you want to test.

   For example, if the master was created at a scale of 1:1, create a drawing page with a scale of 1/4 inch = 1 foot.

2. Drag a master onto the drawing page to create the instance to test.

3. Verify the following:

- The shape is aligned appropriately within its alignment box as you drag it. To see this, pause during dragging until you can see a representation of the shape.
- The shape and the alignment box snap to the grid.
- The shape and its text act as you expect when you resize the shape vertically, horizontally, and proportionately.
- The shape and its text act as you expect when you rotate the shape using the Rotate Left and Rotate Right commands and the rotation tool.

Finishing and testing a stencil

A standalone stencil is a Visio file with the extension .VSS that contains the masters users can drag into a drawing window. Masters are represented in a stencil by icons that you can design. In general, when you want to prepare a stencil for distribution, you:

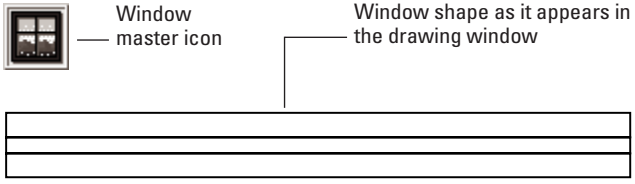
1. Create or open a stencil file.
2. Save shapes as masters on the stencil.
3. Name the masters.
4. Create the master icons.
5. Clean up the file to optimize performance.
6. Test the stencil.

This section describes the final tasks: how to add master names and icons, optimize the stencil file, and then test it. For details about opening stencil files and adding masters, see "Creating masters and stencils" in Chapter 2, "Tools for creating solutions."

Cleaning up masters in a stencil

The stencils you create will be easier to use if the masters look as if they belong together and each conveys the corresponding shape's purpose. You can edit the master name and icon to make your masters easier for users to identify. You can also add a prompt that appears in the Visio status bar to explain the master's purpose.

By default, a master's name is the identifier that Visio assigns, and its icon is a miniature version of the master. When you edit a new master, the icon is updated to reflect the shape you draw unless you specify otherwise.



To help users identify your master, you can design a custom image for its icon.

About master icons

At Visio, we standardize the border of the icons in our stencils and use color to indicate a shape with special attributes. Icons with yellow backgrounds are usually 1-D masters that work well as connectors. Red crosses indicate connection points. Light gray or blue-green backgrounds usually indicate 2-D shapes. Future stencil sets might use different standards, but icons in a stencil set will always have the same "look."

In addition, we avoid adding words to icons or shapes. Because we ship our stencils to many countries, any text in an icon or shape would need to be translated, an expensive and time-consuming task. Instead, we try to use generic text like "ABC" instead of words when text is needed.

To specify a master name and prompt:

1. In the stencil window, right-click a master, then choose Properties from the shortcut menu.

   The stencil must be opened as Original.

2. Under Master Name, type a name for the master.

   If you want the master name to be aligned beneath the icon in some fashion other than centered, select an Align option.

3. Under Prompt, type the text you want to appear in the status bar when the user points to the icon.

4. Under Icon, select the properties you want, then click OK.

To create a custom master icon:

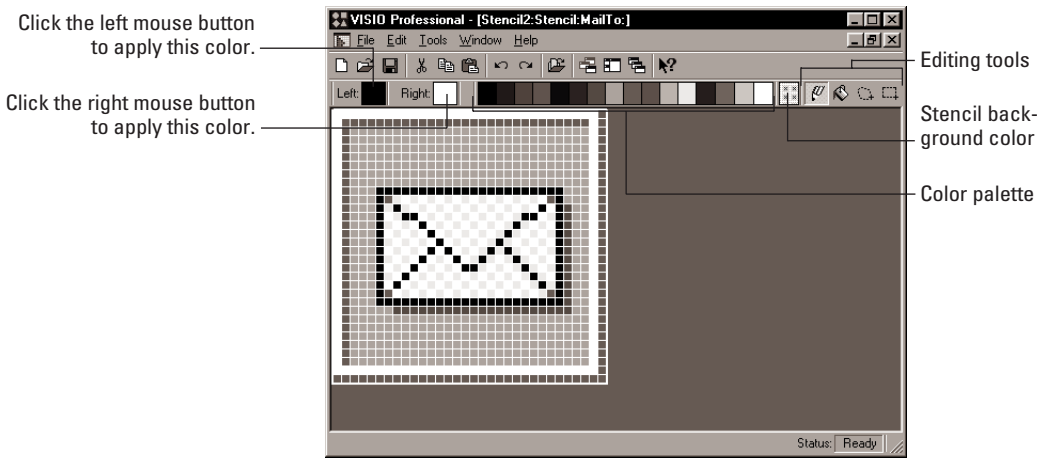
1. In the stencil window, right-click a master, then choose Edit Icon.

2. Use the drawing tools in the icon editing window to edit the icon or create a new design.

   For details about using the drawing tools in the icon editing window, search online help for "edit icon window."

4. When you are finished, close the icon editing window.

5. To protect your new icon from accidental changes, choose Properties from the Master menu. Under Icon, select Manual for Update.



The icon editing window

Cleaning up a stencil file

Before you save a finished stencil, you should perform the following cleanup tasks to enhance performance:

- Arrange the icons in the stencil windows to ensure that they appear onscreen in order from left to right, top to bottom, when the file is opened.
- Include file summary information for the stencil. To do so, make the stencil window active, then choose Properties from the File menu.
- To save file space, make sure your stencil file contains only the required single drawing page and that there are no shapes on it.
- Delete any styles from the drawing page that are not used by the masters in the stencil. A stencil file should contain only masters and their styles.
- Verify that the style definitions in a stencil match those for styles of the same name in any templates that open the stencil. For details, see Chapter 7, "Managing styles, formats, and colors."
- Use the Save As command to save your stencil file, and make sure that Workspace is unchecked in the dialog box. A stencil's workspace list should be empty.

Making your stencils easy to use

Stencil users expect the shapes in a stencil to work together. They expect that masters, stencils, and templates will behave in similar ways, and that these items will be presented in a consistent fashion. Inexperienced and nontechnical software users infer a great deal from the differences in spacing and tone of the text that they see in a software product.

Much of what gives a user the impression of consistency is attention to detail. Details that might seem trivial, such as capitalization of master names, prompts, and file summary information, can often make a big difference to the people using your shapes. Like good shape design, many of these details are felt rather than consciously noticed by the user.

You can make the masters in your stencils more consistent and easier to use if you:

- Use conventions for master names.
- Include master prompts.
- Carefully name stencils and templates to help users identify them.
- Fill in file properties for stencils and templates.
- Use the same style names in both the drawing file and stencils associated with a template—but make sure that the styles' definitions are the same.
- Ensure that shapes in your stencil are compatible with your template's grid.
- Make sure that the drawing scales of templates you provide match the scales used in your masters.
- Standardize the appearance of your stencil's master icons to indicate a master's use and behavior.

Testing stencils

You test stencils by reviewing the Open dialog box information and by reviewing the stencil opened as Original, Copy, and Read Only.

NOTE To protect your original stencil, create a copy that contains the shapes you want to test, then use the copy for testing. After you test, incorporate changes in the original stencil, and then make a new copy for additional testing.

To test the information in the Open dialog box:

1. From the File menu, choose Open.
2. Under File Name, select a stencil file.
3. Verify the following:

- The default Open setting is Read Only.
- Under Description, a title and description should appear. If they don't, be sure to add these later to the original file using the Properties command on the File menu.

To test the original version of a stencil, open the stencil file as Original, and then verify the following:

- The file opens with its name displayed correctly in the title bar. For example, the name should look like this: BASIC.VSS.
- The stencil window occupies the left quarter of the screen.
- File property information is filled out. To check, choose Properties from the File menu and verify the spelling, grammar, content, spacing, capitalization, and punctuation.

To test a copy of a stencil, open a stencil file as a Copy, and then verify the following:

- The file opens with a generic name, such as STENCIL1.
- File property information is blank except for the Author box, which displays the user name specified in the Options dialog box (Tools menu) or when Visio was installed on the computer.

To test the read-only version of a stencil, close all other files, open the stencil file as Read Only, and then verify the following:

- The stencil opens in a docked window.
- The file name in the title bar appears in braces.
- On the File menu, the Save command is dimmed.
- On the Edit menu, the Cut, Clear, Paste, and Duplicate commands are dimmed.
- On the Master menu, all commands are dimmed.

If your stencil includes online help for shapes, test the help. For details, see "Testing shape help" earlier in this chapter.

Finishing and testing a template

A template is a convenient way to open a stencil and store macros, styles, a color palette, and page properties. When a user creates a new document that is based on your template, the stencil and drawing windows appear exactly as you specify. If the template's drawing page has standard elements, such as a title block or border, those elements appear in an untitled drawing.

To create a template, you can open a new .VST file or save an existing drawing, stencil, or template as a .VST file. A template can include:

- One or more stencil files, which are not stored in the template but are opened when you open a new drawing file with the template. Each stencil must be named and saved as a .VSS file before you can save the template.
- One or more drawing pages, including backgrounds. Each page can contain a drawing that uses a unique size and scale.
- Additional drawing (.VSD) files, which open as originals when the template is opened. Each drawing file must be named and saved as a .VSD file before you can save the template.
- Print settings.
- Styles for line, text, and fill.
- Snap and glue settings.
- A color palette.
- VBA modules, class modules, and user forms.
- A workspace list with information about the size and position of each open window.

For details about opening a file as a template, see "Creating templates" in Chapter 2, "Tools for creating solutions."

Cleaning up a template

When you save a template, you should ensure that the workspace list contains only the files you want to be opened, that all the windows are in appropriate positions, and that any window you want to be minimized is minimized. You create a workspace for a template by checking the Workspace box in the Save As dialog box, then saving the template file. After that, unless you uncheck the Workspace box, Visio updates a template's workspace list each time the original file is saved—adding files that happen to be open and eliminating files that happen to be closed.

Before saving the template for distribution, clean up the windows and workspace as follows:

- Delete unnecessary masters from the template's local stencil so that users' files don't become any larger than they have to be. To do this, activate the drawing window, then choose Show Master Shapes from the Window menu. Delete only those instances that are not present in any of the drawings in the drawing file.
- Include summary information for the template.
- Make sure the size of windows and stencils looks good on different systems. To do this, open the template on a system with the display resolution your users are most likely to have and use the Tile command to help position windows correctly. Be sure to open the template on systems with different display resolutions to ensure that the window positions still work.
- Make sure the template's color palette matches that of any stencils that open with the template. For details, see "Managing color in styles, shapes, and files" in Chapter 7, "Managing styles, formats, and colors."
- Verify that the style definitions in the template's drawing page match the definitions for styles of the same name in the stencils. For details, see "Using styles in stencils and templates" in Chapter 7, "Managing styles, formats, and colors."

If you create a template by saving an existing Visio file as a new .VST file, the new template may inherit an irrelevant workspace list. Be sure to test your template to make sure its workspace list opens the files and windows you want before you release the template to users.

Testing templates

To test a template, you need to verify the information about the template that appears in the Open dialog box, and then test how the template acts when it is opened as Original, Copy, or Read Only.

NOTE To protect your original template, create a copy that contains the shapes you want to test, then use the copy for testing. After you test, incorporate changes in the original template, and then make a new copy for additional testing.

To test the information in the Open dialog box:

1. From the File menu, choose Open.
2. Under File Name, select a template file.
3. Verify that a title and description appear under Description.

   If they don't, be sure to add these later to the original file using the Properties command on the File menu.

To test the original version of a template, open the template file as an original, and then verify the following:

- The file opens with its name displayed correctly in the title bar. For example, if the template file is ORGANIZATION CHART.VST, the name in the title bar should look like this:

  Organization Chart.vst

- All stencil (.VSS) files associated with the template open as read-only, unless intended to open as original files.
- The drawing page window opens in Whole Page view, unless you explicitly specify another option. (Whole Page view is the best option for most monitors.)
- The stencil and drawing windows are positioned correctly. Choosing Tile from the Window menu verifies their positions, unless you have already repositioned the windows during the current work session.
- The template includes the correct number of pages. To check, from the Edit menu, choose Go To, then choose Page to display the Page dialog box. Templates should have only one page unless you have intentionally created additional pages.
- The content of each page (including each background) is correct.

- Nothing unintentional appears on the pasteboard (the blue area outside the drawing page). To check, display each page at 5% magnification. To ensure all the shapes are visible, choose Select All from the Edit menu.
- Each page scale is compatible with the shapes intended for use with the template. To check, for each page, choose Page Setup from the File menu, then click the Page Properties tab.
- The page size corresponds to the page orientation used for printing. Unless you specifically want pages to tile when they are printed, the settings should correspond as follows: If the page size is taller than it is wide, the orientation should be portrait. If the page size is wider than it is tall, the orientation should be landscape.
- No masters remain on the local stencil, unless you have created a form on the template's drawing page, in which case no other masters should appear. To check, choose Show Master Shapes from the Window menu.
- File property information is filled out. To check, choose Properties from the File menu and verify the spelling, grammar, content, spacing, capitalization, and punctuation.
- The template settings for each page are as expected. To check, from the Tools menu, choose the Options, Snap & Glue, and Ruler & Grid commands. From the File menu, choose Page Setup, and then click the Page Size, Drawing Scale, and Page Properties tabs. Check the style lists on the toolbar.
- The template display options are set appropriately: rulers, grid, guides, connection points, toolbar, and status bar.

To test a copy of a template, open the template file as a copy, and then verify the following:

- The file opens with a drawing page name that looks like this: Drawing1:Page 1. Verify that the drawing page and any pages you have added look the way you expect them to.
- All stencil files (.VSS) associated with the template open as read-only.
- File properties are blank except for the Author box, which displays the user name specified in the Options dialog box (Tools menu) or when Visio was installed on the computer.

To test the read-only version of a template, open the template file as read-only, and then verify the following:

- The file name in the drawing window title bar appears in braces and starts with the template name.
- On the File menu, the Save command is dimmed.

Installing stencils and templates

For Visio to find your stencil and template files, as well as any add-ons intended to work with them, place them in the \SOLUTIONS folder. You can add folders to this folder so that your solutions appear in the File menu when a user chooses New or Stencils.

If you want to install your stencil and template files elsewhere, you can change the default folder where Visio searches for files. To do this, choose Options from the Tools menu, then click the File Paths tab. In the File Paths dialog box, you can specify the default path you want.

The file name for each stencil and drawing that opens with a template is stored in the template's workspace list as a fully qualified path and name. Problems can arise when files are moved to different machines where local or network drives are configured differently. To prevent some of these problems, Visio checks the path as follows:

1. When Visio is about to open a file from the workspace list, it first examines the file's stored path.
2. If the path is exactly the same as the stored path for the file that contains the workspace list, Visio assumes that these files were meant to be in the same folder.
3. Visio looks in the current folder of the workspace file for the other file.

As long as you copy stencils and templates to the same folder when you must move files, Visio can locate and open all the files in a workspace list.

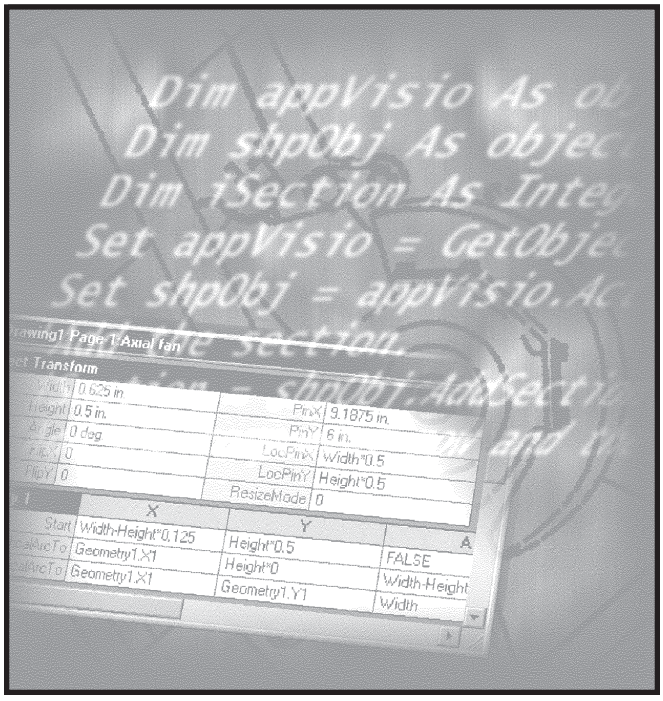
Protecting stencils and templates

The easiest way to protect stencils and templates from accidental changes is to make the files read-only. If a stencil or template is read-only, modifications cannot be saved to the file. When you create a template, open the stencil files you want to include as read-only, and then save the template. That way, Visio automatically opens the stencils as read-only when a user opens the template.

When you use the Save or Save As command, you can check the Read-Only option to save a Visio file with Windows read-only protection. If you save a file in this way, your users cannot open it as an original, only as a copy.

Another way to protect a document is to use the Protect Document command on the Tools menu. This command prevents a user from changing any background pages in a template, all masters in a stencil, all shapes on the drawing, and all styles in the template. If you enter a password, a user must type the password before editing any of the checked items. For details about the Protect Document command, search online help for "protect document command."

PART III



Extending Visio with Automation

10

Automation and Visio

By now you're familiar with Visio and the many possible uses of SmartShapes symbols. You know how to design and build intelligent shapes that can be used repeatedly to create complex and sophisticated drawings. But perhaps you need to create or update a series of drawings based on data that changes from day to day, or perhaps you find yourself performing routine shape development tasks over and over. Perhaps you may support a group of users who need to create drawings but don't need or want to become Visio experts, or you may use their drawings as a means of collecting information. You can automate all of these tasks by using Automation to integrate the graphics functionality of Visio with programs you write in Visual Basic for Applications (VBA), Visual Basic, C/C++, or other programming languages that support Automation.

This chapter introduces Automation and describes how a program can use the objects exposed by an application such as Visio. It also provides guidelines for planning an Automation solution that uses Visio and discusses the methods for controlling Visio, such as VBA macros written in Visio or external programs written in Visual Basic or C/C++.

Topics in this chapter

What is Automation?

Automation (formerly OLE Automation) is a means by which a program written in VBA, Visual Basic, C/C++, or other programming languages that support Automation can incorporate the functionality of an application such as Visio, simply by using its objects.

If you're familiar with Visual Basic, you use objects all the time—controls such as command buttons, user forms, databases, and fields. With Automation, you can use other applications' objects as well—which means you can use Visio drawings, masters, shapes, and other Visio objects as components of your program.

Automation is like a common scripting language among applications that support it. However, Automation takes a different approach to controlling an application. Typically, a scripting language simply automates the same actions you would perform in an application's user interface—choosing menu commands, pressing keys, typing, and so forth.

With Automation, instead of programming an application's actions, you use its objects. An object encapsulates data, behavior, and events with an interface that allows you to access the data, behavior, and events. Each Visio object has properties (data), methods (behavior), and events that you can use to take advantage of that object's capabilities in your program. Visio objects reside in an instance of Visio—a VBA program runs within an instance of Visio and then accesses the objects it needs. An external program runs outside an instance of Visio, so it starts Visio or accesses an instance of Visio that is already running. Then it accesses the Visio objects it needs.

In Automation, the application that provides the objects—typically called the *provider application* or *Automation server*—makes the objects accessible to other applications and provides the properties and methods that control them. (This is sometimes called *exposing* the objects.)

The application (such as your program) that uses the objects—typically called the *controller application*—creates instances of the objects and then sets their properties or invokes their methods to make the objects serve the application. The provider application and controller application interact by making function calls through the OLE libraries, which are installed when any application that supports OLE—such as Visio, Visual Basic, or Windows—is installed.

Planning an Automation solution with Visio

Typically, you'll use Automation either to extend the functionality of Visio or to include Visio as a graphics engine for your own programs. The approach you take in designing your solution will depend on its purpose and the context in which it will be run. You may use Visio to create or update drawings based on data gathered elsewhere—either from user input or from a database—or you may read drawings and gather information from them. Or you may simply extend the behavior of a shape with a VBA macro.

The first step in designing a Visio solution is to decide on the division of labor—what Visio objects should do versus what the program should do.

Start by building the shapes and putting as much of the functionality as possible in ShapeSheet formulas. The most important thing to remember is that *shapes can be smart*—you can use the intrinsic capabilities of Visio shapes to handle much of the graphic functionality that you'd otherwise have to code.

Another important thing to remember is that *shapes are independent of your program*. Once you develop the masters your program will use, you can change the shapes without having to recompile your program, and vice versa.

Starting with smart shapes

If the shape behavior you want is predictable and can be accomplished with formulas, such as sizing or scaling, put it in the shape. If the behavior changes dynamically at runtime—for example, the text in a shape or the arrangement of shapes in a drawing may change—handle that in the program. You can control the appearance and behavior of shapes with great precision by setting shape formulas. If you can create a stencil of masters to accompany your program, you may not need to draw at all.

As you build masters for your program, test them in Visio by creating the kinds of drawings you intend your program to create. This will give you a good idea of the procedures you'll need to code in the program and the data you'll need to provide. It will also show you if your shape is working the way you expect.

Using a smart shape: an example

Suppose you want to draw a property line based on surveyor's measurements entered in a dialog box. The line should have the appropriate symbols and display the length of each leg. You can put most of this functionality in a property line master. All your program needs to do is prompt for the measurements, calculate where each leg of the property line should go, and drop the property line master in the drawing at the appropriate locations.

Providing a template

If your program is designed to create new Visio drawings, you can save both programming effort and execution time by providing a Visio template with your program or by storing your program as a VBA macro in a template. A template can include styles and set up drawing pages using a uniform grid and measurement system. A template can provide shapes already on the drawing page and open one or more stencils. For details about creating stencils and templates, see Chapter 9, "Packaging stencils and templates."

A template can also provide drawings with their own user interface by including ActiveX controls such as command buttons and text boxes, or custom controls that perform special tasks, with VBA code that allows a user to interact with the drawing through the controls. For details, see Chapter 18, "Using ActiveX controls in a Visio solution."

When a template is used to create a document, Visio copies the template's styles, document properties, and VBA macros, modules, and user forms to the new document. You don't need to set the document properties or define styles from the program unless you want them to be different from the template, nor do you need to separately distribute a VBA program. For details about how to include VBA macros, modules, and user forms in a template, see Chapter 2, "Tools for creating solutions."

NOTE Using a template can also prevent some translation difficulties if your program refers to styles and will be used with multiple languages.

Handling the rest in the program

Once you've developed the master shapes and template (if your program needs one), you can handle the rest of the functionality in your program. Exactly what this entails will depend to a great extent on the purpose of the program and the context in which it will be run. However, your program will typically handle the following:

Implementing the user interface. Most standalone programs will need a dialog box, data entry form, or wizard screen to advise the user what to do and prompt for any information the program needs to execute.

Providing a template: an example

Suppose you're writing a program to create scaled office plans. You can provide a template such as OFFICE.VST, which includes a scaled drawing page and opens one or more stencils with office shapes. When you use the template to create a document, the scale is copied to new pages in the document. To create drawings, your program or users can drop master shapes from the stencils.

Storing and retrieving data. Shapes can have custom properties, which can be configured to prompt the user to enter data or shape properties when, for example, a master is dropped on the drawing page. However, to ensure correct types and protect data from unplanned changes in Visio, you may want to handle data entry, storage, and retrieval in your program using an external database.

Placing shapes, setting their properties, or connecting them. If your program creates a drawing, it will need to determine which masters to drop and where to drop them, set the shapes' text and apply styles, and connect shapes. If your program reads drawings or works with existing shapes, it will need to find the shapes, make sure they're appropriate for the program, and get and set shape properties and formulas.

Remember that a shape can have formulas that resize or reorient it appropriately when your program moves or resizes it—just as if you moved or resized the shape yourself, using the mouse in Visio. If you find yourself writing a lot of complex code that manipulates shapes, take a step back and think about whether that functionality can be handled by shape formulas.

VBA, Visual Basic, and C++

The Visio version 5.0 product line integrates VBA, a complete development environment, into the Visio graphics environment. This integration makes it easier than ever before to develop custom business solutions with Visio by eliminating the need for a separate development tool—all programming can be done within Visio and written in the Visual Basic programming language.

Because these Visio products include the VBA development environment, the main focus of this book is on programming Visio with VBA. Most example code in this book is written for the VBA developer. For information about the VBA sample code on your Visio 5.0 CD, see Chapter 11, "Using Visio objects."

For information about how to use the VBA development environment within Visio, see Chapter 2, "Tools for creating solutions."

For information about controlling Visio from a Visual Basic program and the Visual Basic sample code on your Visio 5.0 CD, see Chapter 19, "Programming Visio with Visual Basic."

For information about controlling Visio from a C++ program, the C++ sample code and functions on your Visio 5.0 CD, and about writing a Visio library (VSL), see Chapter 20, "Programming Visio with C++."

Deciding upon a program

The kind of program you write depends on what you're trying to do. You may write a VBA macro in Visio or another Automation controller application, or a standalone program in Visual Basic or C/C++. You may write a special kind of dynamic-link library that runs with Visio, called a Visio library (.VSL). Users may run your program from Windows or from Visio, by choosing a command added to a Visio menu, a button added to its toolbar, or even by double-clicking or right-clicking a shape in a drawing. Or you may write a program that runs when a certain event happens, such as when a document is opened or created.

VBA, Visual Basic, or C/C++? If you're using your program to extend the Visio functionality and want your program to run within a Visio instance, you might write a VBA macro in Visio. If Visio is the main component in your solution, you might write a VBA macro in Visio that controls Visio and other applications such as Microsoft Excel. If you want to run your program from the Windows desktop or Windows Explorer, you might write a standalone program in Visual Basic or C/C++ that controls Visio and other applications that support Automation.

Visual Basic hides many of the details involved in interfacing with Automation, so it's a lot easier to write Automation programs in VBA or Visual Basic than in C/C++. It also takes less code to write VBA macros than programs in Visual Basic or C/C++. However, C/C++ may be a better choice for some programs. For example, Visual Basic allows you to create executable programs (.EXE), but if you want to write a Visio library (.VSL), you must use C/C++.

VBA macro, add-on, or standalone program? If your program extends Visio by adding functionality or creating special shapes, you'll probably want to design it as a VBA macro or add-on. A VBA macro is a procedure that takes no arguments and is contained within a module within a project stored in a Visio template, stencil, or drawing. All code is stored in the Visio file which, eliminates the need to distribute a program separately—just distribute the Visio file. VBA macros are faster than .EXE programs that use the same code because VBA macros run within the Visio task space. Keep in mind that storing VBA macros in a Visio file increases file size. For information about locking a VBA project so users can't inadvertently change the code, see Chapter 2, "Tools for creating solutions."

Migrating to VBA from Visual Basic

Because earlier versions of Visio do not include the VBA development environment, VBA macros written in Visio 4.5 and later are not backward compatible. If you open a Visio 4.5 or later document in Visio 4.0, the VBA macros are not visible—they are still stored in the file, even if you save the file in Visio 4.0, but you cannot access them in Visio 4.0.

If you are a Visual Basic developer who is thinking about migrating to VBA from Visual Basic, see "Migrating from Visual Basic to VBA, in Chapter 19, "Programming Visio with Visual Basic," for details about issues involved in making the transition.

An add-on is similar to a standalone program, except that its .EXE or .VSL file is installed in the Visio add-ons folder so that the program's name appears in the Macro submenu. An add-on is typically not intended to be run from the Windows desktop or Windows Explorer.

To run a VBA macro, add-on, or standalone program, you can associate a program with a menu command or toolbar button that you add to Visio, define an action for a shape's shortcut menu or double-click event, or respond to a document event, such as inserting pages or deleting shapes. For example, the Chart Shape Wizard handles a shape's double-click event by running itself so the user can edit the shape. For details about programming events, see Chapter 4, "Enhancing shape behavior," and Chapter 15, "Handling events in Visio." For details about adding menus and toolbar buttons, see Chapter 16, "Customizing the Visio user interface."

If your program uses Visio as a component in a solution that uses other software, you may want to create a standalone executable program that can be run from Windows Explorer. For example, Network Diagrammer uses Visio to create a network diagram but doesn't require Visio to be running until it actually starts drawing the diagram. A standalone program must be an .EXE file. It may also be easier to fix bugs in one standalone program or add-on than to fix bugs in a VBA macro that may have been copied into multiple documents.

Executable program or Visio library? An executable program (.EXE) can access and control Visio objects, but it can't prevent the user from taking independent action with either Visio or the program. You can increase your program's control and improve its performance by writing the program as a .VSL, a Windows dynamic-link library that implements a prescribed protocol for interacting with Visio. In addition, an .EXE file runs in a separate task space within Windows, which means that there's more overhead involved in calling a Visio method or property than there is in calling the method or property from a Visio library (.VSL) loaded into the Visio task space. This can affect the program's performance.

VBA macros, executable programs, and Visio libraries: a summary

Type of program	Definition	Reasons to use
VBA macro (no separate file)	A procedure that takes no arguments; written in the VBA development environment; macros are stored within modules stored within projects; projects are a collection of modules, class modules, and user forms; every Visio file has a default project	• Easy to code; less code than standalone programs • Fast; runs within the Visio task space • Easy to distribute; stored in Visio files • Integrated; you don't need a separate development tool
Executable program (.EXE)	A program that runs in its own process; written in a programming language that supports Automation	• Flexible; accessible from the Windows desktop or Windows Explorer; can operate as a standalone program or as a Visio add-on • Separate; the program needs to run Visio only when necessary
Visio library (.VSL)	A dynamic-link library specifically written for Visio; written in C/C++	• Fast; runs within the Visio task space

11

Using Visio objects

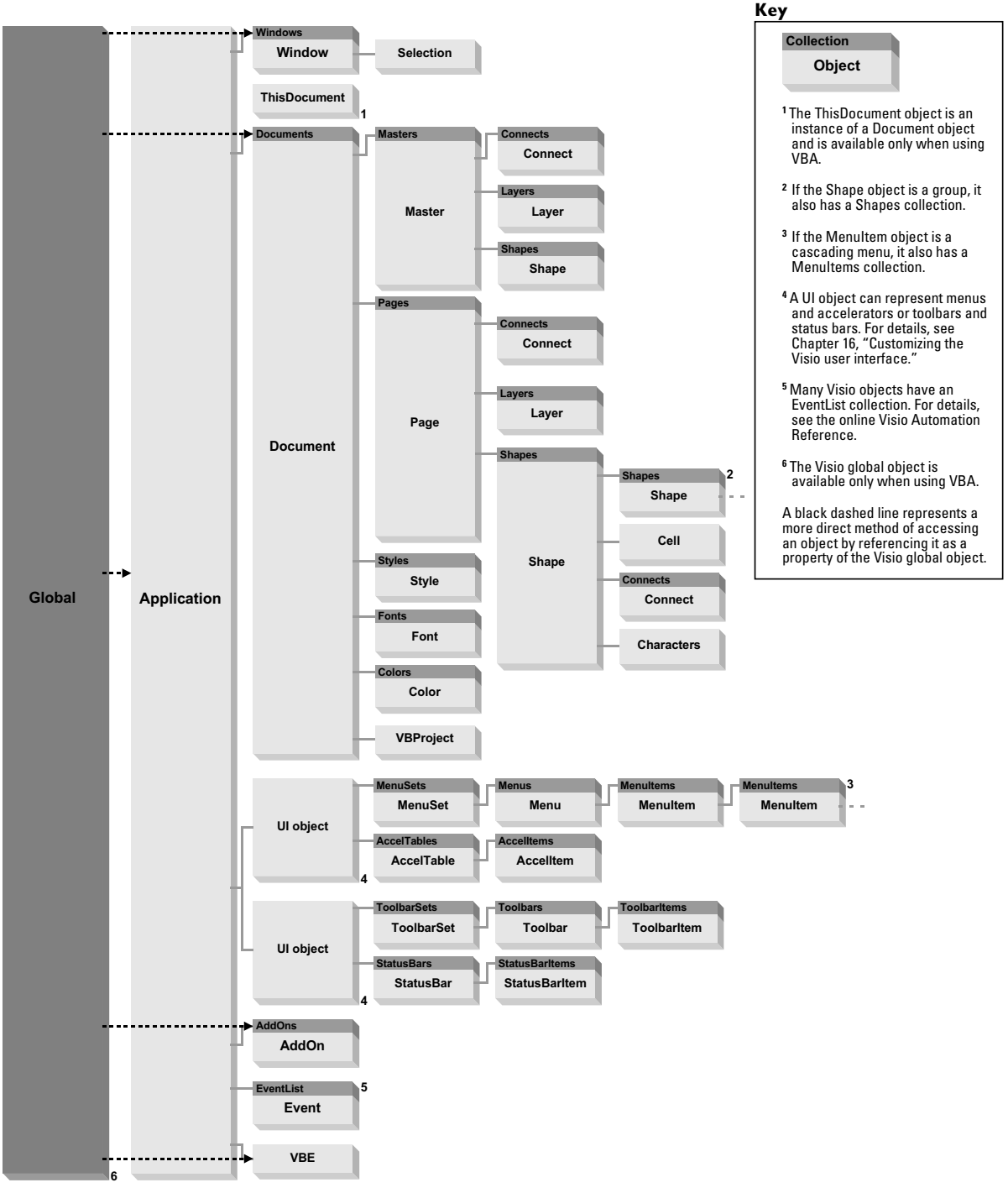
Using a Visio object is really a two-step process: First you get a reference to the object, then you use the object's properties and methods to do something. You usually get a reference to an object by getting a property of an object higher in the Visio *object model*, which is the hierarchy of objects that Visio exposes through Automation.

This chapter describes the objects that Visio exposes through Automation and shows how to access them from a program. It briefly covers the Visual Basic syntax for using objects, properties, and methods, describes the Visual Basic for Applications (VBA) sample code and files provided in the DVS (Developing Visio Solutions) folder on your Visio 5.0 CD, and offers suggestions for handling errors.

For details about how to use the VBA development environment, see Chapter 2, "Tools for creating solutions," or Microsoft Visual Basic online help. For details about creating an external Visual Basic program, see Chapter 19, "Programming Visio with Visual Basic." For comparable information about C++ syntax, see Chapter 20, "Programming Visio with C++."

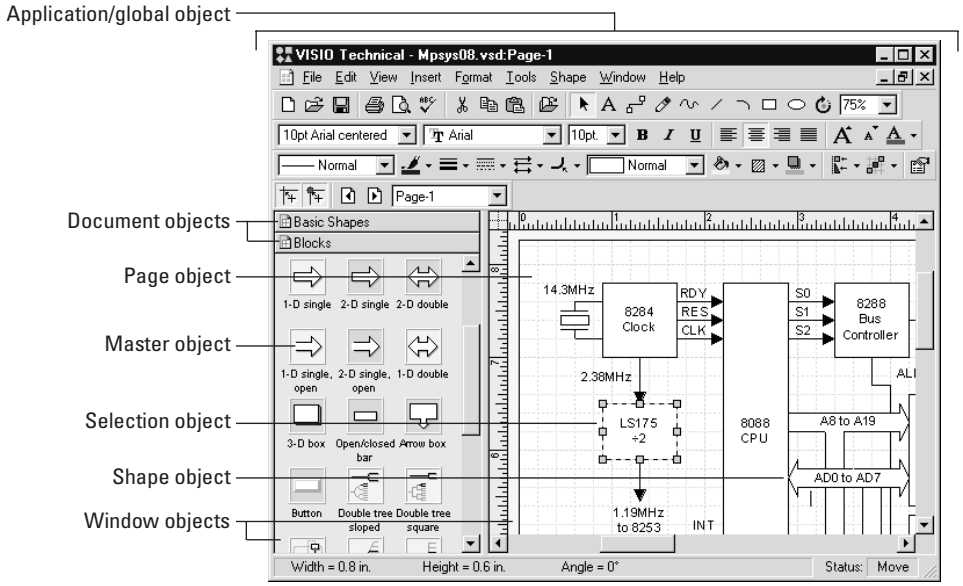
Topics in this chapter

The Visio object model



The Visio object model represents the objects, properties, methods, and events that Visio exposes through Automation. More important, it describes how the objects are related to each other. Many of the objects are used primarily to access other objects. For example, you probably won't do as much with documents and pages as with shapes and cells.

Most objects in the model correspond to items you can see and select in Visio. For example, a Shape object can represent anything on a Visio drawing page that you can select with the pointer tool—a shape, a group, a guide, or an object from another application that is linked, embedded, or imported into a Visio drawing.



Many Visio objects correspond to items you can see and select in Visio.

Some objects represent collections of other objects. A *collection* contains zero or more objects of a specified type. For example, a Document object represents one open document in an instance of Visio; the Documents collection represents all of the documents that are open in the instance. Collections are discussed in more detail later in this chapter.

Online Automation references

You can find details about any Visio object, its properties, its methods, and its events in the online Visio Automation Reference. To use the Visio Automation Reference, choose Automation Reference from the Visio Help menu.

You can find information about VBA programming in Microsoft Visual Basic online help. To use it, choose Microsoft Visual Basic Help from the Help menu in the Visual Basic Editor.

Accessing objects through properties

Most Visio objects have properties whose values refer to other objects. You use these properties to access the objects that you want to control—starting with global properties such as Documents, Windows, or ActivePage or starting with the ThisDocument object, depending on what your program does. The Documents collection represents all documents open within a Visio instance. The ThisDocument object represents the Visio document associated with your VBA project.

When you are running a VBA program within Visio, you don't need to start by creating or getting an Application object because you are already running an instance of Visio. At least one document is also open, so you can directly access the **ActiveDocument**, the **ActiveWindow**, and the **ActivePage** global object properties, for example. For a list of the Visio global object properties that you can access directly, see the online Visio Automation Reference.

The Document or ThisDocument object has a **Pages** property that refers to the Pages collection for that document, which you can use to access a particular page. A Page object has a **Shapes** property that refers to the Shapes collection for that page, which you can use to access individual shapes.

Conversely, most objects have a property that refers to the object above it in the hierarchy, such as the **Document** property of a Page object.

There are often several paths to the same object. For example, to access a particular document, you might use the **Documents** property or the **ActiveDocument** property of the Visio global object, or the **Document** property of a Window object. Which approach you take depends on where you are and what you're trying to do. Using a property of the Visio global object is a more direct approach because you don't have to get references to as many objects.

Once you have a reference to an object, you can set and get the values of its properties or use methods that cause the object to perform actions.

Creating an application object

Earlier versions of Visio did not include the VBA development environment, so programmers using Visual Basic to control Visio started programming by getting or creating an object reference to the Application object—the first object in the Visio object model when writing external programs. For details about getting or creating a Visio Application object, see Chapter 19, "Programming Visio with Visual Basic."

Here is an example of a path of object references, starting with the Documents collection, that gets a reference to the first shape on the first page in the first open document in the Documents collection:

```
Dim docObj as Visio.Document
Dim pagsObj as Visio.Pages
Dim pagObj as Visio.Page
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
docObj = Documents.Item(1)
pagsObj = docObj.Pages
pagObj = pagsObj.Item(1)
shpsObj = pagObj.Shapes
shpObj = shpsObj.Item(1)
```

Getting a document name: an example

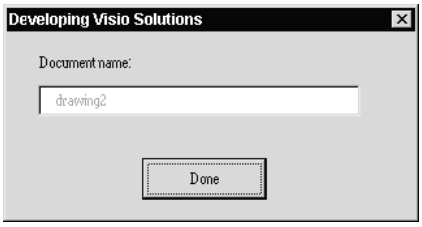
This simple VBA program displays the name of a document open in an instance of Visio in a text field on a user form. To run this VBA program, open the VBA Samples Template (VBA SAMPLES.VST), choose Macro from the Tools menu, then DVS, and then GetDocName. This program follows these steps:

1. Gets the first Document object from the collection.
2. Gets the Document object's **Name** property.
3. Displays the value returned by the **Name** property in a text box on a user form.
4. Sets the label text on the user form and displays the user form.

The code for this example is in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST.

GetDocName macro in the DVS module in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST

```
Sub GetDocName ()
   'Declare variables
   Dim docObj as Visio.Document
   Dim strDocName As String

   'Get the first document in the Documents collection
   Set docObj = Documents.Item(1)

   'Get the current Document object's Name property
   strDocName = docObj.Name

   'Set the Text property of the text box to the document name
   UserForm1.TextBox1.Text = strDocName

   'Set the Label text and show the user form
   UserForm1.Label1.Caption = "Document name:"
   UserForm1.Show

End Sub
```



Displaying a document name in a text box

To see the sample code for a VBA program that iterates through a Documents collection and gets the names of all documents open in an instance of Visio, open the VBA Samples Template (VBA SAMPLES.VST), then the DVS module, and view the GetAllDocNames macro.

Creating a simple drawing: an example

Here's a more elaborate program that creates a drawing. To run this VBA program, open the VBA Samples Template (VBA SAMPLES.VST), choose Macro from the Tools menu, then DVS, and then HelloWorld. This program follows these steps:

Good programming techniques

This book uses the VBA default user form and control names for clarity, but most programmers change the default names to something more descriptive. Many programmers use the following naming conventions:

User form default name = *UserForm1*
Revised name = *frmGetDocName*

Notice the use of *frm* in the revised name of the user form. Many programmers use *frm*, *txt* (textbox), *lbl* (label), *cmd* (command button), and so on in the control name so you know what type of object it is at a glance.

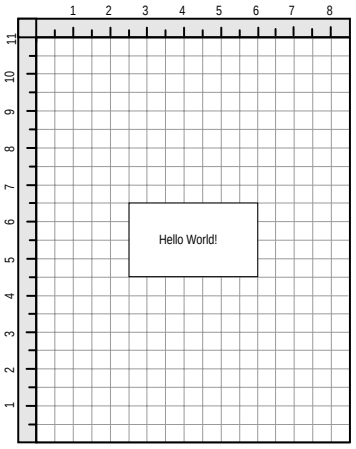
1. Gets the first page in the Pages collection of the document associated with the VBA project.
2. Gets the stencil (VBA SAMPLES.VSS) from the Documents collection.
3. Drops an instance of the Rectangle master from the stencil on the drawing page.
4. Sets the text of the rectangle shape on the drawing page to "Hello World!"
5. Saves the document.

The code for this example is also in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST. Notice that this example, unlike the previous one, actually creates objects in Visio, as opposed to just referring to objects that already exist.

HelloWorld macro in the DVS module in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST

```
Sub HelloWorld ()
  'Object variables to be used in the program.
  Dim stnObj As Visio.Document   'Stencil document that contains master
  Dim mastObj As Visio.Master    'Master to drop
  Dim pagsObj As Visio.Pages     'Pages collection of document
  Dim pagObj As Visio.Page       'Page to work in
  Dim shpObj As Visio.Shape      'Instance of master on page

  'Get the first page in the document associated with the VBA program.
  'A new document always has one page, whose index in the Pages collection is 1.
  Set pagsObj = ThisDocument.Pages
  Set pagObj = pagsObj.Item(1)

  'Get the stencil from the documents collection and set the master shape.
  Set stnObj = Documents("VBA Samples.vss")
  Set mastObj = stnObj.Masters("Rectangle")

  'Drop the rectangle in the approximate middle of an US letter page.
  Set shpObj = pagObj.Drop(mastObj, 4.25, 5.5)

  'Set the text of the rectangle.
  shpObj.Text = "Hello World!"

  'Save the drawing. The message pauses the program so you know the drawing is finished.
  ThisDocument.SaveAs "hello.vsd"
  MsgBox "Drawing finished!", , "Hello World!"
End Sub
```

The drawing created by this program looks something like this.



The drawing created by the Hello World program

Here are some notes on the code:

This program uses the Visio object types such as *Visio.Document, Visio.Master, Visio.Pages, Visio.Page*, and *Visio.Shape* defined in the Visio type library. Using Visio object types instead of the general *Object* variable type increases the speed of your program.

Notice the use of object variables to hold references to the Visio objects used in the program. Each **Set** statement assigns an object reference to an object variable, starting with the ThisDocument object. Note the progression from ThisDocument object, to Pages collection, to Page object.

Set pagsObj = ThisDocument.Pages uses the ThisDocument object, which is equivalent to the specific Document object associated with a VBA project. If you want to reference the Document object of the file associated with your VBA project, you don't need to get it from the Documents collection; just begin by referencing the ThisDocument object. The ThisDocument object is discussed later in this chapter.

Set stnObj = Documents("VBA Samples.vss") doesn't reference an object higher in the Visio object model preceding Documents. **Documents** is a property of the Visio *global object*. The Visio global object has properties and methods you can reference with no qualifying object. The Visio global object is discussed later in this chapter.

Set shpObj = pagObj.Drop(mastObj, 4.25, 5.5) uses the **Drop** method to drop a master on the page represented by *pagObj*. (Although you can draw shapes from scratch from a program, dropping a master is a far easier and more common technique.) The *mastObj* argument specifies the master to drop; *4.25, 5.5* are the page coordinates of the location to drop the pin (its center of rotation) of the new shape. These coordinates are measured from the lower-left corner of the drawing page in drawing units expressed as inches. The **Drop** method returns a reference to a Shape object—the new rectangle—which is assigned to the variable *shpObj*.

ShpObj.Text = "Hello World!" assigns the string "Hello World!" to the **Text** property of the Shape object, which causes Visio to display that string in the rectangle. This is similar to selecting a shape with the pointer tool in a Visio drawing window and typing "Hello World!"

ThisDocument.SaveAs "hello.vsd" uses the **SaveAs** method to save the ThisDocument object under the file name HELLO.VSD. Because no folder path is specified, the document is saved in the working folder (probably the folder that contains the program). The **MsgBox** statement simply lets you know the drawing is finished. When you click OK in the message box, the program finishes.

Getting and releasing objects

You get an object by declaring an object variable, getting a reference to an object, and assigning the reference to the object variable. You can then use the object variable to control the object.

Declaring object variables

A variable that stores a reference to a Visio object should be declared as a Visio object type such as Visio.Page and Visio.Document as defined in the Visio type library, or a variable can be declared as the more general Object type. Using Visio object types will increase the speed of your program. For details about Visio object types, see Chapter 2, "Tools for creating solutions."

Most programs have at least one object variable to store a reference to the Page object, for example, that represents the Page that your program will manipulate. You don't have to assign other object references to variables, but it's almost always a good idea, especially if your program refers to an object more than once. The objects you reference depend on the purpose of your program.

You can declare a variable as local, module-level, or global. The scope you choose depends on how you plan to use the variable in your program. For complete details about declaring variables, see the Microsoft Visual Basic online help.

NOTE You can't store the value of an object variable between program executions. An object reference is like a pointer to a memory address: Its value will probably be different every time the program executes.

Possible object conflicts

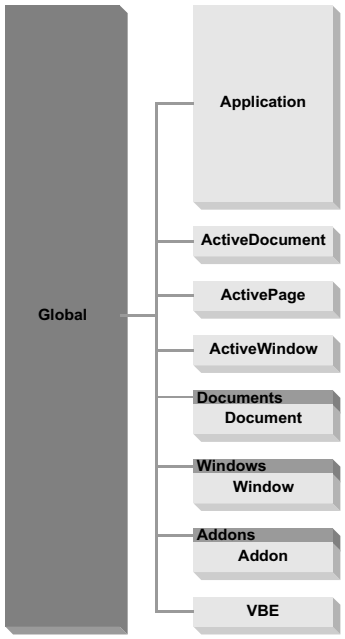
This book uses the syntax *Visio.Page* to define Visio object types. You could eliminate *Visio.* and just use *Page*, but be aware of possible confusion or conflicts in object and property names when programming with other applications and Visio.

For example, other applications may have a Document or Page object. Excel has a Cell object, as does Visio, but the two objects are different and cannot be used interchangeably.

You can decrease the possibility of conflicts by using the syntax shown in this book, which references the library type in object variable declarations.

Using the Visio global object

An external standalone program needs to obtain a reference to the Application object by creating or getting it. When you are running a VBA program, Visio is already running, so you don't need to obtain a reference to the Application object. Instead, Visio provides a *global object* that represents the instance of Visio and provides more direct access to certain properties. Properties of the Visio global object aren't prefixed with a reference to an object.



The Visio global object and its properties

The Application object is a property of the Visio global object, so you can access any of the Application object's properties by directly referencing the **Application** property of the Visio global object.

Here are three examples of code that gets the first document in a Documents collection, but all three use different syntax. The first example creates an Application object—this code is used when writing an external standalone program. The second example uses the **Application** property of the Visio global object. The third example directly accesses the **Documents** property of the Visio global object.

Example 1:

```
Dim appVisio as Visio.Application
Dim docsObj as Visio.Documents
Dim docObj as Visio.Document
Set appVisio = CreateObject("visio.application")
Set docsObj = appVisio.Documents
Set docObj = docsObj.Item(1)
```

Example 2:

```
Dim docsObj as Visio.Documents
Dim docObj as Visio.Document
Set docsObj = Application.Documents
Set docObj = docsObj.Item(1)
```

Example 3:

```
Dim docObj as Visio.Document
Set docObj = Documents.Item(1)
```

Notice in the second and third examples that **Application** and **Documents** are not preceded by an object. When you are referencing any property or method of the Visio global object, you don't need to declare a variable for the global object or reference it as the preceding object of a property—the global object is implied. The third example is the most direct method of accessing the Documents collection. For details about the Visio global object's properties and methods, see the online Visio Automation Reference.

Here are a couple more examples of code for commonly used properties of the Visio global object:

```
Set docObj = ActiveDocument
Set pagObj = ActivePage
Set winObj = ActiveWindow
```

NOTE The Visio global object is not available when you are writing external programs because you are not running your program within an instance of Visio. VBA and other applications also have their own global objects. When more than one type library has a global object with the same name, VBA references the global object with the highest priority. For details about changing the priority of type libraries, see Microsoft Visual Basic online help.

Using the ThisDocument object

The ThisDocument object is equivalent to the specific Document object associated with a VBA project. Every file created with Visio 4.5 and later contains a ThisDocument object that has the same properties, methods, and events as a Document object; however, you don't have to set a reference to ThisDocument as you would with other Visio objects.

If you want to manipulate a document, but not necessarily the document associated with its VBA project, get the Document object from the Documents collection. If you want to manipulate the document associated with your VBA project, use the ThisDocument object.

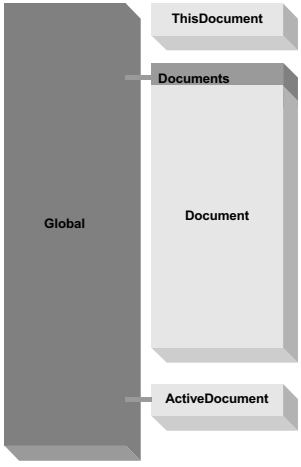
For example, when referencing the document associated with your VBA project, you could get the Document object from the Documents collection. The following example gets the first page of HELLO.VSD, assuming HELLO.VSD is the document object associated with your VBA project:

```
Dim docObj as Visio.Document
Dim pagObj as Visio.Page
Set docObj = Documents.Item("hello.vsd")
Set pagObj = docObj.Pages.Item(1)
```

Or you could just use the ThisDocument object as the following example does:

```
Dim pagObj as Visio.Page
Set pagObj = ThisDocument.Pages.Item(1)
```

You can also add more properties and methods to the ThisDocument object because it is an *extensible object*—an object whose functionality you can extend. The ThisDocument object is the only extensible object Visio provides.



ThisDocument object and related objects in the Visio object model

For details about ThisDocument's properties, methods, and events, select the ThisDocument object in the Project Explorer, open the Object Browser, view the Visio project containing ThisDocument in the project list, and browse the members of ThisDocument.

You can also select ThisDocument in the Project Explorer and change its properties, such as page settings, default styles; and document properties such as title, creator, and subject, in the Properties window.

Getting Visio objects

After you reference a Document or ThisDocument object, you retrieve other Visio objects by getting properties of the Document or ThisDocument object, then of other objects in the object hierarchy.

An object may have more than one property that refers to another object. For example, the Visio global object has two properties you can use to retrieve Document objects: **ActiveDocument** and **Documents**. The **ActiveDocument** property refers to a Document object that represents the active document in an instance of Visio. To retrieve a Document object using the **ActiveDocument** property:

```
Dim docObj as Visio.Document
...
Set docObj = ActiveDocument
```

Referring to an object in a collection

A collection is an object that represents zero or more objects of a particular type. You can iterate through a collection to perform the same operation on all of its objects, or get a reference to a particular object in the collection. A collection differs from an array in that a given object's position is not fixed within its collection—its position may change if another object is added or removed from the collection.

Each collection has two properties you can use to refer to objects in the collection:

- **Item** returns a reference to an object in the collection. This is the default property for any collection.
- **Count** returns the number of objects in the collection.

The **Item** property takes a numeric argument that represents the object's *index,* or ordinal position, within the collection. The first item in most collections has an index of 1 (not 0). To get an object by specifying its index, use code such as the following (where *shpsObj* represents a Shapes collection):

```
Dim pagsObj as Visio.Pages
Dim pagObj as Visio.Page
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
Set pagsObj = ThisDocument.Pages
Set pagObj = pagsObj.Item(1)
Set shpsObj = pagObj.Shapes
Set shpObj = shpsObj.Item(1)
```

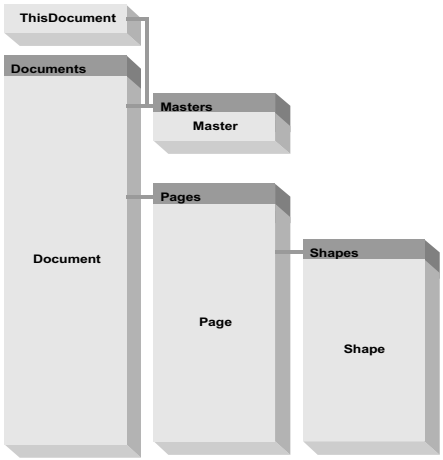
UI object collections

Unlike other Visio collections, the following collections are indexed starting with 0 rather than 1:

- AccelTables
- AccelItems
- Colors
- MenuSets
- Menus
- MenuItems
- StatusBars
- StatusBarItems
- ToolbarSets
- Toolbars
- ToolbarItems

Assuming there's at least one shape on the page, this statement returns a reference to the first Shape object in the Shapes collection. (If the collection is empty, this statement causes an error. You may want to check the **Count** property of a collection before using **Item**, to make sure **Count** is not 0.)

Notice the Visio object type references—*Visio.Pages* represents a Pages collection, and *Visio.Page* represents a page in the collection.



Shape object and related objects higher in the Visio object model

For certain collections—Documents, Pages, Masters, Shapes, or Styles—the **Item** property can also take a string argument that specifies the object's name, which can be more convenient than referring to the object by its index. For example, the following code gets the Master object named 2-D Double in the Masters collection of the stencil document with which the VBA project is associated (using the ThisDocument object):

```
Dim mastsObj as Visio.Masters
Dim mastObj as Visio.Master
...
Set mastsObj = ThisDocument.Masters
Set mastObj = mastsObj("2-D Double")
```

Most objects that belong to a collection have an **Index** property that returns the object's ordinal position within the collection to which it belongs. For example, if a Document object's **Index** property returns *5*, that Document object is the fifth member of its Documents collection.

Iterating through a collection

A collection's **Count** property returns the number of objects in the collection. If the **Count** property is 0, the collection is empty. For example, the following statement displays the number of documents that are open in an instance of Visio:

```
MsgBox "Open documents = " & Str$(Documents.Count)
```

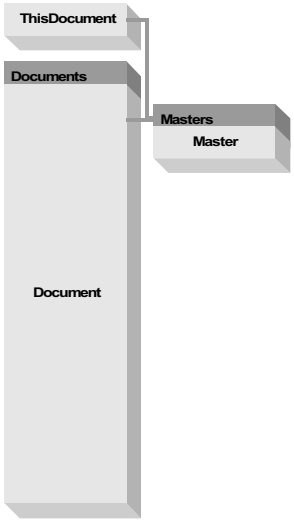
Most often, you'll use the **Count** property to set the limit for an iteration loop. Notice the use of **Count** in the **For** statement of the following example from \DVS\SAMPLE APPLICATIONS\STNDOC\SELSTENC.FRM. The **For** loop iterates through a Documents collection, checking the last three characters of each Document object's file name. If the last three characters are VSS (indicating that the document is a stencil), its name is added to the list in a combo box.

```
Set docs = appVisio.Documents
For i = 1 To Docs.Count
   Set doc = Docs(i)
   If UCase(Right(doc.Name, 3)) = "VSS" Then
      ComboBox1.AddItem doc.FullName
   End If
Next i
```

The code inside a loop such as the previous one should not change the number of objects in the collection (for example, by adding or deleting objects). Otherwise, the value of **Count** changes after each iteration of the loop.

To delete objects from a collection using a loop, decrement the counter rather than incrementing it. Each time an item is deleted from a collection, **Count** decreases by 1 and the remaining items shift position, so an incrementing loop will skip items. Use a loop such as the following instead:

```
Dim shpsObj as Visio.Shapes
...
For i = ActivePage.shpsObj.Count To 1 Step -1
   shpsObj(i).Delete
Next i
```



Master object and related objects higher in the Visio object model

Releasing an object

An object in a program is automatically released when the program finishes running or when all object variables that refer to that object go out of scope. If an object variable is local to a procedure, it goes out of scope as soon as that procedure finishes executing. If the object variable is global, it persists until the program finishes executing, unless the object is explicitly released.

Releasing an object in a program does not affect the corresponding object in Visio. For example, releasing a Document object does not close the corresponding Visio document. The document remains open, but the program no longer has access to it.

To release an object explicitly, set its object variable to the special Visual Basic value **Nothing**. For example:

```
Dim docObj as Visio.Document
...
Set docObj = Nothing
```

If you assign the same object reference to more than one variable, be sure to set each variable to **Nothing** when you release the object.

Don't release an object until you're finished using it. Once you release the object, the program can no longer refer to the corresponding object in Visio. For example, if you release a Document object, the program can no longer manipulate that Visio document, so it is unable to save or close the document or retrieve other objects from it.

On the other hand, if an object reference becomes invalid, you may have to release the object explicitly in your program. For example, if the user closes the Visio document or deletes a shape, references to those objects become invalid. Attempting to use any object variable that contains an invalid object reference will cause an error.

Using properties and methods

Many properties refer to objects, and some methods return object references. Other properties' values are strings or numbers. For example, the value of a Shape object's **Text** property is a string—the text displayed in the corresponding shape.

Declaring variables for return values and arguments

For an object reference, declare an object variable and use a **Set** statement to assign the reference to an object variable. For any other kind of value, you can declare a variable with either an explicit data type or Visual Basic's **Variant** data type, and use a simple assignment statement to assign the value to the variable.

When declaring variables for arguments to a property or method, the same rules apply: Use object variables for objects, and use either the **Variant** data type or the appropriate explicit data type for other kinds of values.

Getting and setting properties

Properties often determine an object's appearance. For example, the following statement *sets* the **Text** property of a Shape object:

```
Dim shpObj as Visio.Shape
...
shpObj.Text = "Hello World!"
```

Read-only, write-only, and read/write properties

Most properties of Visio objects are *read/write*, which means you can both get and set the property's value. Certain properties are *read-only*—you can get them, but you cannot set them. For example, you can get the **Application** property of an object to determine the instance of Visio that contains the object, but you cannot set the **Application** property to transfer the object to a different instance.

A few properties are *write-only*—you can only set their values. Such properties usually handle a special case for a corresponding read/write property. For example, you change the formula in a cell by setting its **Formula** property, unless the formula is protected with the GUARD function. In that case, you must use the **FormulaForce** property to set the formula. However, you cannot get a cell's formula by using **FormulaForce**; you must use **Formula**, whether the cell's formula is protected or not.

For details about properties and methods, including whether a property is read/write, read-only, or write-only, see the online Visio Automation Reference.

The following statement *gets* the text of this shape:

```
Dim shpObj as Visio.Shape
...
shpText = shpObj.Text
```

Some properties take arguments. For example, the **Cells** property of a Shape object takes a string expression that specifies a particular cell in the corresponding shape. When a property takes arguments, enclose them in parentheses. For example, the following statement *sets* the formula of the PinX cell.

```
Dim shpObj as Visio.Shape
...
shpObj.Cells("PinX").Formula = "4.25 in"
```

The following statement *gets* the formula of the PinX cell and stores it in *strPinX*:

```
Dim shpObj as Visio.Shape
...
strPinX = shpObj.Cells("PinX").Formula
```

Using methods

Methods often correspond to Visio commands. For example, a Shape object has a **Copy** method that performs the same action as selecting the shape and choosing the Copy command from the Edit menu in Visio. Other methods correspond to other actions. For example, a Window object has an **Activate** method that you can use to make the corresponding window active, which is the same as clicking that window with the mouse.

The syntax for using a method is similar to that for setting a property. If a method creates an object, like a Page, the method returns a reference to the newly created object, as in the following example. Methods that don't create objects typically don't return values.

```
Dim pagsObj as Visio.Pages
Dim pagObj as Visio.Page
...
Set pagObj = pagsObj.Add
```

Using an object's default property

Most objects have a default property that is used if you don't specify a property when referring to that object. For example, the default property of a Document object is **Name**, so the following two statements return the same value:

```
docName = Documents(5).Name      'long format
docName = Documents(5)           'short format
```

The default property for any collection is Item, so you can use a statement such as the following to specify an object from a collection:

```
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
Set shpObj = shpsObj.Item(1)     'long format
Set shpObj = shpsObj(1)          'short format
```

Using compound object references

You can concatenate Visio object references, properties, and methods in single statements, as you can with VBA objects. However, simple references are sometimes more efficient, even if they require more lines of code.

For example, the following statement refers to the first shape on the third page of the first open document in an instance of Visio:

```
Dim shpObj as Visio.Shape
...
Set shpObj = Documents(1).Pages(3).Shapes(1)
```

Executing this statement retrieves one object—the Shape object assigned to *shpObj*. Compare the following series of statements that use simple object references:

```
Set docObj = Documents(1)
Set pagsObj = docObj.Pages
Set pagObj = pagsObj(3)
Set shpsObj = pagObj.Shapes
Set shpObj = shpsObj(1)
```

Running these statements retrieves five objects: a Document object, a Pages collection, a Page object, a Shapes collection, and a Shape object. References to these objects are assigned to variables and are available for other uses, unlike the previous example. If your program will eventually need access to these intermediate objects, your code will be easier to read and maintain if you retrieve them all in this way.

Using the VBA files provided on your Visio 5.0 CD

The DVS (Developing Visio Solutions) folder on your Visio 5.0 CD contains the VBA and VB code examples discussed in this book. This code is located in the \DVS\VBA SOLUTIONS and the \DVS\VB SOLUTIONS folders. Most of the sample code was designed for use in the VBA development environment with the Visio type library and uses Visio object types; however, this book does include some VB examples also. For details about the Visual Basic files, see Chapter 19, "Programming Visio with Visual Basic."

To run the VBA code samples within Visio (unless otherwise specified):

1. Open the appropriate Visio file—usually the VBA Samples Template (VBA SAMPLES.VST).
2. From the Tools menu, choose Macro, then DVS, then the name of the macro that you want to use.

You can copy and paste, drag and drop, or import the code you need into your own VBA projects.

Files are also provided for creating Visio programs in C++ in the \DVS\LIBRARIES\C-CPP folder. For details about the C++ files, see Chapter 20, "Programming Visio with C++."

IMPORTANT The VBA samples on your Visio 5.0 CD use global constants (from the Visio type library) defined for arguments and return values of properties and methods. For example, suppose you want to find out what type of window—drawing, stencil, ShapeSheet, or icon editing—a Window object represents. The **Type** property of a Window object returns an integer—1, 2, 3, or 4—that indicates the window's type. Because Visio VBA projects automatically reference the Visio type library, you can use the constants **visDrawing**, **visStencil**, **visSheet**, or **visIcon** instead of 1, 2, 3, or 4 to check the window's type.

Handling errors

When an error occurs during program execution, VBA generates an error message and halts execution. You can prevent many errors by testing assumptions before executing code that will fail if the assumptions aren't valid. You can trap and respond to errors by using the **On Error** statement in your program. For details about **On Error**, see your Visual Basic documentation.

Errors can arise from a variety of situations. This section lists some common error situations and suggests ways of preventing them. For more examples of error-handling code, see sample programs such as the Stencil Report Wizard (STNDOC.EXE), in the DVS folder on your Visio 5.0 CD.

Making sure the program is running in the right context

If you've decided which context a program will run in, you can make some assumptions about the environment. For example, if you're writing a VBA program to handle double-click behavior, you can probably assume that a document is open and that the double-clicked shape is selected in the active window. However, there are limits to a program's ability to control user actions. For example, nothing stops a user from attempting to run a VBA program designed to handle a double-click event from the Macros dialog box (instead of double-clicking the shape).

If your program requires a selected shape, check the **Selection** property of the active window to make sure it contains at least one object.

```
Dim selectObj as Visio.Selection
Set selectObj = ActiveWindow.Selection
If selectObj.Count = 0 Then
  MsgBox "You must select a shape first." _
  , , "Select shape"
Else
  'Continue processing
End If
```

Making sure objects exist before attempting to retrieve them

It's a good idea to test whether a collection contains any objects before attempting to access them. The following example checks to see if a document has any masters before attempting to iterate through the Masters collection, which would cause an error if the collection were empty.

```
If ThisDocument.Masters.Count = 0 Then
   stat = appMessage(ERR_FATAL, ERR_NOMASTERS)
End If
```

For details about how this program handles an empty collection, see the **appMessage** procedure in \DVS\SAMPLE APPLICATIONS\STNDOC \SELSTENC.FRM.

Making sure you get what you expect

If a property or method is supposed to return something, it's a good idea to make sure it actually did. For example:

```
Dim shpObj as Visio.Shape
Dim strText As String
strText = shpObj.Text
If strText = "" Then
   MsgBox "The selected shape has no text to format." _
   , , "Format Shape Text"
Else
   'Continue processing
End If
```

Checking for error values

Visual Basic has two error functions, **Err** and **Error**. The **Err** function returns an error code (an integer), and the **Error** function returns a string. When an error occurs in Visio, it returns an error code and a string that describes the error. Use the **Error** function to obtain the string associated with the error code returned by Visio.

Visio's Cell object has an **Error** property, which indicates whether an error occurred when a cell's formula was evaluated. If your program alters ShapeSheet formulas, check this property to make sure the formula works as expected. For a list of possible values, search the online Visio Automation Reference for "error property."

Restricting the scope and lifetime of object variables

Because an object reference exists independently of the item it refers to, object references can become invalid as a result of user actions that are beyond your program's control. For example, if you have a reference to a Shape object and the user deletes the corresponding shape, the reference still exists in your program but it is invalid, because it refers to a nonexistent shape. Or if you have a reference to a Selection object and the user selects different shapes in the drawing, the Selection object no longer represents the shapes that are currently selected.

To prevent invalid references, it's best to restrict the scope and lifetime of an object variable and refresh the variables from time to time. For example, when your program resumes execution after giving control to the user, you may want to release certain objects and retrieve them again to make sure the objects are still available and your program has references to the objects in their current state.

12

Creating Visio drawings from a program

No matter what kind of drawing you create, you'll typically follow certain steps in your program. You'll add shapes to the drawing, often by dropping masters from a stencil. You'll need to determine where to place the shapes, and you may calculate their positions using data gathered from another source. If you're creating a connected diagram such as an organization chart or flowchart, you may glue shapes together.

This chapter describes how to create a drawing from a program by dropping masters onto a drawing page. It describes how to create connected diagrams by gluing shapes and reviews two examples that calculate where to place shapes on the page. The chapter ends with a detailed example of how to create a Visio drawing from a database.

Topics in this chapter

Dropping masters in a drawing

The most convenient means of creating shapes from a program is to drop masters from a stencil. A master is essentially ready to use in a drawing, requiring very little additional processing by your program. You can use masters from a stencil you develop and provide with your program or from any of the stencils provided with Visio.

To drop a master on a page:

1. Create a Document object that represents the stencil containing the master you want.
2. Create a Master object that represents that master.
3. Create a Page object that represents the drawing page where you want to drop the shape.
4. Drop the master on the drawing page.

The following sections explain this procedure in detail.

Getting the stencil

When you're running a VBA program stored in a Visio file, the stencil you want should already be open, so you can simply retrieve it from the Documents collection. For example:

```
Dim stnObj as Visio.Document
...
Set stnObj = Documents("basic shapes.vss")
```

The example above uses the variable name *stnObj* rather than *docObj* to distinguish between the stencil and other kinds of files. This naming convention can prevent confusion later.

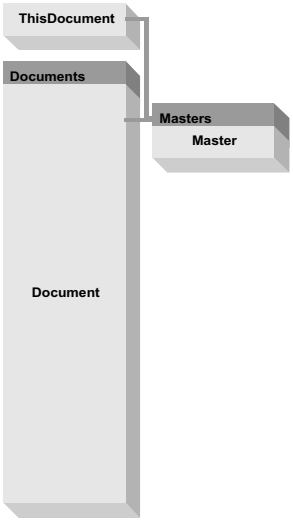
As with any file-related operation, it's prudent to make sure the stencil is actually available before attempting to use it. For example:

```
Dim stnObj as Visio.Document
Set stnObj = Documents("basic shapes.vss")
If stnObj = Nothing Then
   Set stnObj = Documents.OpenEx ("basic shapes.vss", _
   visOpenRO)
End If
```

Drawing original shapes

Although it involves more effort, you can create shapes on the drawing page without using masters. You can create lines, ellipses, and rectangles, or construct more complex shapes from existing shapes. For details, see Chapter 14, "Working with drawings and shapes."

Typically a template is saved as a read-only file to protect it from changes, but it's always possible for the user to open it as an original and alter its workspace, which could affect which stencils are opened when the template is used.



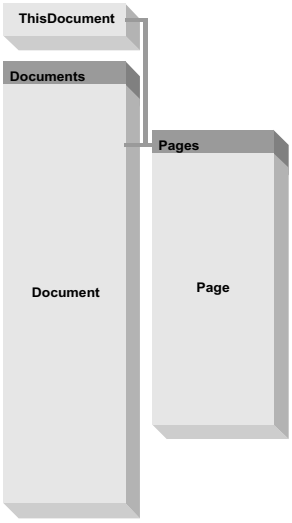
Master object and related objects higher in the Visio object model

Getting the master

A Document object has a **Masters** property that returns a Masters collection of the masters in that document's stencil. You can refer to a Master object by its name or by its index within the Masters collection. For example:

```
Dim stnObj as Visio.Document
Dim mastObj as Visio.Master
...
Set mastObj = stnObj.Masters("Star 5")
```

A common pitfall in this process is to get the Masters collection of the drawing file rather than that of the stencil file. Every Visio document has a stencil, which means that every Document object has a Masters collection. However, the Masters collection of a drawing file contains only the masters that have already been dropped into the drawing; the Masters collection of a new document is usually empty. In either case, this particular Masters collection often won't contain the master you're trying to get. If your program fails to get a Master object, make sure you're getting it from the stencil file and not the drawing file.



Page object and related objects higher in the Visio object model

Getting the drawing page

You create and work with drawing pages by using Page objects and the Pages collection. A Page object represents a drawing page. The Pages collection represents all of the pages in a document.

A new document automatically has at least one page, so you can simply retrieve the first page. Note that the first page in a document is indexed with 1 rather than 0. For example:

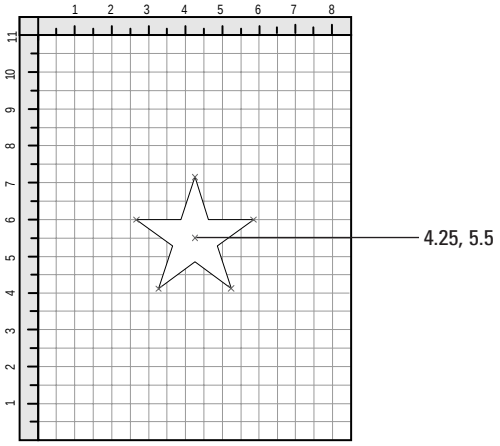
```
Dim pagObj as Visio.Page
...
Set pagObj = ThisDocument.Pages(1)
```

Dropping the master on the page

To drop a master on a page, use the **Drop** method of a Page object. **Drop** takes three arguments: a reference to a Master object and a pair of coordinates that indicate where to position the master's center of rotation (its *pin*) on the drawing page.

```
Dim pagObj as Visio.Page
Dim shpObj as Visio.Shape
Set pagObj = ThisDocument.Pages(1)
Set shpObj = pagObj.Drop(mastObj, 4.25, 5.5)
```

Coordinates are measured from the lower-left corner of the page. In this example, 4.25,5.5 positions the shape's pin in the center of an 8 1/2-in. by 11-in. drawing page in an unscaled drawing. (In a scaled drawing, you specify coordinates in drawing units expressed in inches. For example, if the drawing scale is 1 ft, you would specify the coordinates 51,66 to drop the shape in the center of the page.) For more information about shape coordinates, see Chapter 3, "Controlling shape size and position."



The shape's pin is positioned at the coordinates specified with **Drop**.

Dropping shapes in other places

The **Drop** method is equivalent to dragging and dropping a shape with the mouse. In addition to dropping masters in a drawing page, you can move a shape to another location on the page, add it to a group, or even create a master on the fly by dropping a shape into a stencil (as long as the stencil file is open as an original and not read-only). For details, see Chapter 14, "Working with drawings and shapes."

Dropping multiple shapes

The **DropMany** method is equivalent to dragging and dropping multiple shapes with the mouse.

For an example and details about how to use the **DropMany** method, see the online Visio Automation Reference.

For simplicity's sake, this example drops a single shape in the exact center of the drawing page, and uses constants to indicate its position. However, determining where to place shapes in a real-world drawing can be a challenge, especially in a connected diagram with more than a few shapes. For an example of one approach, see "Placing shapes in an organization chart: an example" later in this chapter.

Adding text to shapes

You'll often set a shape's text from the program rather than providing it as part of a master. You can add text to a shape or change existing text by setting the **Text** property of a Shape object to a string expression. For example:

```
Dim shpObj as Visio.Shape
...
shpObj.Text = "Twinkle"
```

To include quotation marks in the text, use two quotation mark characters ("") to enclose the string. For example:

```
Dim shpObj as Visio.Shape
...
shpObj.Text = """Are you currently a customer of _
XYZ?"""
```

To control where lines break in text, use the Visual Basic **Chr$** function to include an ASCII linefeed with the string. For example:

```
Dim shpObj as Visio.Shape
...
shpObj.Text = "Twinkle," & Chr$(10) & "twinkle" & _
Chr$(10)& "little star"
```

Formatting and positioning text

The text of a shape is formatted according to the existing text format of the shape. If you're using masters to create a drawing, you probably won't need to change this. However, you can change a shape's text format (or its line and fill format, for that matter) by applying a style.

A shape's text is contained in its text block and is positioned in its own coordinate system relative to the shape. You can control the size and position of a shape's text block from a program by setting formulas in the shape's Text Transform section.

For techniques you can use in Visio to change a shape's text block and control text behaviors, see Chapter 6, "Designing text behavior." For details about applying styles and setting shape formulas from a program, see Chapter 14, "Working with drawings and shapes."

Set the **Text** property of a Shape object to add text to the corresponding shape.

To work with part of a shape's text, get the shape's **Characters** property, which returns a Characters object. You set the **Begin** and **End** properties of the Characters object to mark the range of text you want to work with. For an example, see "Creating a network diagram from a database: an example" later in this chapter.

Printing and saving documents

Your program can print or save the drawing it creates. If your program supplements Visio for users who are comfortable with the Visio menu commands, you'll probably create the drawing and leave printing and saving up to the user. If not, you can handle these steps from your program.

Printing documents and pages

You can print a document or a page in a document by using the **Print** method.

To print all of the pages of a document, use **Print** with a Document or ThisDocument object. This is equivalent to choosing All in the Print dialog box in Visio. For example:

```
ThisDocument.Print
```

To print just one page, use **Print** with a Page object. This is similar to displaying that page and choosing Current Page in the Print dialog box in Visio. For example:

```
Dim pagObj as Visio.Page
...
pagObj.Print
```

Saving Visio documents

To save a document from a program, use the **Save** or **SaveAs** method of a Document or ThisDocument object.

Use the **SaveAs** method and supply a file name and path to save and name a new document, to save a copy of an existing document under a different name, or to save an existing document to a different drive or path. For example:

```
ThisDocument.SaveAs "c:\visio\drawings\myfile.vsd"
```

Use the **Save** method only if the document has already been saved and named. For example:

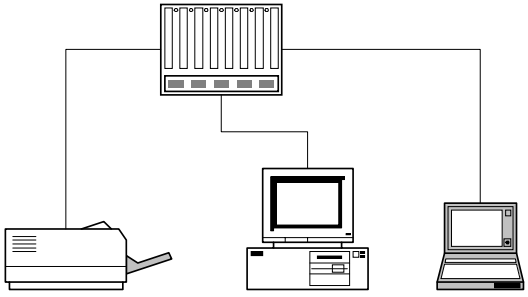
```
ThisDocument.Save
```

Unlike the Save menu command in Visio, which displays the Save As dialog box if a document is unnamed, using the **Save** method on an unnamed document won't invoke the **SaveAs** method—it will cause an error.

To find out whether a document has ever been saved, check its **Path** property, which returns the drive and path of the document's full name or a null string if the document hasn't been saved. To find out whether a document has been saved since changes were made to it, check its **Saved** property. For details, see Chapter 13, "Getting information from Visio drawings."

Creating connected drawings

Connected diagrams are among the most common and useful kinds of drawings you can create with Visio. In Visio, the act of connecting shapes is called *gluing* the shapes. Shapes can be glued to other shapes to create a connected diagram, such as an organization chart, or a directed graph, such as a flowchart. When you create a connected drawing from a program, you drop masters on a drawing page and then glue the shapes together.



A drawing with connected shapes

Gluing is a directional operation, so it's important to know what has been glued to what. Once shapes are glued, you can move a shape that has other shapes glued to it without breaking their connections, but not vice versa. This is true whether you move the shapes from a program or in a Visio drawing window. For example, suppose a line is glued to a rectangle. Moving the rectangle does not break the connection—the line remains glued to the rectangle and stretches as needed. However, moving the line breaks the connection.

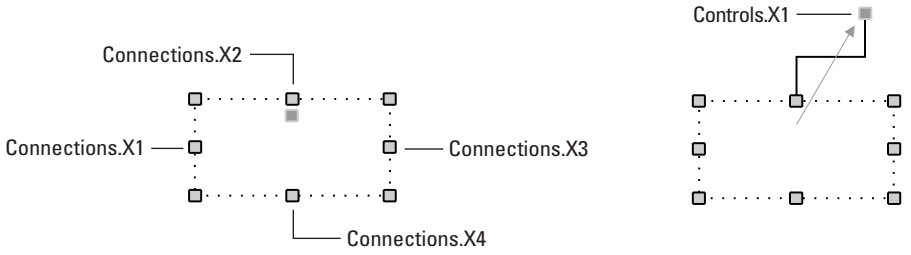
Gluing shapes from a program involves these steps:

- Deciding what shape you want to glue, what shape you want to glue it to, and where to connect the shapes
- Getting a Cell object that represents the part of the shape (such as an end point, control point, or edge of the shape) you want to glue
- Using the **GlueTo** method and specifying part of another shape (such as a connection point, vertex, or selection handle), or using the **GlueToPos** method and specifying a location, to create the connection between the shapes

Deciding what to glue

By far the simplest way to glue one object to another is to use a master with a control handle that extends a line you can glue to another shape. For an example, look at the Position master from the Organization Chart Shapes stencil (ORGANIZATION CHART SHAPES.VSS).

At the top of this master is a control handle that can be glued to another shape. The master also has four connection points, locations to which other shapes can be glued. The cell references you would use to glue these locations are shown in the following illustration.



Cell references for the control handle and connection points on the Position master

Naming connection points

In Visio 5.0, you can provide meaningful names for connection points. For details on naming connection points, see "Adding connection points" in Chapter 5, "Making shapes connect: 1-D shapes and glue."

Control handles work for many kinds of connected diagrams. However, if you don't want to use control handles for this purpose, you can use 1-D shapes instead. You glue the begin point and end point of each 1-D shape between two 2-D shapes, as shown in the following illustration.

Cell references for begin and end points of a 1-D shape

Getting a Cell object

Once you've decided which part of the shape you want to glue, you get a Cell object that represents that part of the shape. To get a Cell object, get the **Cells** property of a Shape object and specify the name of the cell you want. For example, the following statement gets a Cell object that represents the *x*-coordinate of the first control handle of the shape represented by *shpObj1*:

```
Dim shpObj1 as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj1.Cells("Controls.X1")
```

To glue a point on a shape whose coordinates are represented by a pair of cells, you can specify only one cell of the pair. It doesn't matter which cell you specify. In the example above, Controls.Y1 would work equally well.

Gluing the shape to another shape

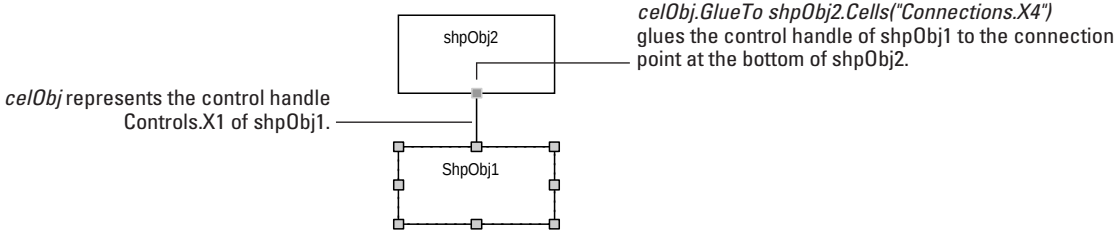
To glue the shape to another shape, you can use the **GlueTo** or **GlueToPos** method of a Cell object. With **GlueTo**, you specify a cell reference for a part of the other shape; with **GlueToPos**, you specify a pair of decimal fractions relative to the other shape's width-height box. **GlueTo** and **GlueToPos** create a connection point at that part of the shape or that location.

For example, the following statement uses **GlueTo** to glue the part of a shape represented by *celObj*—the control handle shown in the following illustration—to the shape represented by *shpObj2*, at that shape's fourth connection point, represented by Connections.X4 in the shape's Connection Points section.

Cell objects and cell formulas

When you glue shapes, the **GlueTo** method sets the formula of the Cell object to the cell reference you supply to identify part of the shape. You'll use Cell objects to do a lot more than glue shapes. You can get and set any cell formula from a program, which gives you a lot of control over a shape's appearance or behavior. For details about getting and setting cell formulas, see "Working with formulas" in Chapter 14, "Working with drawings and shapes."

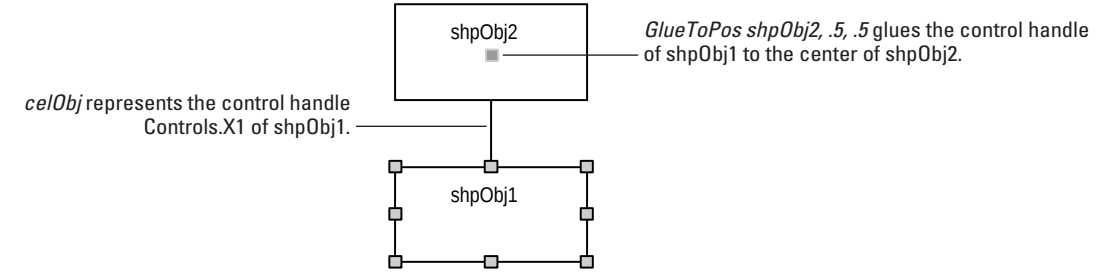
```
Dim shpObj2 as Visio.Shape
Dim celObj as Visio.Cell
...
celObj.GlueTo shpObj2.Cells("Connections.X4")
```



Gluing a control handle to a connection point with **GlueTo**

The following statement uses **GlueToPos** to glue the same shape to the center of *shpObj2*, creating a new connection point at that location. Note that the location is specified as decimal fractions of the shape's width-height box, not as *x,y* coordinates. These fractions can be negative or greater than 1 to create a connection point outside the shape's width-height box.

```
Dim shpObj2 as Visio.Shape
Dim celObj as Visio.Cell
...
celObj.GlueToPos shpObj2, .5, .5
```



Gluing a control handle to a location with **GlueToPos**

What can be glued to what

Only certain parts of shapes can be glued. For example, an endpoint of a 1-D shape or a control handle of a 2-D shape can be glued to a connection point, but a side of a 2-D shape can be glued only to a guide or guide point. The following table lists the ShapeSheet cells that represent the parts of a shape you'll typically want to glue.

To glue	Get one of these cells	And glue it to any of these cells in another shape
The begin point or end point of a 1-D shape	BeginX or BeginY, EndX or EndY	Connections.X*n* or Connections.Y*n*, Geometry.X*n* or Geometry.Y*n*, AlignLeft, AlignCenter, AlignRight, AlignTop, AlignMiddle, AlignBottom, PinX (to dynamically glue)
A control handle	Controls.X*n* or Controls.Y*n*, where *n* is the row number for that control handle	Connections.X*n* or Connections.Y*n*, Geometry.X*n* or Geometry.Y*n*, PinX or PinY, AlignLeft, AlignCenter, AlignRight, AlignTop, AlignMiddle, or AlignBottom, PinX (to dynamically glue)
The edge of a shape	AlignLeft, AlignCenter, AlignRight, AlignTop, AlignMiddle, or AlignBottom	PinX or PinY

Gluing part of a shape represented by a pair of cells. Many points on a shape—control handles, connection points, end points, geometry vertices, and the like—are specified by two ShapeSheet cells, one for each of the *x,y* coordinates for the point. Whenever you glue part of a shape represented by a pair of cells, you can specify either cell of the pair. For example, to indicate the first control handle of the Position shape, you can specify either Controls.X1 or Controls.Y1.

Gluing to selection handles. An alignment cell corresponds to the selection handle in the middle of the specified part of the shape. For example, AlignTop corresponds to the selection handle in the middle of the shape's top edge. Gluing to an alignment cell in a program is the same as gluing to the corresponding selection handle in a drawing window.

Note that you're not actually gluing to the selection handle itself—instead, you're using it to create a connection point at that location on the shape. This is true whether you're gluing the shapes from a program or in a Visio drawing window. A row is added to the shape's Connections section to represent the new connection point.

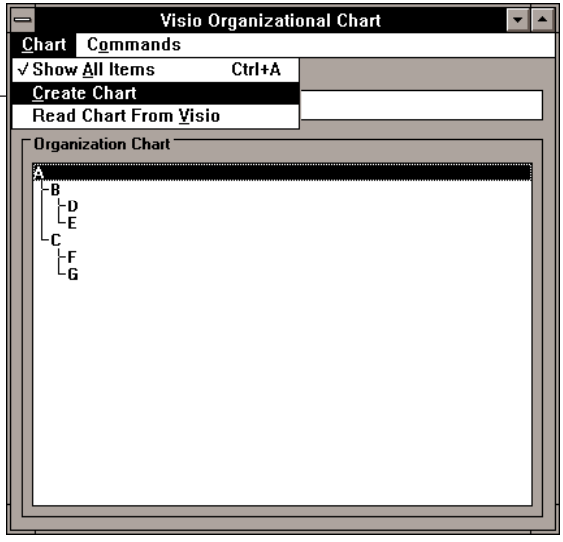
Gluing to guides or guide points. Guides are nonprinting lines dragged out from rulers in the Visio drawing window that you can use to align shapes. You can glue shapes to a guide, then move the guide and the shapes with it. When you glue a 1-D shape to a guide, it doesn't matter whether you specify an X or Y cell as the cell to glue—the type of guide determines which cell or cells are glued.

Connecting shapes in an organization chart: an example

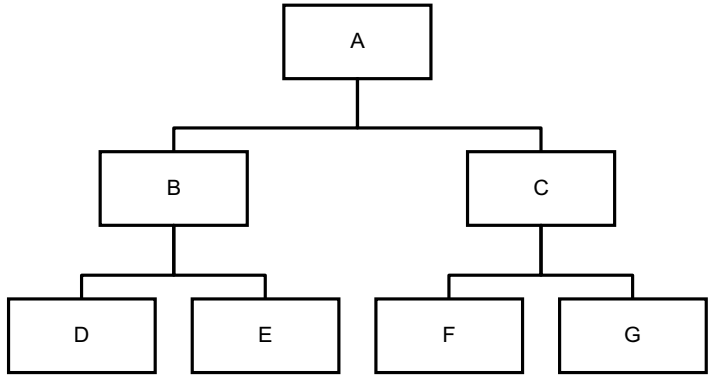
This sample VB program draws an organization chart in Visio based on a hierarchical outline that the user enters in a user form. This program can also read an organization chart drawn in Visio and create an outline from it.

The source code for this program is located in \DVS\VB SOLUTIONS \ORGCHART.

mnuChartItem_Click( ) calls the **CreateOrgChart** subroutine, which calls **DrawOrgChart** to draw the organization chart in Visio.



An outline in ORGCHART.EXE



The organization chart created from the outline

To prepare the organization chart data for Visio, the **CreateOrgChart** subroutine in ORGCHART.BAS loops through the items in the user form's outline control and gets certain information about each item, such as its parent, which is used to connect each box to the box above it in the diagram. Other information is used to determine where to place each box in the drawing. For details, see "Determining where to place shapes" later in this chapter.

For each item in the outline, **DrawOrgChart** calls two subroutines, **PosX** and **PosY**, to calculate the position of the item's box in the organization chart. (More about these subroutines later.) Next **DrawOrgChart** drops the Position master from the organization chart stencil (VB SAMPLES.VSS) at that location on the drawing page. **DrawOrgChart** then assigns the outline item's text to the box in the drawing and repeats with the next item in the outline.

After all the boxes are drawn and their text has been set, **DrawOrgChart** connects the boxes. Notice the statement toward the end of the procedure that uses the **GlueTo** method to glue the control handle (Controls.X1) of each child box to a connection point (Connections.X4) of its parent.

DrawOrgChart macro in \DVS\VB SOLUTIONS\ORGCHART\ORGCHART.BAS

```
Private Sub DrawOrgChart (rgParent() As Integer, rgLeft() As Integer, rgRight() As Integer, _
cLeaves As Integer, cLevels As Integer)
ReDim objArray(cMax) As Object
  Dim objPage As Visio.Page
  Dim objStencil As Visio.Document
  Dim objMasters As Visio.Masters
  Dim objMaster As Visio.Master
  Dim objShapes As Visio.Shapes
  Dim iIndex As Integer
  Dim iIndent As Integer
  Dim X As Double
  Dim Y As Double

'Get the stencil, master, and page objects.
  Set objStencil = Documents.Item("VB Samples.vss")
  Set objMasters = objStencil.Masters
  Set objMaster = objMasters.Item("Position")
  Set objPage = appVisio.ActivePage

'Calculate the pin of each Position shape based on cLeaves and cLevels.
  For iIndex = 0 To frmOrgChart.Outline1.ListCount - 1
    X = PosX(cLeaves, rgRight(iIndex), rgLeft(iIndex))
    Y = PosY(cLevels, iIndex)
    Set objArray(iIndex) = objPage.Drop(objMaster, X, Y)
    objArray(iIndex).Text = (frmOrgChart.Outline1.List(iIndex))
    Set obj.Shapes = objArray(iIndex).Shapes
  Next iIndex

'Glue each child to its parent.
  For iIndex = 0 To frmOrgChart.Outline1.ListCount - 1
    iIndent = rgParent(iIndex)
    If iIndent <> -1 Then
      objArray(iIndex).Cells("Controls.X1").GlueTo _
      objArray(rgParent(iIndex)).Cells("Connections.X4")
    End If
  Next iIndex
End Sub
```

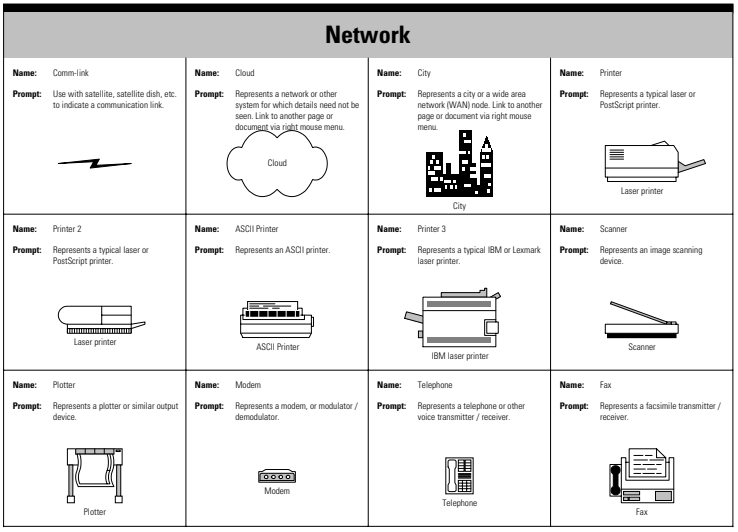
Determining where to place shapes

Determining where to place shapes can be one of the more challenging tasks for a program that creates Visio drawings, especially in connected diagrams or other kinds of drawings with complex relationships between shapes. The ultimate goal is the same: You'll need to calculate a pair of page coordinates for each shape you place on the drawing page. The approach you take will depend on the kind of drawing you're trying to create and the data on which the drawing is based.

This section explores two examples of programs that determine where to place shapes in a drawing.

Arranging shapes on a page: an example

The Stencil Report Wizard (STNDOC.EXE) creates a catalog of the masters in a stencil. Using the stencil and report options selected by the user, the Stencil Report Wizard creates a Visio drawing that shows each master with its name and prompt.



The Stencil Report Wizard creates a drawing with an instance of each master, its name, and its prompt arranged to the user's specifications.

The source code for the Stencil Report Wizard is in \DVS\SAMPLE APPLICATIONS\STNDOC.

The placement of shapes is determined by two subroutines: **pageCompute** and **gridCompute** in MAIN.BAS. The **pageCompute** subroutine obtains dimensional information about the page—its drawing scale, page scale, page height, and page width—by retrieving a special shape called ThePage. (For details about ThePage, see Chapter 14, "Working with drawings and shapes.") The **gridCompute** subroutine uses the metrics calculated by **pageCompute** along with the number of rows and columns requested by the user to build an array of row and column positions.

The **DrawCreate** subroutine in PROGRESS.FRM uses the array produced by **pageCompute** to calculate the page coordinates at which to drop each master in the stencil. Here's the part of **DrawCreate** that actually drops a master on the page:

```
'Drop each master on page
  For row = gGrid.rows - 1 To 0 Step -1
    For col = 0 To gGrid.cols - 1
      If masterIndex > masters.Count Then
        GoTo fexit
      End If

      'Drop master in the center of the grid
      Set master = masters(masterIndex)
      stat = DrawYield("", "", master.Name)

      xLeft = gGridArray(row, col).Left + _
      gGrid.ColWidth / 2
      yTop = gGridArray(row, col).Top - _
      gGrid.RowHeight / 2
      Set inst = page.Drop(master, xLeft, yTop)
```

This subroutine also creates all of the necessary pages for the report, formats them with header, footer, and grid lines, and draws new shapes, which it uses to create new masters by dropping them in the drawing file's stencil. For details, see Chapter 14, "Working with drawings and shapes."

Placing shapes in an organization chart: an example

To calculate where to place each box in the organization chart, the **CreateOrgChart** subroutine in \DVS\VB SOLUTIONS\ORGCHART\ ORGCHART.BAS finds the number of levels (depth of the outline), the

number of leaves (items at the deepest level in the outline), the number of children each item has, and the item's parent. For example, the outline in the illustration shown earlier in this chapter has three levels and four leaves. This information is used to calculate where to place each organization chart box on the drawing page, so that the finished chart is centered vertically and horizontally on the page.

To see how the position of a box is calculated, take a look at the **PosX** and **PosY** functions in ORGCHART.BAS. **PosX** and **PosY** return *x,y* coordinates to **DrawOrgChart**, which passes them as arguments to the **Drop** method to drop the master *objMaster* at the page coordinates *X* and *Y*:

```
X = PosX(cLeaves, rgRight(iIndex), rgLeft(iIndex))
Y = PosY(cLevels, iIndex)
Set objArray(iIndex) = objPage.Drop(objMaster, X, Y)
```

To calculate the *x*-coordinate for a given box in the chart, **PosX** uses the number of leaves in the chart (*cLeaves*) to determine the offset from the left side of the chart, and uses the number of children the box has (*aright* + *aleft*) to center the box over its children.

```
Private Function PosX (cLeaves As Integer, aright As Integer, aleft As Integer) As Double
  Dim MulX As Double, OffX As Double

  MulX = 1.25                                  'Width of the Position shape
  OffX = 5# - (1# * cLeaves) / 2
  PosX = OffX + MulX * (aright + aleft) / 2#
End Function
```

The **PosY** function calculates the *y*-coordinate for the box, using the number of levels in the chart (*cLevels*) to determine the vertical position of the box on the chart.

```
Private Function PosY (cLevels As Integer, index As Integer) As Double
  Dim OffY As Double
  Dim separation As Double

  separation = 1
  OffY = 4.5 + (cLevels * separation) / 2    '1/2(page height - margins)
  PosY = OffY - (frmOrgChart.Outline1.Indent(index)) * separation
End Function
```

Creating a network diagram from a database: an example

This example program shows how to create a simple network diagram from the contents of a database. For a complete listing, see the NetDB1 module in \DVS\VBA SOLUTIONS\NETDB.VST. The database used in this example, NETWORK.MDB, was created with Microsoft Access and is also included on your Visio 5.0 CD.

To create the drawing, the program opens the database and sets up the document and the drawing page. The program drops the Ethernet master on the drawing page, then loops through records in the database, creating a node for each record.

Before creating a node, the program turns screen updating off. Creating a node involves dropping a master for that node in the drawing, labeling the node shape, and connecting the Ethernet shape to the node. The program also formats the first line of the node's label in bold. After each node is created, the program turns screen updating back on so the user sees the node appear in its final user form, rather than seeing each step of its construction.

Opening the database

The database NETWORK.MDB contains one table, NetInfo. Each record in the table contains four text fields: Name, Node, Dept, and Spec. The Name and Dept fields are used to label the nodes in the diagram. The Node field indicates the name of the Visio master from NETDB.VSS that should be used to represent a particular node. The Spec field contains additional information for the diagram.

NetInfo : Table

Name	Node	Dept	Spec
AndyJ	Printer	Product Development	HP LJ III
CindyM	Macintosh	Creative Services	16MB 500MB
JohnF	Desktop PC	Marketing	12 MB 500MB
KenD	Desktop PC	Technical Support	8MB 320MB
MitchS	Workstation	Product Development	24MB 720MB

Record: 3 of 5

Using the data access methods supplied in DAO (Data Access Objects) from VBA, the program first opens NETWORK.MDB, gets a recordset, and assigns the recordset to an object variable called *NetInfo*:

Network Diagrammer

The Network Diagrammer (NETDIAG.EXE) is a more extensive version of the example discussed in this section. To see how Network Diagrammer uses data to create a diagram, look at MAIN.BAS in \DVS\VB SOLUTIONS\NETDIAG.

```
Dim NetBase as DAO.Database
Dim NetInfo as DAO.RecordSet
Set NetBase = OpenDatabase(Visio.Application.Path _
& "dvs\vb solutions\netdb\network.mdb",True,True)
Set NetInfo = NetBase.OpenRecordset("NetInfo", _
dpOpenShapshot)
```

For details about creating and accessing databases for use with programs, see your Visual Basic documentation.

Dropping the Ethernet master

To start creating the network diagram, the program gets the master named Ethernet from the Masters collection of the stencil document, then drops it onto *NetDiagram*. (The Ethernet shape will be positioned more precisely later.) The Shape object returned by **Drop** represents the new Ethernet shape on the page.

```
Set Master = NetStencil.Masters("Ethernet")
Set Ethernet = NetDiagram.Drop(Master, 1, 1)
```

This sample Ethernet shape is a 1-D shape; that is, it has a begin point and an end point rather than sizing handles. The program can use the **SetBegin** and **SetEnd** methods of the Ethernet shape object to position the shape's begin point at the page coordinates 0.5, 2.0 and its end point at the page coordinates 7.5, 2.0.

```
Ethernet.SetBegin .5, 2
Ethernet.SetEnd 7.5, 2
```

Controlling screen updating

With the Ethernet shape in place, the program uses a **While** loop to add a node to the diagram for each record in the database. First, take a look at the loop control structure. (The code inside the loop is discussed in the following sections.) By turning **ScreenUpdating** off before dropping the node master and back on after the master is dropped, the program conceals screen changes, so users see a change only after the new shape has been added. This has a neat and clean effect and is faster than if **ScreenUpdating** is left on. If your drawing is more complex, you might want to move these statements outside the loop to reduce the number of screen updates.

```
NetInfo.MoveFirst
XPos# = 1
Digit% = Asc("1")
While Not NetInfo.EOF
   Visio.ScreenUpdating = False
   ...
   'Drop the nodes in the diagram
   'Label the nodes
   'Connect the nodes to the Ethernet
   'Format the node labels
   ...
   Visio.ScreenUpdating = True
   NetInfo.MoveNext
Wend
```

Placing a node in the diagram

To place a node in the diagram, the program again uses the **Drop** method. For each node, the program drops the master specified in the Node field of the corresponding database record. The coordinate arguments to **Drop** indicate where to place the pin of the shape. In the case of these node shapes, the pin is in the center of the shape. (For clarity, the enclosing statements of the loop control structure are repeated in the code fragments that follow.)

```
While Not NetInfo.EOF
   ...
   NodeType$ = NetInfo.Fields("Node")
   Set Master = NetStencil.Masters(NodeType$)
   Set Shape = NetDiagram.Drop(Master, XPos#, .875)
   XPos# = XPos# + 1.5
   ...
Wend
```

Labeling the nodes

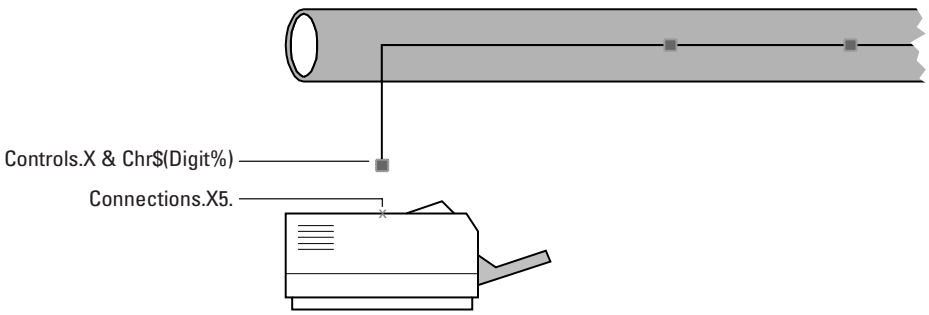
The program constructs the label for each node from the Name and Dept fields in the database table. Once the label is constructed, the program assigns it to the **Text** property of the shape.

```
While Not NetInfo.EOF
   ...
   Label$ = NetInfo.Fields("Name")
   Label$ = Label$ & Chr$(13) & Chr$(10)
   Label$ = Label$ & NetInfo.Fields("Dept")
   Shape.Text = Label$
   ...
Wend
```

Connecting the nodes to the Ethernet shape

The lines of the Ethernet shape have control handles that can be attached to nodes. To attach the Ethernet shape to a node, the program glues one of the control handles of the Ethernet shape to a connection point on a node.

In this example, the object variable *ControlCell* is set to a cell reference of a control handle on the Ethernet shape (Controls.X*n*). The *Digit%* variable picks one handle for each node. Each node has several connection points, but the fifth connection point is at the top of the shape, so that's a reasonable one to use.



Connecting the Ethernet shape to a node shape

The program sets the object called *ConnectCell* to the node's fifth connection point cell (Connections.X5). The final statement uses **GlueTo** to glue the control handle to the connection point.

```
While Not NetInfo.EOF
   ...
   Set ControlCell = Ethernet.Cells("Controls.X" & _
   Chr$(Digit%))
   Digit = Digit + 1
   Set ConnectCell = Shape.Cells("Connections.X5")
   ControlCell.GlueTo ConnectCell
   ...
Wend
```

Formatting node labels

As a final touch, the program formats the first line of each node label in bold. To do this requires slightly more sophisticated knowledge of Visio, because it involves getting a member shape of a group and working with a subset of its text.

Getting the member shape with the text. Each of the node shapes is a Visio group. When the label text was assigned to the node, it was not added to the group but to a member of the group—specifically, the topmost shape. To manipulate the node's text, the program must access this member shape.

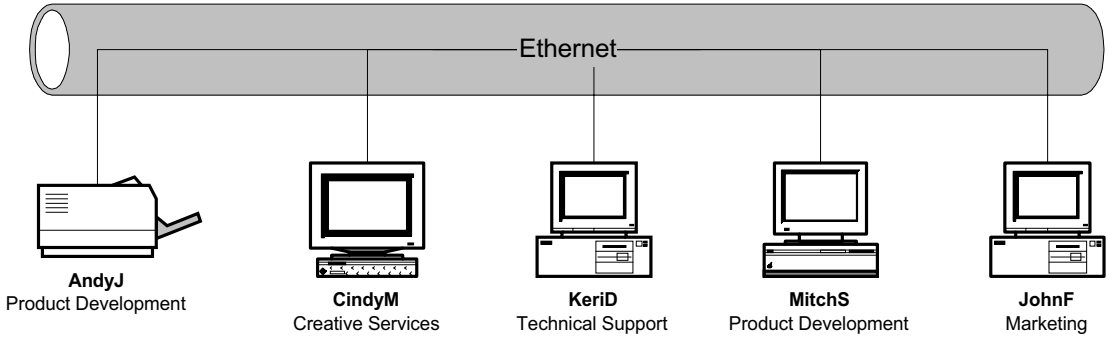
For purposes of Automation, a Visio group is a Shape object with a Shapes collection. To get the topmost shape in the group, the program first gets the count of the shapes in its Shapes collection, then gets the shape with the highest index.

```
While Not NetInfo.EOF
   ...
   Index = Shape.Shapes.Count
   Set TextShape = Shape.Shapes(Index)
   ...
Wend
```

Getting a subset of the shape's text. To work with a range of characters from the shape's text, the program gets a Characters object. This object is retrieved from a shape through its **Characters** property. The character object's **Begin** and **End** properties indicate the begin and end of the desired range. In this example the program wants the text up to the first carriage return. The **Begin** property is initially 0, so it doesn't need to be changed; the program sets **End** to the one character before the return character (Chr$(13)).

After setting the Characters object to the desired text range, the program sets the character style to bold. The constants **visCharacterStyle** and **visBold** are defined by the Visio type library.

```
While Not NetInfo.EOF
   ...
   Set Chars = TextShape.Characters
   Chars.End = InStr(TextShape.Text, Chr$(13)) - 1
   Chars.CharProps(visCharacterStyle) = visBold
   ...
Wend
```



The diagram created from NETWORK.MDB

13

Getting information from Visio drawings

Now you're acquainted with the basics of creating a Visio drawing from a program. You've seen some examples that address common problems, such as creating connected diagrams and determining where to place shapes on a page.

A Visio drawing can also be a rich source of data for other uses. For example, you might generate a furniture order from an office space plan or a parts list from an electrical schematic. This chapter describes how to get information from a Visio drawing by getting properties of documents, pages, and shapes, and by getting the results of formulas. It also describes how to analyze a connected diagram and provides tips for storing Visio data.

Topics in this chapter

Getting information from documents and pages

When working with an existing drawing from a program, you'll often simply get the active page of the active document—that is, the drawing displayed in the active drawing window. However, in some circumstances your program may open a document for the user or retrieve a document that is open but not active.

You'll usually begin by opening a drawing file if necessary, then getting a Page object for the drawing page you want to work with. You can either get the active page in the instance of Visio or get the active document and retrieve a page from its Pages collection.

Getting the active page

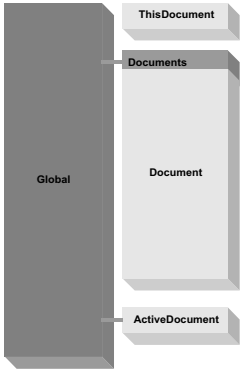
If you expect your user to create or display a drawing and then run your program, it's reasonable to assume that the active page contains the drawing you want. To get the active page for a document, you can get the **ActivePage** property of the global object. For example:

```
Dim pagObj as Visio.Page
...
Set pagObj = ActivePage
```

Getting a document

If you know an open document's file name, you can retrieve it from the Documents collection, whether or not the document is active. For example:

```
Dim docObj as Visio.Document
...
Set docObj = Documents.Item("hello.vsd")
```



Document object and related objects higher in the Visio object model

This example retrieves the document HELLO.VSD from the Documents collection. If HELLO.VSD is not open, attempting to retrieve it causes an error.

Opening a document. You can use the **Open** method of a Documents collection to open a document if you know its path and file name:

```
Dim docObj as Visio.Document
...
Set docObj = _
Documents.Open("c:\visio\drawings\hello.vsd")
```

This statement opens the document HELLO.VSD as an original and adds it to the Documents collection.

You can open any Visio document—stencil, template, or drawing file—with the **Open** method, but this is not recommended for stencils and templates. The **Open** method opens the document as an original rather than as a copy or read-only. An original document can be changed, which is undesirable for stencils and templates because nothing prevents the user from editing masters, altering the template's workspace, or making other potentially unwelcome changes.

To open a Visio document as read-only, use the **OpenEx** method. You can also use **OpenEx** to open a copy of a document, open it without adding its name to the Visio File menu, or open a stencil docked in a drawing window. For details, see **OpenEx** in the online Visio Automation Reference.

Getting the active document. The global object has an **ActiveDocument** property that refers to the document in the active window regardless of the window's type. This statement retrieves the active document in an instance of Visio and assigns it to an object variable named *docObj*:

```
Dim docObj as Visio.Document
...
Set docObj = ActiveDocument
```

As an alternative, if you've retrieved the active window, you can get the **Document** property of that Window object: It refers to the same Document object as does **ActiveDocument**.

Getting information about documents

You can get information about a document by retrieving properties such as **Creator**, **Description**, **Keywords**, **Subject**, and **Title**. These properties correspond to text boxes in the Visio Properties dialog box, which is available from the Properties command on the File menu.

Other ways to get documents

You can get a document by its index in the Documents collection, or you can iterate through the collection to get all of the documents that are open in an instance. For example, you might do this to display each document's file name in a list box. For details about iterating through a collection, see Chapter 11, "Using Visio objects."

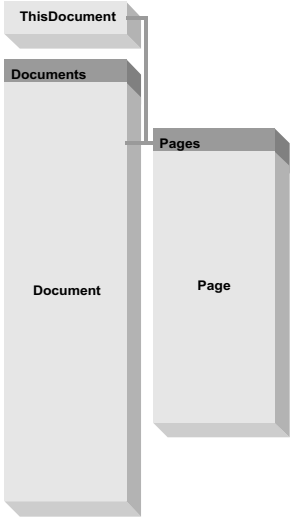
A Document object has three properties you can use to get a document's file name:

- **Name** returns only a document's file name—for example, *hello.vsd*. Until a document is saved, **Name** returns the temporary name of the document, such as *Drawing1*.
- **FullName** returns the drive, path, and file name of a document. For example, *c:\visio\drawings\hellovis.vsd*. Like the **Name** property, until a document is saved, **FullName** returns the temporary name of the document.
- **Path** returns only the drive and path of a document's full name. For example, *c:\visio\drawings*. Until the document is saved, **Path** returns a null string ("").

These properties are read-only. To change the name, drive, or path of a document, use the **SaveAs** method to save the document under a different name or to a different drive or path.

You can get the status of a document by getting its **ReadOnly** or **Saved** property:

- **ReadOnly** returns TRUE if a document is opened read-only.
- **Saved** returns TRUE if the document has no unsaved changes.



Page object and related objects higher in the Visio object model

Getting pages and backgrounds

To get a page from a document, get the Pages collection of the Document or ThisDocument object, then get a Page object from the collection. You can get a page by its index within the collection, or if you know the name of the page, you can get it by name. For example:

```
Dim pagObj as Viso.Page
...
Set pagObj = ThisDocument.Pages.Item("Engineering")
```

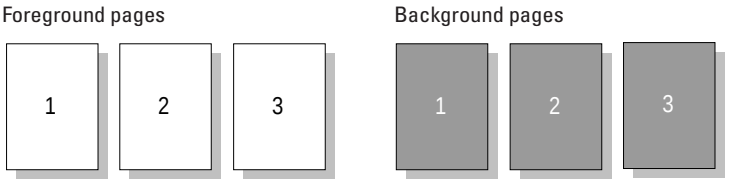
Getting background pages. A drawing can consist of foreground pages and background pages. A background page typically contains a set of shapes, such as a corporate logo or a legend, that appear on more than one drawing in the document. The same background page can be assigned to more than one foreground page. A background page can also have its own background page, so a drawing as it appears in a Visio window can actually consist of any number of pages.

Whether your program should get a background page in addition to the foreground page of a drawing depends on the contents of the background page. A corporate logo or legend probably doesn't contain essential data that your program needs to collect. However, if the shapes on a background page are an integral part of the drawing, you'll want to get these pages, or at least ask the user whether you should.

To determine whether a page has a background, check the Page object's **BackPage** property, which refers to the background page assigned to that Page object. If the page has no background page, **BackPage** returns **Nothing**. Because a background page can have a background page, continue checking the **BackPage** property of each background page until **BackPage** returns **Nothing**. That way, you'll know you have all of the pages that make up the drawing.

To determine whether a page is a background, check the Page object's **Background** property, which is True if the page is a background page or False if it is a foreground page.

Iterating through the Pages collection. The items in a Pages collection are indexed starting with foreground pages in the order they are listed in the Reorder Pages dialog box, followed by background pages in arbitrary order. (To view the Reorder Pages dialog box, choose Drawing Page from the Edit menu, then choose Reorder Pages.)



The order of pages in a Pages collection

Pages, backgrounds, and layers

You can name a page from a program by setting its **Name** property. To name a page in Visio, choose Page Setup from the File menu, then click the Page Properties tab. For details, search Visio online help for "page properties box."

You can create and assign background pages and change page settings from a program. For details, see "Creating and changing pages and backgrounds" in Chapter 14, "Working with drawings and shapes."

Shapes on a page or background can be assigned to layers, which are a means of organizing shapes in a drawing. You can create or change layers or assign shapes to layers from a program. For details, see "Working with layers" in Chapter 14, "Working with drawings and shapes."

The following example iterates through the Pages collection of the active document and lists the names of all foreground pages in a list box on a user form.

```
Sub IteratePages ()

   Dim pagsObj As Visio.Pages
   Dim pagObj As Visio.Page
   Dim i As Integer

Set pagsObj = ThisDocument.Pages
UserForm2.ListBox1.Clear
For i = 1 To pagsObj.Count
   Set pagObj = pagsObj(i)
   If pagObj.Background = False Then
      UserForm2.ListBox1.AddItem pagObj.Name
 End If
Next i
UserForm2.Show

End Sub
```

Getting information about pages

You can get information about the dimensions and appearance of a drawing page by getting a Page object's **PageSheet** property. This property returns a Shape object that represents the page's formulas. These cells correspond to cells in the ShapeSheet window for a page.

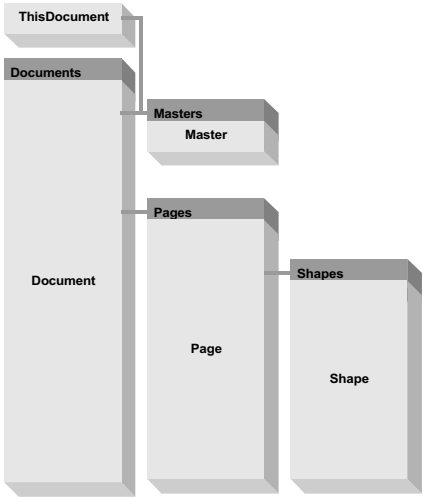
For example, the following statements return the width of the page:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set shpObj = pagObj.PageSheet
Set celObj = shpObj.Cells("PageWidth")
dWidth = celObj.Result("inches")
```

For details about getting Cell objects, see "Getting cells from shapes" later in this chapter. For more details about working with page formulas, see "Creating and changing pages and backgrounds" in Chapter 14, "Working with drawings and shapes." For a list of page cells, see Appendix B, "ShapeSheet sections, cell references, and index constants."

Getting information from shapes

A Shape object represents a basic shape, a group, a guide or guide point, or a linked or embedded object. A Shape object can also represent the formulas of a page or a master. A Shapes collection represents all of the Shape objects on a drawing page, in a group, or in a master.



Shape object and related objects higher in the Visio object model

Getting a shape

To get a Shapes collection of a page, get the **Shapes** property of the Page object. If a Shape object represents a group or the page sheet of a page or master, that Shape object also has a **Shapes** property.

You can get a Shape object from a Shapes collection by its index within the collection, by its name, or by its unique ID.

Getting a shape by its index. The order of items in a Shapes collection is the same as the stacking order of the corresponding shapes on a drawing page. The first item in a Shapes collection is the shape farthest to the back on the drawing page, and the last item is the shape closest to the front. For example, to get the shape closest to the front on the page:

```
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
shpIndex = shpsObj.Count
Set shpObj = shpsObj.Item(shpIndex)
```

Getting a shape by name or unique ID. A Shape object has three properties that identify the shape—**Name**, **NameID**, and **UniqueID**.

Name returns *shapeName[.nnnn]* where *shapeName* is the name of the shape displayed in the Special dialog box and *nnnn* is the shape's ID. (To view the Special dialog box, choose Special from the Format menu.) A shape's name can be from 1 to 31 characters and is usually descriptive of the shape, such as Desktop PC. For example, to get a shape named Workstation:

```
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
Set shpObj = shpsObj.Item("Workstation")
```

Variations on shape names

If a shape has been named, either by setting its **Name** property or by typing a name in the Special dialog box, the **Name** property returns that name. Otherwise, the values returned by **Name** can vary in the following ways:

- If the shape is the first or only instance of a particular master on the page or in a group, the shape's name is the name of the master with no ID number. For example, Decision.
- If the shape is a second or subsequent instance of a particular master on a page or in a group, the shape's name is the name of the master followed by the shape's ID number. For example, Decision.43.
- If the shape is not an instance of a master and has not been named, its name is Sheet. For example, Sheet.34. In this case, the **Name** and **NameID** properties for the shape return the same string.

NameID returns *Sheet.nnnn* where *nnnn* is an integer from one to four digits that indicates the order in which the shape was created on a drawing page—for example, Sheet.34. This ID is assigned to the shape when it is created on a drawing page and is guaranteed to be unique on that page. For example, to get a shape whose ID is 5:

```
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
Set shpObj = shpsObj.Item("Sheet.5")
```

UniqueID(visGetGUID) returns the shape's unique ID if it has one. You can work with unique IDs only from a program—you can't access them in Visio. Most often, unique IDs are used to identify shapes that have corresponding records in a database. For example, an office floor plan might have dozens of identical desk, chair, and PC shapes, but you can use the unique ID of each shape to associate a particular shape in the floor plan with a particular record in a facilities database.

A shape doesn't have a unique ID until a program generates one for it. By contrast, a master always has a unique ID, generated by Visio. A unique ID is stored internally as a 128-bit value, but Visio returns this value as a string. You can pass the unique ID string with the **Item** method to get a shape by its unique ID. For example:

```
Set shpObj = shpsObj.Item("{667458A1-9386-101C-9107-
00608CF4B660}")
```

For more details about unique IDs for shapes and masters, see "Associating data with shapes" in Chapter 14, "Working with drawings and shapes."

Identifying a shape's type

Because a Shape object can represent more than just a basic shape, you may need to determine its type. A Shape object has a **Type** property that indicates the type of shape it is. The values returned by the **Type** property are represented by the following constants defined in the Visio type library:

- **visTypeShape** identifies a shape that is not a group, such as a line, ellipse, or rectangle, including shapes with multiple paths.

- **visTypeGroup** identifies a group. A group can contain shapes, groups, foreign objects, and guides.
- **visTypeForeignObject** identifies an object imported, embedded, or linked from another application.
- **visTypeGuide** identifies a guide or guide point.
- **visTypePage** identifies the page sheet of a page or master.

An instance of a master may be a basic shape or a group, depending on how the master was created.

In the Shapes collection of a Page object, each group counts as a single shape. However, a group has a Shapes collection of its own. The following example counts the shapes on a page, including those in groups (but not the groups themselves) by iterating through the Shapes collection of a Page object, and checking the **Type** property of each Shape object to see whether the shape is a group. If **Type** returns **visTypeGroup**, the example retrieves the number of shapes in the group and adds that number to the total number of shapes.

ShapesCount function in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Function ShapesCount (root As Visio.Shape) As Integer

  Dim iCount As Integer               ' Return value
  Dim shpsObj As Visio.Shapes         ' Shapes collection
  Dim shpObj As Visio.Shape           ' Shape object
  Dim i As Integer                    ' Shapes index

  iCount = 0
  Set shpsObj = root.Shapes           ' Assumes root.Shapes is a group or a page.
  For i = 1 To shpsObj.Count
    Set shpObj = shpsObj(i)

    If shpObj.Type = visTypeGroup Then
      iCount = iCount + ShapesCount(shpObj)
    Else
      iCount = iCount + 1
    End If
  Next

  ShapesCount = iCount

End Function
```

Custom formula fields

Some fields in a shape's text may be custom formula fields, which evaluate to the results of formulas stored in the shape's Text Fields section. To get a custom formula as a string, use the **Formula** property of a Cell object that represents that custom formula's cell in the Text Fields section.

Getting a shape's text

The **Text** property of a Shape object returns a string containing the text displayed in the shape. If a shape's text contains fields, such as a date, time, or custom formula, the string returned by the shape's **Text** property contains field codes, not the expanded text that is displayed for the fields. Each field code is 4 bytes long and starts with the hexadecimal value 1E (decimal 30). The next three bytes contain field data and display format. Field code constants are defined in the Visio type library.

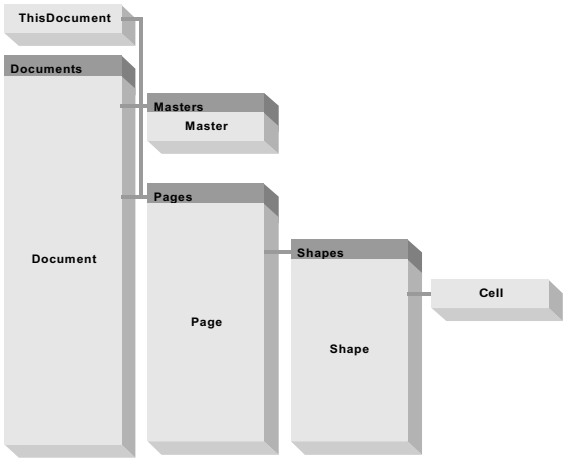
Getting the text with field codes allows you to preserve a shape's text fields when retrieving and setting the shape's text from a program. If you want the shape's text with fields fully expanded to what they display in the drawing window, get the shape's **Characters** property and then get the **Text** property of the resulting Characters object. You can also get a subset of the shape's text by setting the **Begin** and **End** properties of the Characters object. For details, see "Creating a network diagram from a database: an example" in Chapter 12, "Creating Visio drawings from a program."

If the shape has user-defined cells or custom properties, these may also contain text. For details, see the next section, "Getting cells from shapes."

In addition to the **Text** property, which contains the text displayed in the shape, the **Data1**, **Data2**, and **Data3** properties contain the text that appears in the Special dialog box in Visio for that shape. For details about the Special dialog box, search online help for "special command."

Getting cells from shapes

Certain objects have formulas that determine their appearance and behavior. In the Visio object model, a Shape object can have many cells, each having a formula that determines the value of the attribute represented by the cell. To work with a formula, you get a Cell object that contains the formula you want. You can then use the formula or its result in your program.



Cell object and related objects higher in the Visio object model

To work with formulas of a page or master, get the Shape object returned by the **PageSheet** property of the Page object or Master object. You can then use the **Cells** property of that Shape object to work with its formulas. For example, to retrieve a page's width:

```
Dim pagObj as Visio.Page
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
shpObj = pagObj.PageSheet
celObj = shpObj.Cells("PageWidth")
```

Getting a cell by name. To get a Cell object, use the **Cells** property of a Shape object and specify the name of the cell you want. For example, to retrieve a shape's Width cell:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("Width")
```

To refer to the first cell in a shape's first Geometry section:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("Geometry1.X1")
```

Getting cells with CellsSRC

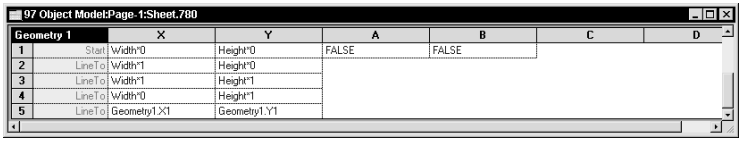
You can also use the **CellsSRC** property to get a cell by section, row, and cell indexes, whether or not that formula also appears in the ShapeSheet window. For example, you can retrieve the formulas that control tabs in text, even though tab settings are not shown in the ShapeSheet window. For details about **CellsSRC**, see "Working with formulas" in Chapter 14, "Working with drawings and shapes."

To refer to the Font cell in the third row of a shape's Character Format section:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("Char.Font[3]")
```

For a list of cell names you can use with **Cells**, see Appendix B, "ShapeSheet sections, cell references, and index constants."

NOTE Because of the way Visio stores the third and fourth cells in the Start row of a Geometry section, their names differ slightly from what appears in the ShapeSheet window. The cell that appears to be Geometry*n*.A1 is actually Geometry*n*.NoFill (Geometry*n*.X0 in earlier versions of Visio). The cell that appears to be Geometry*n*.B1 is actually Geometry*n*.NoShow (also Geometry*n*.A0). You can refer to these cells by either name.



Geometry*n*.NoFill and Geometry*n*.NoShow cells in the ShapeSheet window

Getting a user-defined or custom properties cell. Certain shapes may have cells named by the user or the shape developer. User-defined cells are defined in the shape's User-Defined Cells section; custom property cells are defined in the shape's Custom Properties section. Each row in the User or Custom Properties section has a Value cell that contains the value of the user-defined cell or property, and a Prompt cell that can contain a string. A custom property row has additional cells that control how the custom property can be used.

The Value cell is the default for a user-defined or custom property row, so you can get the Value cell by specifying just the section and name of the row. For example, to get the Value cell of a user-defined cell named Vanishing_Point:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("User.Vanishing_Point")
```

To get any other cell in a user-defined cell or custom property row, you must include the name of the cell you want. For example, to get the Prompt cell for a custom property named Serial_Number:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = _
shpObj.Cells("Prop.Serial_Number.Prompt")
```

For details about defining user-defined cells and custom properties in the ShapeSheet window, see Chapter 4, "Enhancing shape behavior."

Getting the result of a formula

To get the result of a formula, use one of the following methods of a Cell object that represents the formula:

- **Result** returns the formula's result in the units you specify.
- **ResultIU** returns the result in Visio internal units, inches or radians.
- **ResultInt** returns the result as an integer.
- **ResultStr** returns the result as a string.

For example, the formulas that determine local coordinates of a shape's center of rotation are stored in its LocPinX and LocPinY cells. The following statements get the result of the formula in the LocPinX cell:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("LocPinX")
localCenterX = celObj.Result("inches")
```

The **Result** and **ResultIU** methods return both real numbers and integers as floating point numbers with 15 significant digits. When getting the results of certain shape formulas, especially those that determine a shape's dimensions or vertices, you'll probably want to preserve this level of precision. To do this, assign numbers to *Variant* or *Double* variables. This reduces the possibility of rounding errors and maintains the same level of precision if you use the numbers to re-create shapes in other Visio drawings.

Manipulating multiple formulas

You can get and set the results of multiple formulas by using the **GetResults** and **SetResults** methods.

For more details and examples, see the online Visio Automation Reference.

You can specify units using any string that is acceptable to Visio. To specify the Visio internal units (inches or radians), specify a null string ("") for the units, or use the **ResultIU** method instead of **Result**.

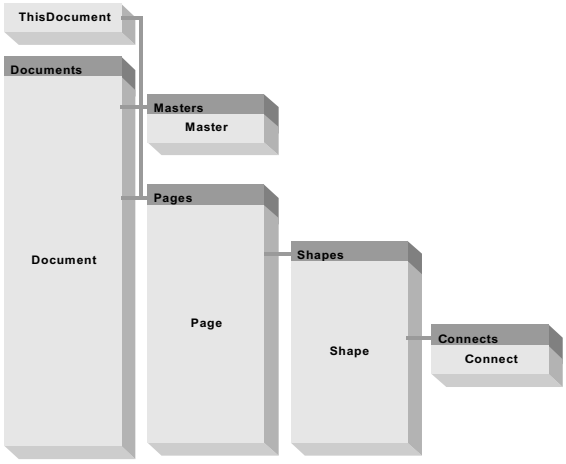
As an alternative to specifying units as a string, use the unit constants defined in the Visio type library. For example, you can specify centimeters by using the constant **visCentimeters**.

```
Dim celObj as Visio.Cell
...
localCenterX = celObj.Result(visCentimeters)
```

Use **visPageUnits** to specify the units defined for the page or **visDrawingUnits** to specify the units defined for the drawing.

Getting information from connected diagrams

Connected diagrams often illustrate relationships in a system, whether between people in an organization or stages in a manufacturing process. It's often easier to design relationships by diagramming them, then using the diagram as a source of data about those relationships.



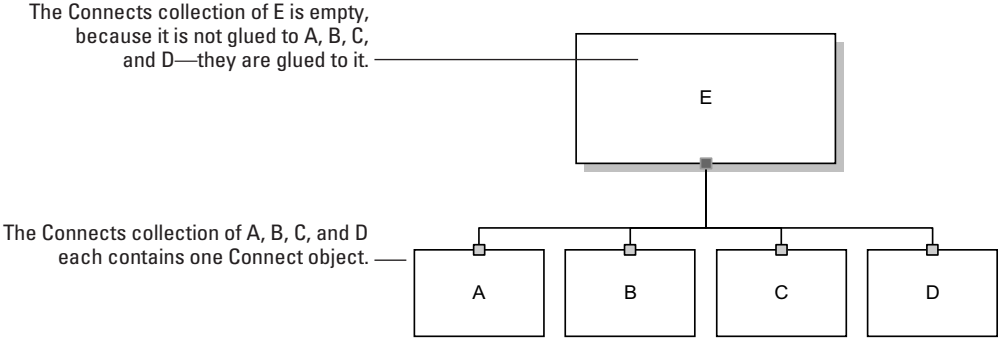
Connect object and related objects higher in the Visio object model

In Visio, a shape can be connected or glued to another shape in a drawing. In the Visio object model, this relationship is represented by a Connect object. You can analyze directed graphs, such as flowcharts, or connected diagrams, such as organization charts, by getting properties of Connect objects for Shape, Master, and Page objects. You can find out which shapes are connected and how they are connected.

Getting a Connect object

The **Connects** property of a Shape object returns a Connects collection that includes a Connect object for each shape, group, or guide to which that shape is glued. The **FromConnects** property of a Shape object returns a Connects collection that includes a Connect object for each shape, group, or guide glued to that shape.

For example, suppose a drawing contains four shapes named A, B, C, and D. Each shape has a control handle that is glued to a shape named E.



Four shapes glued to one shape in a drawing

To get the Connects collection of shape A, get the **Connects** property of that Shape object. For example:

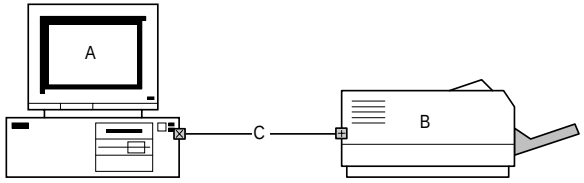
```
Dim shpObj as Visio.Shape
Dim consObj as Visio.Connects
...
Set consObj = shpObj.Connects
```

The Connects collection of shape A contains one Connect object that represents A's connection to E. This is also true of the Connects collections of shapes B, C, and D. The Connects collection of shape E is empty, because it's not glued to the other shapes—they are glued to it.

Page and Master Connect objects

The Page and Master objects also have a Connects collection and Connect object. For an example, see **ShowPageConnections** in "Iterating through the connections on a page: an example" later in this chapter.

Or suppose a drawing contains two shapes named A and B and a 1-D shape named C that connects A and B.



Two shapes connected by a 1-D shape

The Connects collection of shape C contains two Connect objects: one representing its connection to A, and the other representing its connection to B. The Connects collections of A and B are empty, because those shapes are not glued.

Getting Connect object properties

A Connect object has several properties that return information about the connection it represents. You can determine the shapes that are connected and the parts of the shapes that are connected—for example, the top or side of the shape.

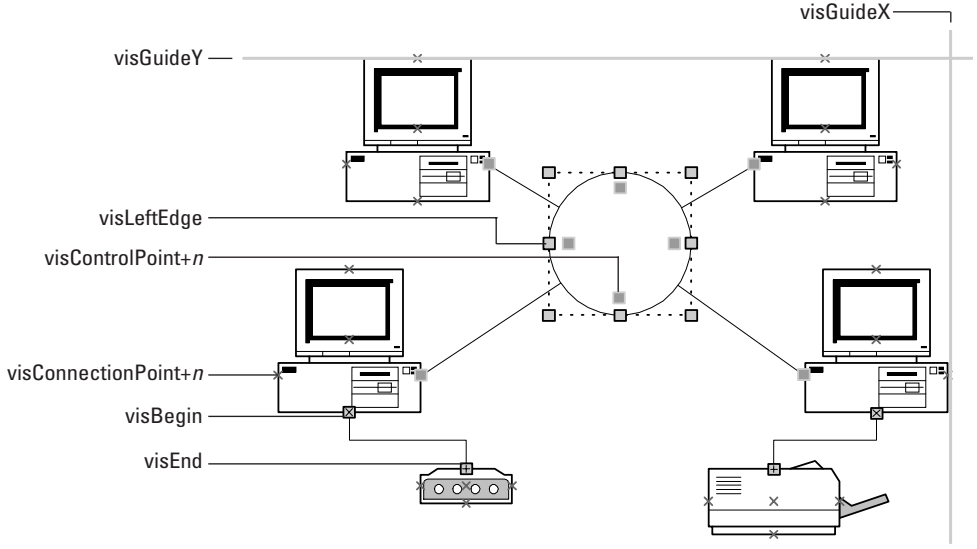
Determining which shapes are connected. The **FromSheet** and **ToSheet** properties refer to Shape objects that represent the shapes that are connected. A shape is defined internally in a spreadsheet similar to that displayed in a ShapeSheet window. These properties derive their names from this internal spreadsheet.

FromSheet returns the shape from which the connection originates; **ToSheet** returns the shape to which the connection is made. For example, suppose a drawing contains two shapes named Executive and Position, and the Position shape is glued to the Executive shape. The Position shape's Connects collection contains one Connect object, whose **FromSheet** property returns Position and whose **ToSheet** property returns Executive.

To find out whether more than one shape is glued to a particular shape, check the **ToSheet** property of each Connect object in a drawing for identical **NameID** values. Continuing the example of the four shapes A, B, C, and D, each glued to E, the **ToSheet** property of each Connect object refers to shape E.

Determining which parts of shapes are connected. The **FromPart** and **ToPart** properties return integer constants that identify the general location of a connection on a shape. **FromPart** identifies the part of the shape from which a particular connection originates; **ToPart** identifies the part of the shape to which a particular connection is made. Constants for valid **FromPart** and **ToPart** values are defined in the Visio type library.

The following illustration shows the **FromPart** and **ToPart** values that would be returned to indicate the parts involved in typical connections in a drawing.



FromPart and ToPart values for typical connections in a drawing

The following table lists typical connections between shapes and the constants for values returned by the **FromPart** and **ToPart** properties of the shapes involved in the illustation above.

Connection	FromPart		ToPart
A control handle glued to a connection point, guide, or guide point	visControlPoint + *n*		visConnectionPoint + *n* visGuideX visGuideY
A 1-D shape glued to a connection point	visBegin	visEnd	visConnectionPoint + *n*
A 2-D shape glued to a guide or guide point	visLeftEdge visCenterEdge visRightEdge	visBottomEdge visMiddleEdge visTopEdge	visGuideX visGuideY
A 1-D shape glued to a guide or guide point	visBeginX visBeginY	visEndX visEndY	visGuideX visGuideY

Because a shape can have more than one control handle, **visControlPoint** is a base that represents the first control handle defined for a shape. If the value returned by **FromPart** is greater than **visControlPoint**, it represents the (*n*+1)th control handle for that shape. (To get *n*, subtract **visControlPoint** from the value returned by **FromPart**.) This is also true of **visConnectionPoint**—if the value returned by **ToPart** is greater than **visConnectionPoint**, it represents the (*n*+1)th connection point.

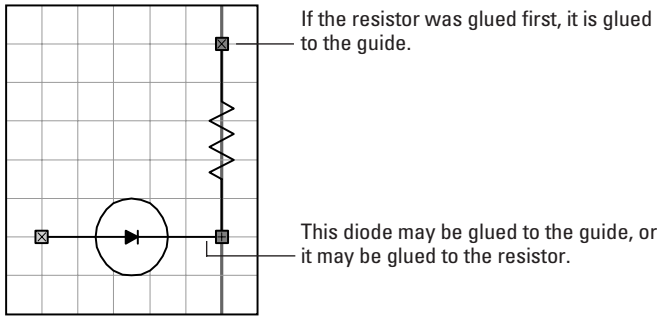
Gluing to a selection handle, vertex, or location within a shape automatically creates a connection point, which is why constants for these items are not defined.

Getting the cells in a connection. The **FromCell** and **ToCell** properties of a Connect object refer to Cell objects that represent the ShapeSheet cells involved in a connection. You can get the cell's formula, its result, or any other property of the Cell object and use it as you would any other Cell object—for example, as an argument to the **GlueTo** method.

Analyzing connections in a drawing. When analyzing a connected drawing, it often helps to know what kinds of shapes it contains. For example, does the drawing use 1-D shapes as connectors between 2-D shapes, or does it rely on control handles of 2-D shapes to draw lines from one shape to another? Are all lines between shapes instances of a connector master, or are some of them drawn with the line tool?

It also helps to remember that the connection data you gather from a drawing created with the mouse may have different connections than

you might assume from looking at the drawing. For example, shapes have a stacking order on the page that can affect what a shape is actually glued to. If the user glues two or more shapes to the same point on another shape using the mouse, some shapes may actually be glued to other glued shapes instead of to the intended shape, as the following figure shows.



The stacking order of shapes can affect their connections.

Direction is another area of potential uncertainty. The parts of shapes that are glued may or may not correspond to directions that are indicated visually in a directed graph such as a flowchart. For example, you can glue either the begin point or the end point of a 1-D shape to another shape, and you can format either the begin point or the end point with an arrowhead. If you assume that an arrowhead in a drawing indicates an end point of a 1-D shape, you may not get an accurate analysis of the drawing.

For more control over what is connected to what, enter connection formulas in ShapeSheet cells, or glue the shapes from a program.

Iterating through the connections on a page: an example

The **ShowPageConnections** macro in the \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE iterates through the Connect objects for the first page in the active Visio document. For each Connect object, **ShowPageConnections** retrieves the shapes that are connected (**FromSheet** and **ToSheet**) and the part of each shape that is connected (**FromPart** and **ToPart**). It then compares the values of **FromPart** and **ToPart** to each possible value, using the constants from the Visio type library, and displays the corresponding string, along with other data for the connection, in a list box on a user form.

ShowPageConnections macro in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub ShowPageConnections ()

  Dim pagsObj As Visio.Pages          ' Page collection of document
  Dim pagObj As Visio.Page            ' Page to work on
  Dim fromObj As Visio.Shape          ' Object from connection connects to
  Dim toObj As Visio.Shape            ' Object to connection connects to
  Dim consObj As Visio.Connects       ' Connects collection
  Dim conObj As Visio.Connect         ' Connect object from collection
  Dim curConnIndx As Integer          ' Loop variable for iterating through connections
  Dim fromData As Integer             ' Type of From connection
  Dim fromStr As String ' String to hold description of From connection
  Dim toData As Integer ' Type of To connection
  Dim toStr As String' String to hold description of To connection

  'Get the pages collection for the document
  'Note the use of ThisDocument to refer to the current document
  Set pagsObj = ThisDocument.Pages

  'Get a reference to the first page of the collection
  Set pagObj = pagsObj(1)

  'Get the connects collection for the page
  Set consObj = pagObj.Connects

  'Make sure the list box is empty
  UserForm2.ListBox1.Clear

  'Loop through the connects collection
  For curConnIndx = 1 To consObj.Count

    'Get the current connect object from the collection
    Set conObj = consObj(curConnIndx)
    'Get the From information
    Set fromObj = conObj.FromSheet
    fromData = conObj.FromPart

    'Get the To information
    Set toObj = conObj.ToSheet
    toData = conObj.ToPart
```

```
    'Use fromData to determine type of connection
    If fromData = visConnectError Then
      fromStr = "error"
      ElseIf fromData = visNone Then
      fromStr = "none"
      ...
      'Test fromData for visRightEdge, visBottomEdge, visMiddleEdge, visTopEdge,
      'visLeftEdge,visCenterEdge, visBeginX, visBeginY, visBegin, visEndX, visEndY,
      'visEnd
      ...
      ElseIf fromData >= visControlPoint Then
      fromStr = "controlPt_" & CStr(fromData - visControlPoint + 1)
      Else
      fromStr = "???"
    End If
    'Use toData to determine the type of shape the connector is connected to
    If toData = visConnectError Then
      toStr = "error"
      ElseIf toData = visNone Then
      toStr = "none"
      ElseIf toData = visGuideX Then
      toStr = "guideX"
      ElseIf toData = visGuideY Then
      toStr = "guideY"
      ElseIf toData >= visConnectionPoint Then
      toStr = "connectPt_" & CStr(toData - visConnectionPoint + 1)
      Else
      toStr = "???"
    End If

  'Add the information to the list box
  UserForm2.ListBox1.AddItem "from " & fromObj.Name & " " & fromStr & " to " & _
  toObj.Name & " " & toStr
  Next curConnIndx

  UserForm2.Show

End Sub
```

Storing Visio data

This section provides tips for storing text and numbers from a Visio drawing. It also includes an example of a program that gathers data about shapes from a drawing and stores it in a simple database.

Retrieving and storing text

No string returned by Visio exceeds 64 kilobytes; most strings are much shorter. The following table lists common properties that return strings and the maximum size of each string.

Property	Object	Maximum size
Creator, Description, Keywords, Subject, Title	Document	63 characters each
Data1, Data2, Data3	Document	64K characters each
Formula	Cell	127 characters
Fullname	Document	255 characters
Name	Document	255 characters
Name	Layer, Master, Page, Shape, Style	31 characters
NameID	Shape	36 characters
Path	Document	255 characters
Prompt	Master	255 characters
Text	Shape, Characters	64K characters
UniqueID	Master, Shape	39 characters

NOTE Visio stores formulas in parsed form rather than as ASCII strings. Depending on the local language version of Visio being used, a formula can exceed 127 characters when it is displayed in that language. In Visio, however, the formula is truncated to 127 characters. Keep this in mind if you retrieve formulas written in an English version of Visio and set them in a German version, for example.

Unique IDs and custom properties

In more sophisticated database solutions you can use unique IDs and custom properties of Visio shapes to ensure uniqueness of database records and associate more data with shapes in a Visio drawing. For details, see "Associating data with shapes" in Chapter 14, "Working with drawings and shapes."

Certain ShapeSheet cells, such as the Prompt cell in a user-defined or custom property row, contain strings you may want to retrieve for other uses. For example, to get the Prompt cell for a custom property:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("Prop.Soc_Sec_Num.Prompt")
```

You can get the result of any cell as a string by using the **ResultStr** method of a Cell object. For details, see the online Viso Automation Reference.

Retrieving and storing numbers

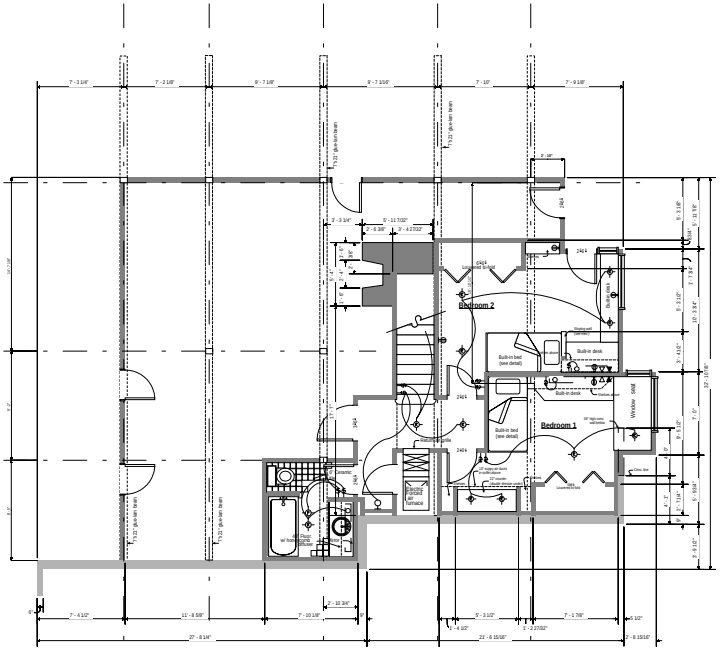
Visio returns both real numbers and integers as floating point numbers with 15 significant digits. To preserve this level of precision, assign numbers to **Variant** or **Double** variables and store numbers in fields with the appropriate data type. This reduces the possibility of rounding errors and maintains the same level of precision if you use the stored numbers to create shapes in other Visio drawings.

True/false properties such as **ReadOnly**, **Saved**, or **OneD** and Boolean values in ShapeSheet cells are FALSE if 0 and TRUE if nonzero.

Storing Visio data in a database: an example

The Visio Inventory program VBINV.EXE gathers data about shapes on a drawing page, stores the data in a Microsoft Access database, and displays the data on a user form. The source code for this program is in \DVS\VB SOLUTIONS\VBINV.

For example, suppose you want an inventory of the shapes in the following drawing.



A Visio drawing to inventory

The program gathers data from the drawing and displays it as shown.

Visio Inventory

File Edit Options Help

Document: floorpln.vsd
Page: Page-1
Sort Field: Name

Choose Fields
ReQuery Visio

Name	Text	Width	Height
30" door.205		30	5.00000000000005
30" door.211	2668	30	4.24999999999994
30" door.214	2668	30	4.24999999999994
30" door.363	2668	30	4.24999999999994
30" door.39		32	5
30" door.54		32	5
30" door.6		32	5
30" door.76		32	5
30" door.83		32	5
36" door	3068	36	4.99999999999984
Base 2		56	24
Bathtub		30	60
Callout	Return air grille	8.22208035054854	4.99999999999995
Callout.258	Shelves above	6.75000000000006	4.99999999999998
Callout.288	22" counter	12	6.5
Callout.289	10" supply air ducts	15.4998581283263	4.99999999999998

Ready

The Visio Inventory program gathers data about shapes on a drawing page.

The **CopyShapesToTable** subroutine in QUERY.BAS gathers the data from the drawing and stores it in the database. This subroutine is called by the **ReQuery** procedure, which is called when a user clicks the ReQuery button on the Visio Inventory form.

CopyShapesToTable in \DVS\VB SOLUTIONS\VBINV\QUERY.BAS

```
Private Sub CopyShapesToTable (shpsShapeColl As Object)

  Dim I As Integer, strStatus As String
  Dim MTable As Table, shpCurShape As Visio.Shape
  Set MTable = m_dbDatabase.OpenTable(QI_MTABLE_NAME)
  EmptyTable Mtable

  For I = 1 To shpsShapeColl.Count         'Loop through shapes
    strStatus = "Retrieving Shape " & Str$(I) & " of "
    strStatus = strStatus & Str$(shpsShapeColl.Count)
    StatusLineMsg strStatus                'Update status line
    Set shpCurShape = shpsShapeColl(I)     'Get next shape
    MTable.AddNew
    MTable.Fields(IDX_NAME) = "" & shpCurShape.Name & " "
    MTable.Fields(IDX_DATA1) = "" & shpCurShape.Data1 & " "
    MTable.Fields(IDX_DATA2) = "" & shpCurShape.Data2 & " "
    MTable.Fields(IDX_DATA3) = "" & shpCurShape.Data3 & " "

    If shpCurshape.Type = visTypeShape Or shpCurshape.Type = visTypeGroup Then
      MTable.Fields(IDX_TEXT) = "" & shpCurshape.Text & " "
    Else
      MTable.Fields(IDX_TEXT) = " "
    End If

    MTable.Fields(IDX_WIDTH) = "" & shpCurShape.Cells("Width") & " "
    MTable.Fields(IDX_HEIGHT) = "" & shpCurShape.Cells("Height") & " "
    MTable.Update
  Next I

  MTable.Close
  ClearStatusLine

End Sub
```

Certain variables used by **CopyShapesToTable** are declared and set in other procedures:

- **ReQuery** retrieves the Shapes collection of the active page and assigns it to the object variable *shpsShapeColl*.

- **InitDatabase** creates a Database object named ~VBINV.MDB and assigns it to *m_dbDatabase*. **InitDatabase** also creates a Table object named **Shapes_Table**.
- The database ~VBINV.MDB and the table **Shapes_Table** are represented by the global constants QI_DBASE_FILE_NAME and QI_MTABLE_NAME, respectively. These constants and the variable *m_dbDatabase* are declared in QUERY.BAS with other constants and variables used by the program.

CopyShapesToTable clears **Shapes_Table** of its existing data, then gets the **Name**, **Data1**, **Data2**, **Data3**, and **Text** properties of each shape in the drawing. It also uses the **Cells** property of the Shape object *shpCurShape* to retrieve the shape's width and height:

```
shpCurShape.Cells("Width") & " "
shpCurShape.Cells("Height") & " "
```

These are actually compound references to the cells named Width and Height in the Shape object's Cells collection, and take advantage of that object's default method—**ResultIU**—to obtain the value calculated by the formulas in those cells. These compound references are equivalent to the following statements:

```
Dim celWidth As Visio.Cell
Dim celHeight As Visio.Cell
Set celWidth = shpCurShape.Cells("Width")
Set celHeight = shpCurShape.Cells("Height")
MTable.Fields(IDX_WIDTH) = _
"" & celWidth.ResultIU & " "
MTable.Fields(IDX_HEIGHT) = _
"" & celHeight.ResultIU & " "
```

The value of each property is stored in the corresponding field in **Shapes_Table**. All of the fields in **Shapes_Table** have the data type DB_TEXT and a length of 255 characters. Although the width and height of a shape are numeric values, the programmer chose to define all of the fields as text and to make each field a generous size, because the number of records collected from a particular drawing is likely to be small. If storage space is a consideration or you need to perform calculations on the numeric data in the table, you would need to choose appropriate data types and sizes.

14

Working with drawings and shapes

By now you're familiar with most of the techniques you'll use to work with Visio from a program. Earlier chapters have covered the basics of getting Visio objects, getting and setting properties, and using methods to create drawings and shapes or get information about them.

This chapter applies these techniques to more specialized tasks, such as creating and changing pages and backgrounds in multiple-page drawing files, drawing shapes that are not based on masters, creating new masters, and modifying shapes. It describes how to work with layers in a drawing and how to create and apply styles.

This chapter provides more detail on working with cell formulas and describes how to modify shapes by working with their ShapeSheet sections and rows. It also provides information on working with data associated with shapes and pages.

Topics in this chapter

Creating and changing pages and backgrounds

A Visio document can contain more than one page. Each page of a document may contain a unique drawing, and some pages can serve as backgrounds to other pages.

You can create multiple-page documents from a program by adding pages and assigning backgrounds to them. You can also change page settings such as the drawing scale and page width and height.

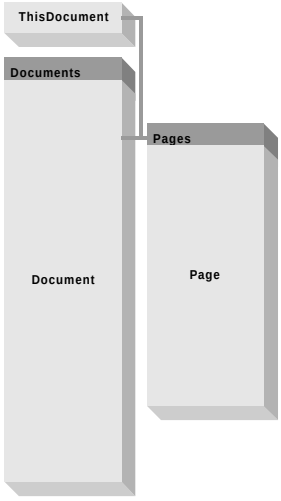
Adding pages to a drawing

Initially, a new document has one page. To create another page, use the **Add** method of the document's Pages collection. For example:

```
Dim pagsObj as Visio.Pages
Dim pagObj as Visio.Page
...
Set pagObj = pagsObj.Add
```

Creating and assigning background pages

When you want the same arrangement of shapes to appear in more than one drawing, you can place the shapes on a background page. For example, if your program creates drawings on multiple pages, you might create a background page with header and footer shapes, or title block and border shapes.



Page object and related objects higher in the Visio object model

The same background page can be assigned to any number of foreground pages. And although a foreground page can have only one background page, a background page can have its own background page, so it's possible to construct a drawing of many pages.

To create a background page, add a page to the drawing and set its **Background** property to TRUE. For example:

```
Dim backPagObj as Visio.Page
...
backPagObj.Background = True
```

To assign the background page to another page so that the background's shapes appear in the drawing window when that page is displayed, set the foreground page's **BackPage** property to the *name* of the background page. For example:

```
Dim pagObj as Visio.Page
...
pagObj.BackPage = "Floor Plan"
```

Changing page settings

Visual Basic for Applications (VBA) programs usually use a Visio template to create a drawing, so you may not need to change settings such as the drawing scale or page scale, because the template can provide the correct settings for the drawings created by your program. If you need to change these settings, or if you create a drawing without using a template but don't want to use the Visio defaults, you can change the page settings by changing page formulas.

To change a page formula, get the **PageSheet** property of a Page object. This property returns a Shape object that represents the formulas of the page.

You use the **Cells** property of this Shape object to retrieve a page cell by name, as you would retrieve a cell for a shape on the drawing page. You can then get or set the Cell object's properties and use its methods to work with the page cell.

For example, suppose your program allows the user to change the scale of a space plan from your program rather than from Visio. The following statements set the scale of *pagObj* so that 1 foot in the drawing equals 1/8 inch on the drawing page.

```
Dim pagSheetObj As Visio.Shape
Dim pagCelPageScale As Visio.Cell
Dim pagCelDrawScale As Visio.Cell
...
Set pagSheetObj = pagObj.PageSheet
Set pagCelPageScale = pagSheetObj.Cells("PageScale")
Set pagCelDrawScale = _
pagSheetObj.Cells("DrawingScale")
pagCelPageScale.Result("in") = 0.125
pagCelDrawScale.Result("ft") = 1.0
```

The page cells you're mostly likely to work with are those that control the drawing's size and scale. Other page cells control the density of the rulers and the grid, the layers defined for the page, actions, and user-defined cells. For a list of page sections and cells, see Appendix B, "ShapeSheet sections, cell references, and index constants."

ThePage shape

As an alternative to getting the **PageSheet** property of a Page object, you can access page settings by getting a special shape called ThePage from the Page object's Shapes collection—both return the same Shape object. The following statements both return the same object:

```
Set shpObj = pagObj.PageSheet
Set shpObj = _
pagObj.Shapes.("ThePage")
```

Page formulas for masters

A Master object also has a **PageSheet** property that you can use to get the same settings—drawing scale, page scale, and so forth—for the master as you can for the page. You might do this, for example, to find out whether the scale of a master is appropriate for the drawing page before you drop the master in the drawing.

Setting up pages and backgrounds: an example

The Stencil Report Wizard creates a multiple-page drawing that contains an instance of each master in a stencil. The **formValid** function in \DVS\SAMPLE APPLICATIONS\STNDOC\SELSTENC.FRM creates the background page and sets its page scale and size to match the scale of the masters that will appear in the report.

This function gets the scales of the master and background page from their respective page sheets. The global variable *gDocDraw* has been previously set by the **Form_Load** subroutine (also in SELSTENC.FRM) to a new document based on the template that accompanies the Stencil Report Wizard (STNDOC.VST). This template happens to be unscaled, which is appropriate for many stencils, such as those with flowchart shapes or shapes for other kinds of diagrams.

The formValid function in \DVS\SAMPLE APPLICATIONS\STNDOC\SELSTENC.FRM

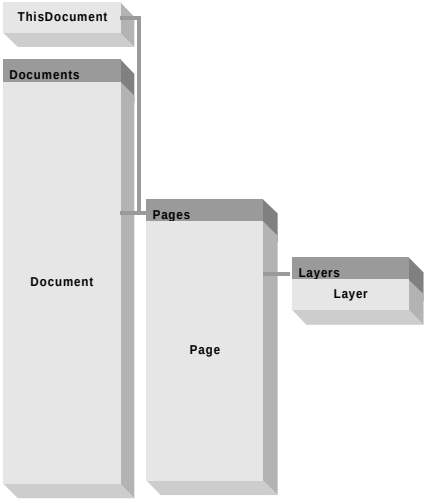
```
Function formValid ()
...
' Declarations and unrelated statements have been omitted.
  Set master = masters(1)        ' assume all masters have same scale
  Set masterSheet = master.Shapes("ThePage")

  ' Page setup for background page.
  Set gPageBack = gDocDraw.Pages.Item(1)
  gPageBack.Name = STR_BACKGROUND
  gPageBack.Background = True

  ' Set page scale and size for background page.
  Set pageSheet = gPageBack.Shapes("ThePage")
  masterDrawingScale = masterSheet.Cells("DrawingScale").formula
  masterPageScale = masterSheet.Cells("PageScale").formula
  pageDrawingScale = pageSheet.Cells("DrawingScale").formula
  pagePageScale = pageSheet.Cells("PageScale").formula
```

After retrieving the scales of the master and the page, **formValid** compares the master's scales with those of the page. If either of the scales differ—as it would for a stencil of architectural shapes, for example—it changes the page scale to the master's scale. It also converts the page's height and width from unscaled to scaled values (*pageHeight * drawingScale / pageScale*).

The formValid function (continued)

```
  If (masterDrawingScale <> pageDrawingScale Or masterPageScale <> pagePageScale) Then
    ' Drawing Scale = Custom
    pageSheet.Cells("DrawingScaleType").Formula = 3
    pageSheet.Cells("DrawingScale").Formula = masterDrawingScale
    pageSheet.Cells("PageScale").Formula = masterPageScale

    ' Drawing Size = Dimensions
    drawingScale = masterSheet.Cells("DrawingScale")
    pageScale = masterSheet.Cells("PageScale")
    pageHeight = pageSheet.Cells("PageHeight")
    pageWidth = pageSheet.Cells("PageWidth")

    pageSheet.Cells("PageHeight").Formula = pageHeight * drawingScale / pageScale
    pageSheet.Cells("PageWidth").Formula = pageWidth * drawingScale / pageScale
  End If

End Function
```

Working with layers



Layer object and related objects higher in the Visio object model

A page can have named layers, which you can use to organize the shapes on a page. You assign shapes to a layer in order to work with categories of shapes—to show them or hide them, print them or not, or protect them from changes—without having to place the shapes on a background page or incur the overhead of grouping them. A shape's layer is independent of its stacking order or even its membership in a group.

A master can be associated with layers. When a master with layers is dropped in a drawing, the instance of that master is assigned to those layers on the page. If the layers don't already exist, Visio creates them.

When you work with layers from a program, you can find out which layers are available in a drawing page or master, and which layers a shape is assigned to in a drawing. You can assign shapes to layers, add layers, and delete layers. You can also show or hide the layer, make it printable or editable, and change other layer settings, similar to the way you set layer properties in the Layer Properties dialog box or the Layers section of the ShapeSheet window.

Identifying layers in a page or master

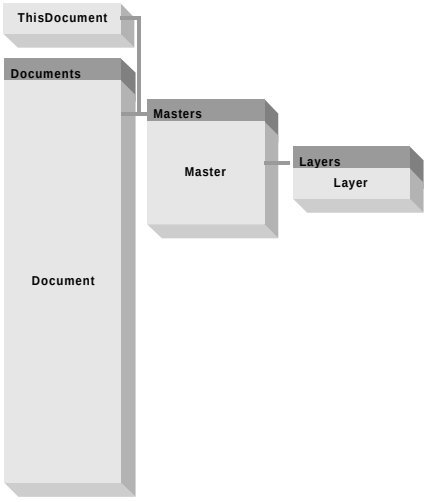
To identify the layers defined for a page or master, get its **Layers** property. This property returns a Layers collection, which contains a Layer object for each layer defined for the page or master. If the page or master has no layers, its Layers collection is empty. A Layer object has a **Name** property that returns the name of the layer as a string. This is the default property of the object.

You can get a Layer object from the Layers collection by its name or by its index within the collection. For example, to get a Layer object for the layer named "Plumbing":

```
Dim layerObj as Visio.Layer
Dim layersObj as Visio.Layers
...
Set layerObj = layersObj.Item("Plumbing")
```

The following example gets all of the layers in a collection and prints their names in the Visual Basic Editor Debug window.

```
Dim pagObj as Visio.Page
Dim layersObj As Visio.Layers
Dim layerObj As Visio.Layer
Dim layerName As String

Set layersObj = pagObj.Layers

For i = 1 To layersObj.Count
   Set layerObj = layersObj.Item(i)
   layerName = layerObj.Name
   Debug.Print layerName
Next i
```



Layer object and related objects higher in the Visio object model

As in most collections, objects in the Layers collection are indexed starting with 1. Each layer in the collection is represented by one row in the Layers section of the page or master.

A Layer object's **Index** property tells you the index of a layer in the Layers collection. A Layer object's **Row** property tells you the corresponding row in the Layers section of the page sheet. These will usually be different numbers.

Identifying the layers a shape is assigned to

Use the **LayerCount** property of a Shape object to get the total number of layers the shape is assigned to, then use the Shape object's **Layer** property to get a particular layer. For example, this statement gets the second layer the shape is assigned to:

```
Dim shpObj as Visio.Shape
Dim layerObj as Visio.Layer
...
Set layerObj = shpObj.Layer(2)
```

Check the properties of the Layer object, such as **Name**, to find out more about that layer.

If the shape is not assigned to any layer, its **LayerCount** property returns 0 (zero), and getting its **Layer** property will cause an error.

Assigning and removing shapes from layers

To assign a shape to a layer, add that Shape object to the Layer object. For example:

```
Dim shpObj as Visio.Shape
Dim layerObj as Visio.Layer
...
layerObj.Add shpObj, preserveMembersFlag
```

The **preserveMembersFlag** argument should be 1 (TRUE) if you're assigning a group to the layer but you don't want to affect the layer membership of shapes within that group. Otherwise, use 0 (FALSE) to assign a single shape or a group and each of its members to that layer.

To cancel a shape's layer assignment, remove that Shape object from the Layer object. The arguments are the same for removing a layer as for adding one. For example:

```
Dim shpObj as Visio.Shape
Dim layerObj as Visio.Layer
...
layerObj.Remove shpObj, preserveMembersFlag
```

Adding and deleting layers from pages and masters

To add a layer to a page or master, use the **Add** method with the Layers collection of a Page object or Master object. For example, to add a new layer named "Plumbing" to a page:

```
Dim pagObj as Visio.Page
Dim layersObj as Visio.Layers
Dim layerObj as Visio.Layer
...
Set layersObj = pagObj.Layers
Set layerObj = layersObj.Add("Plumbing")
```

The name of the new layer must be unique to the page or the master. If successful, the **Add** method returns a Layer object that represents the new layer.

To delete a layer from a page or master, use the **Delete** method of the Layer object. For example:

```
Dim layerObj as Visio.Layer
...
layerObj.Delete fDeleteShapes
```

The **fDeleteShapes** argument should be 1 (TRUE) to delete the shapes assigned to the layer. Otherwise, use 0 (FALSE) to retain the shapes. The shapes' layer assignments are updated so that they no longer refer to the deleted layer.

Changing layer settings

In Visio, you can change settings in the Layer Properties dialog box to make a layer visible or printable or to set its highlight color, among other things.

You change layer settings from a program by setting the formulas of cells that control these settings. To do this, use the **CellsC** property of a Layer object to get the cell that controls the setting you want to change, then set the formula of that cell.

For example, to access the cell that contains the layer's name, use a statement such as the following:

```
Dim layerCellObj as Visio.Cell
...
Set layerCellObj = layerObj.CellsC(visLayerName)
```

TIP You can also access layer settings by using the **CellsSRC** property of a Shape object that represents a page sheet. For details, see "Working with formulas" later in this chapter.

To determine whether a layer is visible, use statements such as the following:

```
Dim layerObj as Visio.Layer
...
If layerObj.CellsC(visLayerVisible).ResultIU = 0 Then
     text1.Text = "invisible"
Else
   text1.Text = "visible"
End If
```

To hide a layer:

```
Dim layerCellObj as Visio.Cell
...
Set layerCellObj = layerObj.CellsC(visLayerVisible)
layerCellObj.Formula = 0
```

The constants **visLayerName** and **visLayerVisible** are defined in the Visio type library. For a list of constants that control layer settings, see Appendix B, "ShapeSheet sections, cell references, and index constants." For details about changing layer settings in Visio, search online help for "layer properties box."

Applying and creating styles

Styles offer the easiest and most versatile way of formatting shapes from a program. A style is a named set of formatting attributes that you can apply to a shape. A style can include text, line, or fill attributes, or any combination of these.

When working with styles from a program, most often you'll simply apply styles that are already defined. You can, however, create new styles from a program, either from scratch or based on existing styles. This section describes how to apply and create Visio styles from a program.

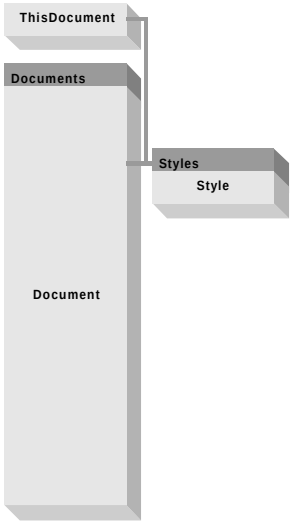
Identifying the styles in a document

To determine what styles are available in a document, get the **Styles** property of a Document or ThisDocument object. The **Styles** property returns a Styles collection, which represents the set of styles defined for a document. The **Name** property of a Style object returns the style name that appears in style lists and the dialog box in Visio.

The following example iterates through the document's Styles collection and lists the style names in a list box on a user form.

ListStyles in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub ListStyles ()

   Dim stylsObj as Visio.Styles
   Dim stylObj as Visio.Style
   Dim curStyleIndx As Integer
   Dim styleName as String
   ...
   Set stylsObj = ThisDocument.Styles
   UserForm2.ListBox1.Clear
   For curStyleIndx = 1 To stylsObj.Count
      Set stylObj = stylsObj(curStyleIndx)
      styleName = stylObj.Name
      UserForm2.ListBox1.AddItem styleName
   Next curStyleIndx
   UserForm2.Show

End Sub
```



Style object and related objects higher in the Visio object model

ThisDocument default styles

During design time, you can change the default style for the ThisDocument object in the Visual Basic Editor. Select the ThisDocument object in the Project Explorer, then change the styles listed in the Properties window. The styles in the Properties window are the default line, text, and fill styles for the ThisDocument object.

Identifying and applying styles to shapes

A Shape object has properties that identify the text, line, and fill styles applied to that shape.

- **FillStyle** identifies a shape's fill style.
- **LineStyle** identifies a shape's line style.
- **TextStyle** identifies a shape's text style.

You can get these properties to determine a shape's text, line, or fill style, or you can set these properties to apply styles to the shape. The following example draws a rectangle and applies two styles.

```
Dim pagObj as Visio.Page
Dim shpObj as Visio.Shape
...
Set shpObj = pagObj.DrawRectangle(5, 4, 3, 2)
shpObj.FillStyle = "10% Gray fill"
shpObj.LineStyle = "9pxl line"
```

You can also set the **Style** property to apply a multiple-attribute style to a shape. If you get a shape's **Style** property, however, it returns the shape's fill style, because a property cannot return multiple objects.

Instead of using styles, you can format any attribute of a shape by setting the cell formulas that control those attributes. For details, see Appendix B, "ShapeSheet sections, cell references, and index constants."

Preserving local formatting

Your program or your user can apply specific formatting attributes to a shape in addition to its text, line, or fill styles. This kind of formatting is called *local formatting*. If you apply a style to that shape later, the style overrides the local formatting unless you preserve it.

To preserve local formatting when applying a style from a program, use one of the following properties instead of **FillStyle**, **LineStyle**, or **TextStyle**:

- **FillStyleKeepFmt**
- **LineStyleKeepFmt**
- **TextStyleKeepFmt**
- **StyleKeepFmt**

These properties correspond to checking Preserve Local Formatting in the Style dialog box.

Creating a style

To create a style from a program, use the **Add** method of a Styles collection and specify the name of the new style. You can optionally specify the name of a style on which to base the new style, and whether the style includes text, line, and fill attributes.

For example, to create a new style named Caption based on the Normal style that includes only text attributes:

```
Dim stylsObj as Visio.Styles
Dim stylObj as Visio.Style
...
Set stylObj = stylsObj.Add("Caption", "Normal", 1, 0, 0)
```

To create a new style that is not based on another style, with text, line, and fill attributes:

```
Dim stylsObj as Visio.Styles
Dim stylObj as Visio.Style
...
Set stylObj = stylsObj.Add("Street Sign","", 1, 1, 1)
```

You can change the style's name by setting its **Name** property, or change whether it includes text, line, or fill attributes by setting its **IncludesFill**, **IncludesLine**, or **IncludesText** property. For details about creating styles in Visio, search online help for "styles."

Changing style attributes

A Style object has a **Cells** property you can use to set formulas for ShapeSheet cells that control formatting of the style. The **Cells** property of a Style object is similar to that of a Shape object, but it can retrieve only cells that control formatting—that is, any cell from the Character, Paragraph, Tabs, Line Format, Fill Format, or Text Block Format sections.

For example, to change the font size of a style:

```
Dim fontsizeCellObj as Visio.Cell
Dim stylObj as Visio.Style
...
Set fontsizeCellObj = stylObj.Cells("Char.Size")
fontsizeCellObj.Formula = "18 pt"
```

For details about defining styles in Visio, see Chapter 7, "Managing styles, formats, and colors."

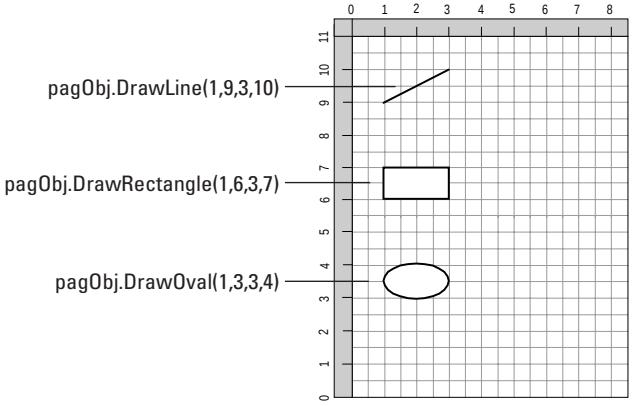
Creating and changing shapes

Most of the work your program does will be with shapes—creating new shapes or changing the way shapes look. You'll often create shapes by dropping masters into a drawing page, as described in Chapter 12, "Creating Visio drawings from a program."

You can also draw original shapes and modify existing shapes, or change a shape's appearance and behavior by setting its formulas.

Creating shapes by drawing

You can draw lines, ellipses, and rectangles from a program by using the **DrawLine**, **DrawOval**, and **DrawRectangle** methods of a Page object. When you draw lines, ovals, and rectangles instead of dropping a master, you supply coordinates for the two opposite corners of the width-height box for the new shape.



Creating shapes with DrawLine, DrawRectangle, and DrawOval

The order in which you specify the corners doesn't really matter when you draw ellipses and rectangles. However, the order does matter for lines. It's often important to know which end of a line is the begin point and which is the end point.

For example, you may want to apply a line style that formats a line's end point with an arrowhead, or glue a line's begin point to another shape. When you draw a line from a program, the first *x,y* coordinate pair determines the line's begin point, and the second *x,y* coordinate pair determines the line's end point—the same as when you draw a shape with the mouse.

Scaled drawings

When you draw or drop a shape in a scaled drawing, specify the coordinates in drawing measurements converted to inches. For example, suppose you want to draw a rectangle from 3 ft, 4 ft to 5 ft, 6 ft. The statement would look like this:

```
Dim pagObj as Visio.Page
...
pagObj.DrawRectangle(36, 48, 60, 72)
```

Copying, cutting, deleting, and duplicating shapes

You can copy, cut, delete, and duplicate shapes using the **Copy**, **Cut**, **Delete**, and **Duplicate** methods of a Shape object. These methods work in the same way as the corresponding menu commands in Visio. You can use these methods with a Shape object whether or not the corresponding shape is selected in the drawing window.

To duplicate a shape at a particular location on the page, use the **Drop** method for a Page object. You supply a reference to the shape you want to duplicate and to the coordinates where you want to position the shape's center of rotation, which is usually the same as the shape's pin. For example:

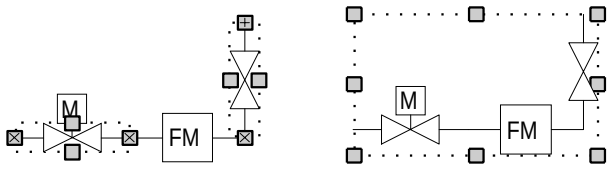
```
Dim pagObj as Visio.Page
Dim shpObj as Visio.Shape
...
Set shpObj = pagObj.Shapes("Sheet.1")
pagObj.Drop shpObj,1,2
```

To paste shapes from the Clipboard, use the **Paste** method of a Page object. For details about pasting from the Clipboard, see the **Paste** method in the online Visio Automation Reference.

Creating groups

A group is a shape composed of other shapes. To create a group from a program, use the **Group** method of a Window object or a Selection object. The following statement creates a group from the selected shapes in a drawing window:

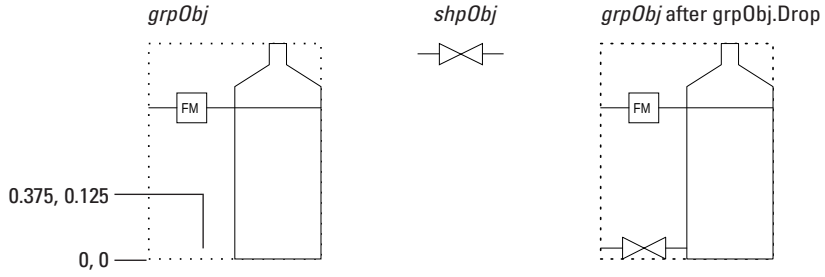
```
Dim winObj as Visio.Window
...
winObj.Group
```



The **Group** method creates a group from selected shapes.

To add a shape to a group, use the **Drop** method of a Shape object that represents the group, with a reference to the shape you want to add and the position of its pin, assuming it is a master, inside the group. For example:

```
Dim grpObj as Visio.Shape
Dim shpObj as Visio.Shape
...
grpObj.Drop shpObj, 0.375, 0.125
```



Use a group's **Drop** method to add a shape to the group.

The coordinates 0.375, 0.125 are expressed in local coordinates of the group. The pin of the added shape is positioned at those coordinates.

Creating masters from a program

To create masters from a program, you drop a shape from a drawing page into a document (often a stencil document), as you do when creating a master with the mouse. Supply a reference to the Shape object that you want to make into a master to the Document object's or ThisDocument's **Drop** method. For example:

```
Dim stnObj as Visio.Document
Dim shpObj as Visio.Shape
Set stnObj = Documents("basic shapes.vss")
stnObj.Drop shpObj, 0, 0
```

Before you can drop a Shape object in a stencil, the stencil must be opened as an original rather than read-only, as is typically the case when a stencil is opened by a template (.VST) file. To open a stencil as an original, use the **Open** method:

```
Dim stnObj as Visio.Document
...
Set stnObj = Documents.Open("basic shapes.vss")
```

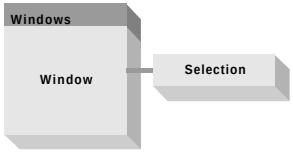
A master often consists of several components, which, for best performance, should be grouped. Visio does not require the components of a master to be grouped in the stencil. However, if they are not, Visio automatically groups the shapes when the master is dropped in a drawing. This increases the time required to create an instance of the master.

Working with selected shapes

You can access a shape's properties and methods from a program whether the shape is selected or not. You can, however, create a Selection object to work with multiple shapes. A Selection object is similar to a Shapes collection in that it represents a set of Shape objects and has an **Item** and a **Count** property. Unlike a Shapes collection, a Selection object represents only the shapes that are selected.

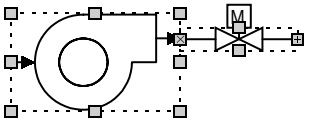
You can get a Selection object that represents the shapes that are selected in a window, or create a Selection object that represents shapes you specify from any Shapes collection.

The order of items in a Selection object follows the order in which the corresponding shapes are selected. The first item returned is the first shape selected.



Selection object and related objects higher in the Visio object model

Getting shapes that are selected in a window. To work with shapes the user has selected in a window, get the **Selection** property of that Window object.



A Selection object represents shapes the user has selected in a drawing window.

Keeping a Selection object current

Whether shapes in a window are selected or deselected by the user or by your program, a Selection object represents the selection that exists when that object is retrieved by your program, and subsequent operations that change the selection in the drawing window have no effect on the object. It's good practice to create a Selection object just before you need it to ensure that it represents the shapes that are actually selected.

The following example gets the Selection object of the active window:

```
Dim selectObj as Visio.Selection
selectObj = ActiveWindow.Selection
```

If all of the shapes on a page are selected, the Selection object of the window and the Shapes collection of the page are the same set of shapes. If nothing is selected, the Selection object is empty and its **Count** property returns 0. If your program requires a selected shape, you might check the **Selection** property of the active window to make sure it contains at least one object.

```
Dim selectObj as Visio.Selection
Set selectObj = ActiveWindow.Selection
If selectObj.Count = 0 Then
  MsgBox "You must select a shape first." _
  , , "Select shape"
Else
  'Continue processing
End If
```

In Visio, each drawing window can have different shapes selected, even if some windows are showing the same drawing page. Only selected shapes and groups are included in the Selection object; shapes subselected within a group are not included.

Adding and removing shapes in selections. To add an object to a selection, use the **Select** method of the Selection object and specify the Shape object to select. You can add a shape to a selection or cancel the selection of a shape without affecting the other selected shapes.

The constants **visSelect** and **visDeselect**, defined in the Visio type library, control the action that is performed. For example, the following statement adds a shape to those already in the Selection object:

```
Dim selObj as Visio.Selection
Dim shpObj as Visio.Shape
...
selObj.Select shpObj,visSelect
```

The following statement cancels the selection of that shape:

```
Dim selObj as Visio.Selection
Dim shpObj as Visio.Shape
...
selObj.Select shpObj,visDeselect
```

Selecting and deselecting shapes in a window. To select a shape from a program, use the **Select** method of a Window object and specify the Shape object to select. You can add a shape to a selection or cancel the selection of a shape without affecting the other selected shapes. For example, the following statement adds a shape to those already selected in a drawing window:

```
Dim winObj as Visio.Window
Dim shpObj as Visio.Shape
...
winObj.Select shpObj,visSelect
```

To select all the shapes on a drawing page, use the Window object's **SelectAll** method; to deselect all selected shapes, use the **DeselectAll** method. If you get a Selection object after using **SelectAll**, the new Selection object includes a Shape object for each shape on the drawing page displayed in that window. If you get a Selection object after using **DeselectAll**, the new Selection object is empty.

Performing operations on selected shapes. After you have a Selection object, you can perform operations on the selected shapes, similar to the actions you can perform in a drawing window.

For example, you can use the **Copy**, **Cut**, **Delete**, or **Duplicate** method of a Window or Selection object to copy, cut, delete, or duplicate selected shapes.

```
Dim selectObj as Visio.Selection
...
selectObj.Delete
```

Union, Combine, and Fragment

Before using **Union**, **Combine**, or **Fragment**, make sure that only the shapes you want to affect are selected. These methods delete the original shapes, so any smart formulas in the original shapes are lost and the Selection object that represents the shapes is no longer current. For details about using **Union**, **Combine**, and **Fragment** in Visio, see the online Visio Automation Reference.

Or you can join or fragment selected shapes using the **Union**, **Combine**, and **Fragment** methods. These methods correspond to the Union, Combine, and Fragment commands in Visio, which create one or more new shapes that replace the selected shapes.

```
Dim selectObj as Visio.Selection
...
selectObj.Union
```

For details about what you can do with a Selection object, see the list of properties and methods for that object in the online Visio Automation Reference.

Finding out where a selection came from. To find out whether a Selection object gets its shapes from a Page object, a Master object, or a Shape object (group), check the Selection object's **ContainingPage**, **ContainingMaster**, and **ContainingShape** properties.

If the shapes are on a page, the **ContainingPage** property returns that Page object (and **ContainingMaster** returns Nothing). Conversely, if the shapes are in a master, **ContainingMaster** returns that Master object and **ContainingPage** returns Nothing.

If the shapes are in a group, the **ContainingShape** property returns a Shape object that represents the group. Otherwise, this property returns a Shape object that represents the page sheet of the master or page that contains the shapes.

Working with formulas

To work with a formula, you use the **Cells** property of a Shape object to get a Cell object. After you retrieve a Cell object, you can get or set the cell's formula or its value using the methods and properties of the Cell object.

You can alter a shape more dramatically by working with whole sections and rows of its formulas. For example, you can add Geometry sections, delete vertices, or change the row type of segments, converting them from lines to arcs or vice versa. For details, see "Modifying a shape's sections and rows" later in this chapter.

Getting a Cell object

To get a Cell object, use the **Cells** property of a Shape object and specify the cell name. You can use any valid cell reference with the Cells property.

For example, to get the PinX cell of a shape:

```
Dim shpObj as Visio.Shape
Dim pinXCellObj as Visio.Cell
...
Set pinXCellObj = shpObj.Cells("PinX")
```

To get the *y*-coordinate of the shape's fourth connection point:

```
Dim shpObj as Visio.Shape
Dim conYCellObj as Visio.Cell
...
Set conYCellObj = shpObj.Cells("Connections.Y4")
```

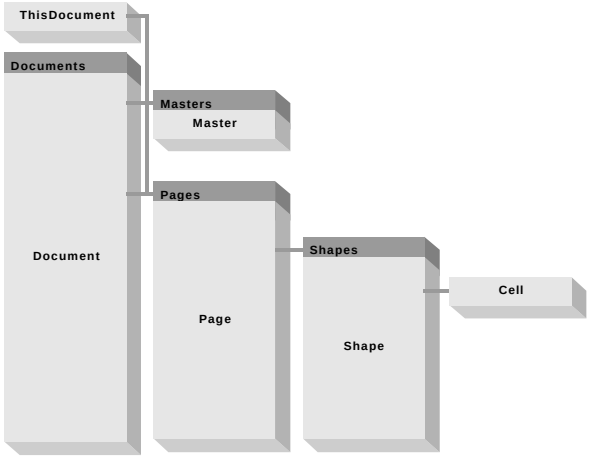
For a list of valid ShapeSheet cell names, see the online Visio Automation Reference.

Shortcuts for shape formulas

You can exercise fine control over a shape by setting shape formulas, but often changing a single characteristic can require setting more than one formula. For example, the location of a shape on a page or within a group is defined by two ShapeSheet cells—PinX and PinY.

You can change certain characteristics of shapes more easily by using the following methods and properties of Shape objects:

- **SetCenter** moves a shape's pin to the local coordinates you specify.
- **OneD** determines whether a shape behaves as a 1-D shape or a 2-D shape. If **OneD** is TRUE, the shape is 1-D; if **OneD** is FALSE, the shape is 2-D.
- **SetBegin** moves a 1-D shape's begin point to the parent (page or group) coordinates you specify.
- **SetEnd** moves a 1-D shape's end point to the parent coordinates you specify.



Cell object and related objects higher in the Visio object model

Getting a Cell object by section, row, and cell indexes. You can use the **CellsSRC** property to retrieve any cell by its section, row, and indexes. For a list of index constants you can use to access cells, see Appendix B, "ShapeSheet sections, cell references, and index constants."

For example, to get the Font cell in the first row of a shape's Character section:

```
Dim shpObj as Visio.Shape
Dim fontCellObj as Visio.Cell
...
Set fontCellObj = shpObj.CellsSRC _
(visSectionCharacter, visRowCharacter + 0, _
 visCharacterFont)
```

If a section contains more than one row and you want to refer to a cell in the second row or beyond, add an integer offset to the row constant for that section. Although you can use a row constant without an offset to get the first row of a section, it's good practice to use the row constant as a base and add an integer offset to it, starting with 0 for the first row. For example:

```
visRowScratch + 0 'First row of the Scratch section
visRowScratch + 1 'Second row of the Scratch section
visRowScratch + 2 'Third row of the Scratch section
```

The position of a section or row can change as a result of operations that affect other sections and rows. For example, if a Scratch section contains three rows and you delete the second row, the third row shifts to become the second row. **VisRowScratch + 2** is no longer a valid reference because the section no longer has a third row.

You can also use section and row indexes to add or delete sections or rows from a shape or to iterate through rows in a section. For details, see "Modifying a shape's sections and rows" later in this chapter.

Changing cell formulas

To change a cell's formula, set the **Formula** property of a Cell object to a string that is a valid formula for that cell. For example, to set the formula of a shape's LocPinX cell to = 2 * Width:

```
Dim shpObj as Visio.Shape
Dim cellObj as Visio.Cell
...
Set celObj = shpObj.Cells("LocPinX")
celObj.Formula = "2 * Width"
```

If you omit the equal sign from a formula string, Visio automatically adds it to the formula.

If the formula string contains quotation marks—for example, if inches are specified as " rather than *inches* or *in.*—use two quotation mark characters ("") to pass one quotation mark to Visio. For example, to set the formula of a shape's LocPinX cell to = 3 * 2.5":

```
celObj.Formula = "3 * 2.5"""
```

Replacing a formula with a result

Every cell has a formula, and every formula evaluates to a result. You can see this in the ShapeSheet window by choosing Formulas or Values from the View menu. If you're viewing formulas, a cell might display Width*0.5. If you're viewing values, and if Width is 5.0 in., the same cell would display 2.5 in.

Occasionally you may want to replace a formula with its result expressed as a constant, either to improve the performance of a shape, or because you no longer need to preserve its formulas. Visio evaluates formulas any time you retrieve a Cell object or make a change to a shape that affects its formulas. Depending on how often this occurs while your program is executing, it can have a noticeable effect on performance.

To replace a formula with its result, use the cell's **Result** property to set its formula. This is similar to setting the cell's **Formula** property—it's a shortcut for evaluating the formula and replacing it with the equivalent constant as the cell's new formula.

For example, suppose a shape's LocPinX formula is = 3 in. + 1 ft/2, which evaluates to 9 inches. To replace that formula with its result, use the following statement:

```
Dim cellLocPinX as Visio.Cell
...
celLocPinX.Result("inches") = _
celLocPinX.Result("inches")
```

After this statement executes, the LocPinX cell's formula is = 9 in.

The **Result** property returns a cell's result as a floating point number expressed in Visio internal units, but there are times when you might prefer a different data type:

- **ResultIU** returns the result as a floating point number expressed in Visio internal units.
- **ResultInt** returns the result as an integer.
- **ResultStr** returns the result as a string.

ResultStr takes a units argument like any other result method, effectively giving you a way to convert between any units. You might also use **ResultStr** to access cell formulas that contain strings, such as the Prompt cell in a custom property row.

Overriding guarded formulas

Visio has a GUARD function that protects a cell's formula from changes. If a cell's formula is protected with GUARD, attempting to set the formula with the **Formula**, **Result,** or **ResultIU** property causes an error. You can, however, change the cell's formula as follows:

- Use **ResultForce** or **ResultIUForce** instead of **Result** or **ResultIU.**
- Use **FormulaForce** instead of **Formula.**

Be cautious when overriding guarded formulas. Often a shape developer guards the formulas of a master shape to protect its smart behavior against inadvertent changes by a user. If you override these formulas, the shape may no longer behave as originally designed.

Sections and rows you can add

A shape can have only one of each kind of section except Geometry (represented by the constant **visSectionFirstComponent + n**). If a shape already has a particular section and you attempt to add it, you'll get an error. You can use the shape's **SectionExists** property to find out whether it has a section, then add it if necessary.

As an alternative, you can simply add a row. If the section doesn't already exist, it is created automatically. The row is added, and an error does not occur.

You cannot add or delete rows from the **visSectionCharacter**, **visSectionParagraph**, or **visSectionTab** sections.

Moving shapes by setting formulas: an example

The sample Visual Basic program NUDGE.EXE moves selected shapes in the active window by setting formulas for the pin of a 2-D shape, or the begin and end points for a 1-D shape. The program uses a user form with four buttons that call the **Nudge** subroutine with the parameters shown.

Nudge in \DVS\VB SOLUTIONS\NUDGE\NUDGE.FRM

```
Sub Nudge (dx As Double, dy As Double)

   'Call Nudge as follows:
   'Nudge 0, -1     Move down one unit
   'Nudge -1, 0     Move left one unit
   'Nudge 1, 0   Move right one unit
   'Nudge 0, 1   Move up one unit

   On Error GoTo lblErr
   Dim selObj As Visio.Selection
   Dim shpObj As Visio.Shape
   Dim unit As Double
   Dim i As Integer

   ' Establish a base unit as one inch
     unit = 1
   Set appVisio = GetObject(, "visio.application")
   Set selObj = appVisio.ActiveWindow.Selection

   ' If the selection is empty, there's nothing to do.
   ' Otherwise, move each object in the selection by the value of unit
     For i = 1 to selObj.Count
       Set shpObj = selObj(i)
       Debug.Print "Nudging " ; shpObj.Name; " ("; shpObj.NameID; ")"
       If (Not shpObj.OneD) Then
         shpObj.Cells("PinX").ResultIU = (dx * unit) + shpObj.Cells("PinX").ResultIU
         shpObj.Cells("PinY").ResultIU = (dy * unit) + shpObj.Cells("PinY").ResultIU
       Else
         shpObj.Cells("BeginX").ResultIU = (dx * unit) + shpObj.Cells("BeginX").ResultIU
         shpObj.Cells("BeginY").ResultIU = (dy * unit) + shpObj.Cells("BeginY").ResultIU
         shpObj.Cells("EndX").ResultIU = (dx * unit) + shpObj.Cells("EndX").ResultIU
         shpObj.Cells("EndY").ResultIU = (dy * unit) + shpObj.Cells("EndY").ResultIU
       EndIf
     Next i

   lblErr:
     Exit Sub

End Sub
```

Modifying a shape's sections and rows

The ShapeSheet window displays most of the formulas that define a shape, organized in sections such as Shape Transform, Geometry, and Connection Points. However, a shape may have more formulas than are shown in the ShapeSheet window. Certain other objects, such as drawing pages, masters, and styles, also have formulas.

You can change certain characteristics of a shape, or those of a page or master, by adding and deleting sections and rows. You can also iterate through sections or rows to perform the same operation on each item, such as listing all of a shape's Geometry formulas.

To refer to sections and rows in a program, you use constants defined in the Visio type library. For a list of these constants and how they correspond to sections and cells in the ShapeSheet window, see Appendix B, "ShapeSheet sections, cell references, and index constants."

Adding sections and rows

In many cases, you'll want to add an entire section to a shape. For example, you might add a Geometry section to create a shape with multiple paths, or a Scratch section to serve as a working area for building complex formulas. Before you can use a newly added section, you need to add at least one row to the section. Depending on the kind of row you add, you may also need to set the formulas of cells in the row.

To add a section, use the **AddSection** method for a Shape object. For example, to add a Scratch section to a shape:

```
Dim shpObj as Visio.Shape
...
shpObj.AddSection visSectionScratch
```

To add a row to a section, use the **AddRow** method and specify the section, row, and row tag. When you add a row to a Geometry section, the row tag indicates the type of row to add—for example, **visTagLineTo** indicates a LineTo row. For any other section, use a row tag of 0 as a placeholder. For example, to add a row to a Scratch section:

```
Dim shpObj as Visio.Shape
...
shpObj.AddRow visSectionScratch, visRowScratch + 0, 0
```

Sections and rows you can delete

You cannot delete the section **visSectionObject**, although you can delete rows within that section. You cannot delete rows in the following sections:

- **visSectionCharacter**
- **visSectionParagraph**
- **visSectionTab**

For best results, don't delete sections or rows that define fundamental characteristics of a shape, such as the last remaining Geometry section (**visSectionFirstComponent**) or the 1-D Endpoints row (**visRowXForm1D**), the component row (**visRowComponent**), or the MoveTo row (**visRowVertex + 0**) in a Geometry section.

Row tag constants are defined in the Visio type library. The following table lists row tags with the rows they represent in a Geometry section in the ShapeSheet window.

Row tag	Geometry row
visTagComponent	Display properties (NoFill and NoShow cells in a Start row)
visTagMoveTo	MoveTo row (X and Y cells in a Start row)
visTagLineTo	LineTo row
visTagArcTo	ArcTo row
visTagEllipticalArcTo	EllipticalArcTo row
visTagSplineBeg	SplineStart row
visTagSplineSpan	SplineKnot row

Deleting sections and rows

Deleting a section automatically deletes all of its rows and cells. To delete a section, use the **DeleteSection** method of a Shape object. For example, the following statement deletes the Scratch section of a shape:

```
Dim shpObj as Visio.Shape
...
shpObj.DeleteSection visSectionScratch
```

Deleting a nonexistent section does not cause an error.

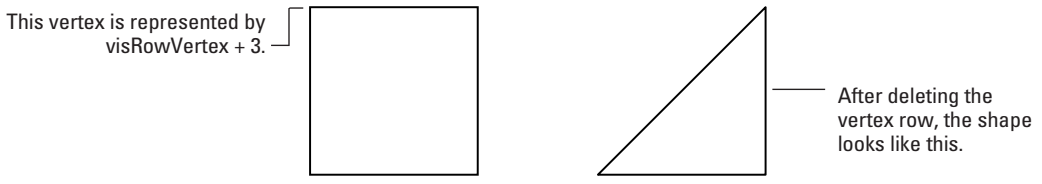
You can also delete a row from a section. For example, you can remove a vertex from a shape by deleting the row that defines the vertex from the shape's Geometry section. The following statement deletes the last vertex of a rectangle:

```
Dim shpObj as Visio.Shape
...
shpObj.DeleteRow visSectionFirstComponent + 0, _
visRowVertex + 3
```

Changing row types

Changing the row type of a Display Properties row (**visRowComponent**) or a MoveTo row (**visRowVertex + 0**) is not recommended—it can cause a shape to behave in unexpected ways.

Changing line or arc segments to spline rows or vice versa requires an understanding of how splines are defined in the shape's Geometry section. For details, see Appendix A, "Arcs and splines in Visio."



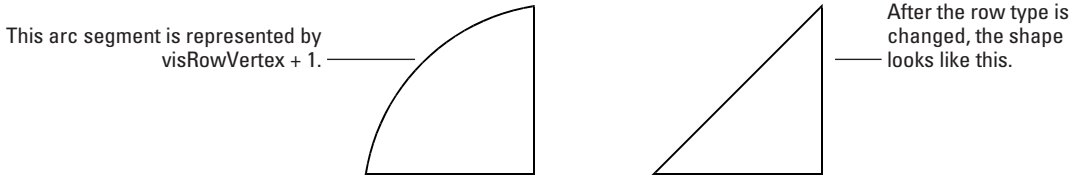
Deleting a vertex row

Changing the type of a segment

In Visio, you can define a segment as a line, arc, elliptical arc, or spline by setting the type of row or rows that represent the segment. From a program, you can do this by setting the **RowType** property of a Shape object.

For example, the following statement converts the first segment of a shape to a line segment.

```
shpObj.RowType(visSectionFirstComponent + 0, _
visRowVertex + 1) = visTagLineTo
```



Changing the row type of a vertex row

Working with inherited data

A shape may not have a local copy of all the data that appears in the ShapeSheet window or that you can access from a program. The shape behaves as if the data were local, but some data may be local and other data may be inherited from a master or a style.

Everything you do when adding, modifying, or deleting data is done locally. If the data doesn't exist locally—that is, if the shape inherits the data from a master or style—Visio first creates a local copy of the data, then performs the requested action. Once data exists locally, inheritance is severed, and changes to the data in the master no longer affect the shape's local data.

For example, if a shape is an instance of a master that has connection points, the shape inherits the master's connection points. The shape has the same connection point behavior as the master and will display the inherited Connection Points section in its ShapeSheet window. However, the shape doesn't have a local copy of the Connection Points data—instead, it inherits that data from the master. If you attempt to delete this shape's Connection Points section, the Connection Points data doesn't change because there is no local copy to delete, and the shape continues to inherit its Connection Points data from the master.

To override inheritance for an entire section, either delete each row in the section, or delete the entire section and add a new, empty section of the same type. In the latter case, you delete the section to make sure no local copy already exists, which would cause an error if you attempted to add the section.

To restore an inherited section, delete the local copy. The shape inherits that section again from the master or style.

Iterating through sections and rows

You can perform the same operation on multiple sections or rows by iterating through them, using the following Shape object properties to limit the iteration loop:

- **GeometryCount** represents the number of Geometry sections for a shape.
- **RowCount** represents the number of rows in a section. This includes "hidden" rows such as row 0 of a Geometry section. Use the row constant as the base and add an integer offset, starting with 0 (zero).
- **RowsCellCount** represents the number of cells in a row.

The following example iterates through the rows and cells in a shape's Geometry section and uses **CellsSRC** to retrieve each cell. It then displays each cell's formula in a list box on a user form.

IterateGeometry in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub IterateGeometry ()

   'This example assumes the active page contains a shape.
   Dim shpObj As Visio.Shape         ' shape object
   Dim curGeomSect As Integer        ' Section number for accessing geometry section
   Dim curGeomSectIndx As Integer    ' Loop variable for geometry sections
   Dim nRows As Integer              ' number of rows in section
   Dim nCells As Integer             ' number of cells in row
   Dim curRow As Integer             ' current row number (0 based)
   Dim curCell As Integer            ' current cell index (0 based)
   Dim nSects As Integer             ' number of geometry sections in shape

   Set shpObj = ActivePage.Shapes(1)
   UserForm2.ListBox1.Clear
   nSects = shpObj.GeometryCount
   For curGeomSectIndx = 0 To nSects - 1
      curGeomSect = visSectionFirstComponent + curGeomSectIndx
      nRows = shpObj.RowCount(curGeomSect)
      For curRow = 0 To (nRows - 1)
         nCells = shpObj.RowsCellCount(curGeomSect, curRow)
         For curCell = 0 To (nCells - 1)
            UserForm2.ListBox1.AddItem _
            shpObj.CellsSRC(curGeomSect, curRow, curCell).LocalName & _
            ": " & shpObj.CellsSRC(curGeomSect, curRow, curCell).Formula
         Next curCell
      Next curRow
   Next curGeomSectIndx
   UserForm2.Show

End Sub
```

You can also use logical position constants to set the beginning or end of a loop. These constants are most useful when you want to perform the same operation on all sections, rows, or cells, but the order in which the operation is performed is not important. (If the order in which an operation is performed *is* important, use a row constant with an offset rather than a logical position constant.) For example, you might want to print all the formulas for a shape. For a list of logical position constants, see Appendix B, "ShapeSheet sections, cell references, and index constants."

Adding a Geometry section to a shape: an example

A basic shape in Visio consists of one or more components, or paths. Each path is a sequence of connected segments. In most shapes, a segment is either a line segment or an arc segment, which can be a circular or an elliptical arc. Each path is represented by a Geometry section, and each segment is represented by a row in a Geometry section.

To add a Geometry section to a shape, use the **AddSection** method with **visSectionFirstComponent** to insert the section before existing Geometry sections, or **visSectionLastComponent** to append the section after existing Geometry sections. For example:

```
Dim shpObj as Visio.Shape
...
shpObj.AddSection visSectionLastComponent
```

After adding a Geometry section, you must add at least two rows. (Visio does not automatically add rows to a section added from a program.) Use the **AddRow** method with the following row tags:

- **visTagComponent** determines whether the component defined by the Geometry section can be filled and whether it is hidden or visible.
- **visTagMoveTo** determines the first vertex, or starting point, of the component.

You can add additional vertex rows using the row tags **visLineTo**, **visArcTo**, or **visEllipticalArcTo**. Each vertex row defines the local coordinates of a vertex and the type of segment—line, circular arc, or elliptical arc—that connects the vertex with the previous one.

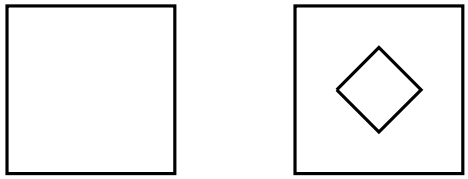
You can add spline rows using the row tags **visTagSplineBeg** and **visTagSplineSpan**. Precede the spline start row (**visTagSplineBeg**) with a start row (**visTagMoveTo**) or a vertex row, and use **visTagSplineSpan** to add spline knot rows. For details about spline rows, see Appendix A, "Arcs and splines in Visio."

The following example inserts a Geometry section before other existing Geometry sections of a shape. It adds the component row, the MoveTo row, and one LineTo row. (These are the rows you typically need to define a straight line.) It then sets cell formulas in each row to draw the line diagonally across the shape's width-height box.

AddGeometry in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub AddGeometry ()

  Dim shpObj as Visio.Shape
  Dim iSection as Integer
  Dim i as Integer

  'Set an error handler to catch the error if no shape is selected.
  On Error GoTo errNoShp
  Set shpObj = ActiveWindow.Selection(1)
  On Error GoTo 0
  iSection = shpObj.AddSection(visSectionFirstComponent)
  shpObj.AddRow iSection, visRowFirst + 0, visTagComponent
  shpObj.AddRow iSection, visRowVertex + 0, visTagMoveTo
  For i = 1 To 4
    shpObj.AddRow iSection, visRowVertex + i, visTagLineTo
  Next i
  shpObj.CellsSRC(iSection, visRowVertex + 0, visX).Formula = "Width * 0.25"
  shpObj.CellsSRC(iSection, visRowVertex + 0, visY).Formula = "Height * 0.5"
  shpObj.CellsSRC(iSection, visRowVertex + 1, visX).Formula = "Width * 0.5"
  shpObj.CellsSRC(iSection, visRowVertex + 1, visY).Formula = "Height * 0.25"
  shpObj.CellsSRC(iSection, visRowVertex + 2, visX).Formula = "Width * 0.75"
  shpObj.CellsSRC(iSection, visRowVertex + 2, visY).Formula = "Height * 0.5"
  shpObj.CellsSRC(iSection, visRowVertex + 3, visX).Formula = "Width * 0.5"
  shpObj.CellsSRC(iSection, visRowVertex + 3, visY).Formula = "Height * 0.75"
  shpObj.CellsSRC(iSection, visRowVertex + 4, visX).Formula = "Geometry1.X1"
  shpObj.CellsSRC(iSection, visRowVertex + 4, visY).Formula = "Geometry1.Y1"
  'Exit the procedure bypassing the error handler
  Exit Sub
  errNoShp:
  MsgBox "Please select a shape then try again.", vbOKOnly, DVS_TITLE

End Sub
```

The following illustration shows the shape before and after inserting the Geometry section.



Inserting a Geometry section in a shape

Note that **visRowVertex** with no offset, or with an offset of 0, refers to the MoveTo row. When adding vertex rows, always add an offset of 1 or more to **visRowVertex** so you don't inadvertently replace the MoveTo row, which can cause the shape to behave in unexpected ways.

For details about Geometry cells and the formulas they can contain, search online help for "geometry section."

Associating data with shapes

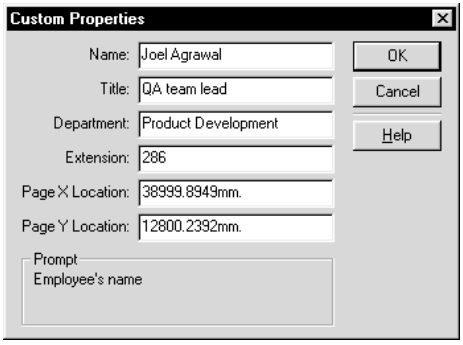
If you're developing a solution that combines Visio with a database, you'll be interested in ways you can associate data with shapes. Visio shapes and pages can store additional data in user-defined cells or as custom properties. Shapes and masters can have unique IDs you can use to distinguish identical shapes in a drawing or to track the original source of a master.

For details about creating user-defined cells and custom properties in Visio, see Chapter 4, "Enhancing shape behavior."

Working with user-defined cells and custom properties

User-defined cells are a convenient way for a program to store a value in a cell and reliably find it again. User-defined cells are preferable to Scratch cells for this purpose, because other programs can write to the same Scratch cell, destroying its value for your program, or they can add or delete Scratch rows, making your Scratch cell reference invalid. If your user-defined cells have names that are likely to be unique to your program (for example, names prefixed with your company acronym or another identifier), other programs are unlikely to inadvertently use your cells, and they can add or delete other user-defined cells without affecting yours.

Custom properties are a way to associate database-like fields with a shape or a page. Although you can get and set custom properties exclusively from a program, you'll often collect values filled in by the user in Visio. For example, you might provide masters that prompt the user to fill in certain data when a master is dropped in a drawing and then use a program to gather that data from the user's drawings.



Custom Properties dialog box: an example

Adding user-defined and custom property rows. To add a user-defined or custom property row from a program, use the **AddNamedRow** method of a Shape object. For example, to add a user-defined cell named Latitude to a page:

```
Dim pagObj as Visio.Page
Dim shpObj As Visio.Shape
...
set shpobj = pagObj.Shapes("Germany")
shpObj.AddNamedRow visSectionUser, "Latitude", 0
```

How unique is a unique ID?

If a shape or master has a unique ID, you can assume that no other shape or master in the same document has the same unique ID. Visio generates unique IDs using the same technology that OLE uses to guarantee unique object IDs and never reuses them, so the chance of Visio generating duplicate unique IDs is extremely remote, even on different systems.

Under certain circumstances, however, it is possible to duplicate a unique ID. If you copy a drawing file or save it under a different file name, all of its shapes and their unique IDs are copied. If you then copy a shape from the new file and paste it back into the original, the original file contains two shapes with identical unique IDs. To ensure unique IDs after copying a file, delete all the unique IDs from one of the files and generate new ones.

If you copy a master, the copy has the same unique ID as the original until you edit the copy—then Visio assigns a different unique ID to the edited master.

To get the Value cell of the new Latitude row:

```
Dim shpObj as Visio.Shape
Dim celObj as Visio.Cell
...
Set celObj = shpObj.Cells("User.Latitude")
```

For details about adding sections and rows to a shape or page, see "Modifying a shape's sections and rows" earlier in this chapter.

Generating and using unique IDs

Unique IDs are typically used to create a persistent link between a shape or master and a record in an external database. Unique IDs allow applications to bind data to shapes more reliably than is possible with shape names and name IDs. Database applications that can use unique IDs include facilities management, geographic information systems, and mechanical assemblies, in which identical shapes in a drawing may represent different records in a database.

A unique ID is stored internally as a 128-bit value and is passed as a null-terminated 39-character string, formatted as in the following example:

```
{2287DC42-B167-11CE-88E9-0020AFDDD917}
```

Unique IDs for shapes. By default, shapes do not have unique IDs; they must be generated by a program. To generate a unique ID for a shape, use the **UniqueID** method of a Shape object. For example:

```
Dim shpObj as Visio.Shape
...
IDString = shpObj.UniqueID(visGetOrMakeGUID)
```

If the shape already has a unique ID, this statement gets the ID; if the shape does not have a unique ID, the statement creates one.

To find out whether a shape has a unique ID, use the following statement. If the shape has no unique ID, this statement returns a null string ("").

```
Dim shpObj as Visio.Shape
...
IDString = shpObj.UniqueID(visGetGUID)
```

To delete a shape's unique ID, use the following statement:

```
Dim shpObj as Visio.Shape
...
shpObj.UniqueID visDeleteGUID
```

Some actions cause Visio to delete a shape's unique ID automatically. If you cut a shape to the Clipboard and paste it once, or drag a shape to a different drawing window, its unique ID is preserved. However, if you paste the same shape from the Clipboard a second time or duplicate the shape by holding down the Ctrl key and dragging, its unique ID is deleted.

Unique IDs for masters. A master always has a unique ID that is generated by Visio and cannot be deleted or reassigned. A master's unique ID provides a link to the original master, because copies of masters retain their original unique IDs, making them more persistent than names.

For example, when you drop a master from a standalone stencil into a drawing file, a copy of that master is placed in the drawing file stencil. If you then drop a master with the same name from a different stencil into the same drawing file, Visio alters the second master's name so that names are unique within the drawing file stencil. The two masters still have different unique IDs, so you can distinguish them from each other this way also.

A master's unique ID changes only when you edit the master—then Visio assigns the master a new unique ID.

Getting a shape or master by its unique ID. You can pass a unique ID as an argument to the **Item** method of a Shapes or Masters collection. For example:

```
Dim shpsObj as Visio.Shapes
Dim shpObj as Visio.Shape
...
Set shpObj = shpsObj.Item("{2287DC42-B167-11CE- _
88E9-0020AFDDD917}")
```

15

Handling events in Visio

An *event* is something that happens. In Visio, events happen as a result of a user's actions. For example, the user may open or close documents, drop or delete shapes on the drawing page, edit the text of shapes, or alter shape formulas. Knowing that such events have occurred can be extremely useful, because it allows your solution to handle user actions that can otherwise be difficult to predict.

This chapter describes how to handle Visio events in these ways:

- Defining a cell formula in a shape's Events section.
- Writing code behind the event in the Visual Basic for Applications (VBA) project of a Visio document. This allows a solution to handle events related to documents, such as saving a document.
- Creating Visio Event objects that run programs when specified events occur.
- Creating Visio Event objects that send notifications to special-purpose objects (called *notification sinks*). This enables two-way communication between a standalone solution and Visio.

This chapter describes handling events in VBA or Visual Basic (VB) programs. For information about handling events in C++ programs, see Chapter 20, "Programming Visio with C++."

Topics in this chapter

Handling events with ShapeSheet formulas

You can define how a shape responds to a few user actions by writing event formulas. An event formula typically performs an action in response to the event, such as running a macro or add-on, or navigating to another drawing page. Whenever the user performs one of these actions, the formula in the corresponding cell is evaluated and the action is performed.

Events that trigger formulas

Visio supports five events that trigger formulas in the following cells in the shape's Events section.

Events section cells

Events cell	Event that triggers the formula
TheText	The shape's text or formatting is changed.
EventXFMod	The shape's position, size, or orientation on the page is changed.
EventDblClick	The shape is double-clicked.
EventDrop	A new instance is created by pasting, duplicating, or Ctrl+dragging a shape, or by dragging and dropping a master.
TheData	Reserved for future use.

By entering formulas in these cells, you define how the shape responds to each event. Unlike most formulas, event formulas are evaluated only when the event happens, not when you enter the formula or when cells referenced by the formula change. This means that Events cells behave somewhat differently than other ShapeSheet cells:

- The value displayed in an Events cell may appear to be out of date or inconsistent with the cell's formula. For example, suppose you entered this formula in the EventDblClick cell:

  ```
  = Width > 1 in.
  ```

  This formula returns 1 if the expression is TRUE, or 0 if it is FALSE. However, the formula is evaluated when the user double-clicks the shape, not when the shape is resized, so the value displayed in the cell may not indicate what is true for the shape.

About the Double-Click command

The Visio Double-Click command creates event formulas that perform various actions when the user double-clicks a shape. Visio stores the event formula in the shape's EventDblClick cell. Experiment with Double-Click to learn about event formulas or to create formulas you can modify.

- The order of evaluation and the number of times an event is evaluated are unpredictable. For example, if a shape's text is formatted and the shape is resized, the order in which these events trigger evaluation of event formulas is unknown. However, each event triggers its formula at least once.

Functions for event formulas

Visio includes several built-in functions that perform actions rather than produce a value, making them especially useful in event formulas. For details about function syntax, search online help for "functions."

- CALLTHIS("procedure",["project"],[arg1,arg2,…])

  Calls a procedure in a VBA project and passes the procedure a reference to the formula's shape. For example:

  ```
  CALLTHIS("myProc",,Height,Width)
  ```

- GOTOPAGE("pagename")

  Displays the page in the currently active window. For example:

  ```
  GOTOPAGE("Page-2")
  ```

  A page's name is displayed in the title bar of its window and in the Page dialog box (Go To submenu). GOTOPAGE can also be used with a URL to display a Web site.

- OPENFILE("filename")

  Opens a file in a new window.

- OPENSHEETWIN()

  Opens and displays the ShapeSheet window that contains this formula.

- OPENTEXTWIN()

  Opens the text block for the shape that contains this formula.

- PLAYSOUND("filename"|"alias",isAlias,beep,synch)

  On systems with a sound card, plays the sound recorded in *filename*, or plays the system alias for a sound if *isAlias* is a nonzero number. If the sound cannot be played, Visio can beep to indicate an error. Sounds can be played asynchronously or synchronously.

Using event formulas

Because event formulas are evaluated each time the event occurs, they can affect the usability and performance of your shapes. In general:

Use event formulas sparingly. Avoid event formulas for frequent events such as moving or sizing the shape (**EventXFMod**) or editing its text (**TheText**). Handling these events can interrupt the user's work flow and make shapes awkward to use.

Keep event formulas simple. A complex formula takes longer to evaluate, which slows the performance of the shape.

Know that some actions take longer than others. Even a simple event formula may trigger a time-consuming action. For example, it takes longer to launch a standalone executable add-on than to run a macro, and longer to navigate to a Web page than to navigate to another page in the same Visio document.

Simulating events with DEPENDSON

DEPENDSON(cellref[,cellref2,...]) creates a cell reference dependency. This function has no effect in an Events section cell, but you can use it to simulate events in other ShapeSheet sections such as the Scratch section. For example, if you put the following formula in a Scratch cell, Visio opens the shape's text block whenever the shape is moved:

```
OPENTEXTWIN() +
DEPENDSON(PinX, PinY)
```

For another example, if you put the following formula in a Scratch cell, the add-on MYPROG.EXE is launched whenever the shape is flipped in either direction.

```
RUNADDON("myprog.exe") +
DEPENDSON(FlipX, FlipY)
```

For example:

```
PLAYSOUND("chord.wav", 0, 0, 0)
```

plays the wave audio file CHORD.WAV asynchronously with no warning beep.

- RUNADDON("name")

  Launches the specified macro or add-on. To pass arguments to a macro, include them in the string. To pass arguments to an add-on, use the RUNADDONWARGS function.

- SETF("cell", formula)

  Sets the formula of a cell in the same or another shape. For example:

  ```
  SETF("Scratch.A1", Scratch.A1+1)
  ```

  evaluates the formula =Scratch.A1+1 and sets the formula of the Scratch.A1 cell to the result, which is its previous value incremented by 1.

Writing code behind events

If you've written any Visual Basic code, you've almost certainly written event procedures. An *event procedure* contains code that is executed when an event occurs. For example, a button on a Visual Basic form usually has an event procedure to handle the Click event. In VBA, this is called *code behind events*.

You can handle certain Visio events by putting code behind them in the VBA project of a Visio document. For example, the following program handles two events, **DocumentCreated** and **ShapeAdded**, to keep count of shapes added to a drawing that are based on a master called Square:

- The **DocumentCreated** event handler runs when a new drawing is based on the template that contains this code. The handler initializes an integer variable, *nSquares*, which is used to store the count.

- The **ShapeAdded** event handler runs each time a shape is added to the drawing page, whether the shape is dropped from a stencil, drawn with a drawing tool, or pasted from the Clipboard. The handler checks the **Master** property of the new shape and, if the shape is based on the Square master, increments *nSquares*.

Code behind ThisDocument in \DVS\VBA SOLUTIONS\VBA EVENT SAMPLE.VST

```
' Number of squares added to drawing
Dim nSquares As Integer

Private Sub Document_DocumentCreated()
  ' Initialize number of squares added
  nSquares = 0
End Sub

Private Sub Document_ShapeAdded( ByVal Shape As Visio.IVShape )

  Dim mastObj As Master

  ' Get the Master property of the shape
  Set mastObj = Shape.Master

  ' Check whether the shape has a master. If not, the shape was created locally.
  If Not ( mastObj Is Nothing ) Then
    ' Check whether the master is "Square"
    If mastObj.Name = "Square" Then
      ' Increment the count for the number of squares added
      nSquares = nSquares + 1
    End If
  End If

  MsgBox "Number of squares: " & nSquares, vbInformation, "Developing Visio Solutions"

End Sub
```

To put code behind an event:

1. Open the VBA Project Editor. (From the Visio Tools menu, choose Macro, then choose Visual Basic Editor.)
2. In the Project Explorer, double-click the object for which you want to put code behind an event—for example, ThisDocument.

   If the object doesn't appear in the Project Explorer, you may need to open the drawing's object folder. Double-click the drawing name, then double-click the folder named Visio Objects.

When you double-click the object, Visio opens the Visual Basic code window and creates an empty procedure for the object's default event—for **ThisDocument**, the default event is **DocumentCreated**.

3. Choose the event you want to handle from the Procedure list at the top right of the code window—for example, **DocumentSaved**.

   Visio creates an empty event procedure for that event.

4. Fill in the event procedure with the code you want to execute when the event occurs.

For details about using the VBA Project Editor in Visio, see Chapter 2, "Tools for creating solutions."

Declaring a variable "with events"

You can use the VBA keyword **WithEvents** to declare an object variable for the Visio object whose events you want to handle within a class module or the Visio Document object in your VBA project. For example:

```
Private WithEvents m_winObj as Visio.Window
```

In addition to the usual access to an object's properties and methods, this declaration gives the object variable the capacity to handle events fired by the object assigned to that variable. When you select the variable in the Object list in the Visual Basic Editor, the Procedure list shows the events that can be fired by that object. When you choose an event from the Procedure list, VBA creates an empty event procedure that you can fill in with code to handle the event.

For example, the following event procedure prints the names of selected shapes in the Debug window whenever the selection changes in the window represented by m_winObj:

```
Public Sub m_winObj_SelectionChanged(ByVal _
Selection as Visio.IVSelection)
  Dim i as Integer
  For i = 1 to Selection.Count
    Debug.Print Selection(i).Name
  Next i
End Sub
```

Handling events for multiple objects

You can use the **WithEvents** keyword in a class module to create specialized event handlers for any number of shapes of a particular type. For example, in an architectural floor plan, you might want to handle events from each window shape that is dropped in the drawing, to monitor whether a window is positioned appropriately in a wall if the window is moved. Or, you might want to handle events from each wall shape, to monitor whether a wall has been moved or changed.

To create such event handlers, write a **ShapeAdded** event handler that creates an instance of your class module and assigns a reference to the newly added shape to a variable defined **WithEvents** inside the class module—for example, m_shpObj. Then, write event procedures for the events you want to handle for m_shpObj—for example, m_shpObj_CellChanged.

You can use this technique to set up any number of event handlers for any type of object whose number can vary at run time, such as pages and windows as well as shapes. For an example, see VBA WITHEVENTS SAMPLE.VSD on the Visio 5.0 CD.

To assign a window to m_winObj, use a statement such as the following:

```
Set m_winObj = Visio.ActiveWindow
```

Because this statement must run before the **SelectionChanged** event procedure, you might put this and similar statements in a public subroutine, which you call from an event procedure for an event that you know will execute before **SelectionChanged**, such as **DocumentOpened**.

Handling events with a sink object

You can streamline the process of handling events fired by a particular kind of Visio object by defining a class to receive the events. A class that receives events is sometimes called an *event sink* or a *sink object*.

For example, the following class module defines a sink class called *ShapeSink* that declares the object variable *m_shpObj* using the **WithEvents** keyword. It contains a procedure, *InitWith*, that assigns a particular Shape object, *aShape*, to *m_shpObj*. The class module also contains an event handler for the **CellChanged** event, which can be fired by a Shape object—in this case, the Shape object represented by *aShape*.

ShapeSink class module in \DVS\VBA SOLUTIONS\VBA WITHEVENTS SAMPLE.VSD

```
Dim WithEvents m_shpObj As Visio.Shape

Public Sub InitWith(ByVal aShape As Visio.Shape)
   Set m_shpObj = aShape
End Sub

Private Sub m_shpObj_CellChanged(ByVal Cell As Visio.IVCell)
   Debug.Print Cell.Shape.Name & " " & Cell.Name & " changed to =" & Cell.Formula
End Sub
```

Actions in the drawing window, such as moving, sizing, or connecting a shape, can change a shape's formulas. The **CellChanged** event fires once for each formula that changes. This event handler responds by listing the name of the Shape object, the name of the cell, and the new formula in the VBA Immediate window. (If the Immediate window is not visible, choose Immediate Window from the View menu in the Visual Basic Editor to see the output of the **Debug.Print** statement.)

To put this event handler to work, the program must create an instance of the sink class and pass it a reference to a Shape object. You can do this in whatever way makes sense for your program, but this example happens to use code behind the **ShapeAdded** event in the document. This example also uses a collection to manage the sink objects so it can release a sink object when its corresponding shape is deleted. For details about collections, see your VBA documentation.

ShapeAdded event handler in ThisDocument in \DVS\VBA SOLUTIONS\VBA WITHEVENTS SAMPLE.VSD

```
Private Sub Document_ShapeAdded(ByVal Shape As Visio.IVShape)
  Dim sinkObj As New ShapeSink
  sinkObj.InitWith Shape
  sinks.Add sinkObj, Str(Shape.ID)
End Sub
```

When a shape is added to the drawing page, the **ShapeAdded** handler creates a new *ShapeSink* object and assigns it to the variable *sinkObj*. Next, it calls the object's *InitWith* procedure with a reference to the newly added shape, which associates *sinkObj* with the new shape. Whenever any formula of the shape changes, *sinkObj*'s **CellChanged** event handler runs. This situation persists until either *sinkObj* is released or another Shape object is assigned to the *m_shpObj* variable of *sinkObj*, which causes the first object to stop firing events and the second object to start.

Handling events with Event objects

In earlier chapters, you've seen how to control Visio objects by using Automation to get and set properties and to invoke methods. This one-way communication has its limitations: Your program can tell Visio what to do, but it cannot find out what is happening in Visio without explicitly checking for each possible case.

You can handle Visio events from a standalone Visual Basic, C, or C++ program by using *Event objects*. An Event object pairs an event with an action—either to run an add-on or to notify an object in your program that the event occurred. When the event occurs, the Event object fires, triggering its action.

When you create an Event object, you need to decide:

- The scope in which the Event object should fire. The scope determines the object whose EventList collection the Event object is added to.
- The action to perform when the event occurs—run an add-on or send a notification to an already running program. The action determines which method you use to create the Event object.
- The event or events that should trigger the action. This determines the event code you specify when you create the Event object.

If the event's action is to send a notification, you must also tell Visio which object to notify.

Deciding the scope of an event

An event has both a subject and a source, which are typically different objects. The *subject* of an event is the object to which the event actually happens. For example, the subject of a **ShapeAdded** event is the shape that was added.

The *source* of an event is the object that produces the event. Most events have several potential sources. You create an Event object by adding it to the EventList collection of the source object, so any object that has an EventList collection can be a source of events.

The source object you choose determines the scope in which the event fires—the higher the source object in the object hierarchy, the greater the scope. For example, if the source is a Page object, the **ShapesAdded** event fires whenever a shape is added to that page. If the source is the Application object, the **ShapesAdded** event fires whenever a shape is added to any page of any document that is open in the instance of Visio.

Visio doesn't act on an event unless it has an Event object for it. Obviously, the more often an Event object fires, the more likely it is to affect the performance of your solution. Therefore, when you pick a source object, think first about the scope in which you want to handle the event. Then add the Event object to the EventList collection of the *lowest* object in the hierarchy that can fire the Event object in the scope you want.

Sources of events in Visio

In Visio 5.0, the following objects can be sources of events:

Application	Selection
Cell	Shape
Characters	Shapes
Document	Style
Documents	Styles
Master	Window
Masters	Windows
Page	
Pages	

For a list of the events that can be produced by a particular source object, use the Object Browser in the Visual Basic Editor. Events are marked with a lightning bolt. Or, in the Visio Automation Reference, you can search for the object, or search for a particular event by name.

If the Event object's action is to send a notification, both the source and subject objects are passed to the event procedure in the event sink object.

Deciding the action to perform

After you've decided what the source object should be, you can create the Event object by adding it to the EventList collection of the source object. The action you want to trigger determines which method you use:

- Run an add-on or other external program. To create an Event object for this action, you use the **Add** method of the source object's EventList collection.
- Call an event procedure of another object in your program. To create an Event object for this action, you use the **AddAdvise** method of the source object's EventList collection.

Indicating the event code

You indicate the event you're interested in by supplying its event code to the **Add** or **AddAdvise** method. Event codes are prefixed with **visEvt** in the Visio type library.

In some cases, an event code is a combination of two or more codes. For example, the event code for the **ShapeAdded** event is **visEvtAdd + visEvtShape**. The event code of **PageAdded** is **visEvtAdd + visEvtPage**.

In some cases you can combine codes to indicate an interest in multiple events with a single Event object. For example, the event code **visEvtAdd + visEvtPage + visEvtShape** indicates that you're interested in both **ShapeAdded** and **PageAdded**. The event code **visEvtAdd + visEvtDel + visEvtPage** indicates that you're interested in both **PageAdded** and **PageDeleted**.

When an Event object fires, Visio passes the event code for the event that actually occurred, even if the Event object's event code indicates multiple events. To continue the last example, if a page is added, Visio passes the event code **visEvtAdd + visEvtPage**.

Event codes and the Visio type library

Although event codes appear with other Visio constants in the Visio type library, some cannot be used from the type library because of the way it handles large hexadecimal values. Any constant with the high bit set (that is, a constant with a hexadecimal value greater than &H7FFF, or 32,767) causes a numeric overflow condition, and some Visio event codes happen to fall in this range.

You can prevent the overflow condition by including VISCONST.BAS in your project. The Visual Basic interpreter checks for constant definitions in all available modules before it checks the type library, so if VISCONST.BAS is present it will provide definitions of event codes, and the numeric overflow condition will not occur. For details about importing a file into a VBA project, see "Programming Visio with VBA" in Chapter 2, "Tools for creating solutions." For more information about VISCONST.BAS, see Chapter 19, "Programming Visio with Visual Basic."

Creating an Event object that runs an add-on

When you create an Event object that runs an add-on, you supply the event code for the event or events that you're interested in and the action code **visActCodeRunAddon**. You also supply the name of the add-on to run and, optionally, a string of arguments to pass to the add-on when the Event object fires.

When the Event object fires, Visio passes the argument string as command line arguments if the add-on is an .EXE file, or as the **lpCmdLineArgs** field of the **VAOV2LSTRUCT** structure passed to an add-on implemented by a Visio library (.VSL). For details about Visio libraries, see Chapter 20, "Programming Visio with C++."

For example, the following code creates an Event object that runs the add-on SHOWARGS.EXE and passes the string *"/args=Shape Added!"* as a command line argument. The Event object is added to the EventList collection of the document.

FormLoad in \DVS\VB SOLUTIONS\VB EVENT SAMPLE.FRM

```
Dim docObj As Visio.Document

Private Sub Form_Load()
  Dim eventsObj As Visio.EventList
  ...
  ' Create a new drawing.
  ' An instance of Visio has already been assigned to g_appVisio.
  Set docObj = g_appVisio.Documents.Add("")

  ' Get the EventList collection of this document.
  Set eventsObj = docObj.EventList

  ' Add an Event object that will run an add-on when the event fires.
  eventsObj.Add visEvtShape + visEvtAdd, visActCodeRunAddon, _
    "SHOWARGS.EXE", "/args=Shape added!"
...
End Sub
```

When a shape is added to any page in the document, the **ShapeAdded** event fires and triggers the action, which is to run the add-on SHOWARGS.EXE, whose sole purpose is to display its command line arguments—in this case, the string "Shape added!".

Storing an Event object with a document. Certain source objects can store certain Event objects with a Visio document. This is sometimes called *persisting an event*. An Event object can be stored with a document if it meets the following conditions:

- The Event object's action must be to run an add-on. Event objects that send notifications cannot be stored. If an Event object can be stored, its **Persistable** property is TRUE.
- The source object must be able to persist the event. In the Visio version 5.0 product line, Document, Page, and Master objects can do this. If a source object can persist events, its **PersistsEvents** property is TRUE.

Whether a persistable Event object actually persists depends on the setting of its **Persistent** property. If the Event object is persistable, Visio assumes that it should be stored with the document, so the initial value of its **Persistent** property is TRUE. If you do not want Visio to store the Event object, set its **Persistent** property to FALSE.

NOTE Before you attempt to change an Event object's **Persistent** property, make sure its **Persistable** property is TRUE. Setting the **Persistent** property of a nonpersistable event causes an error.

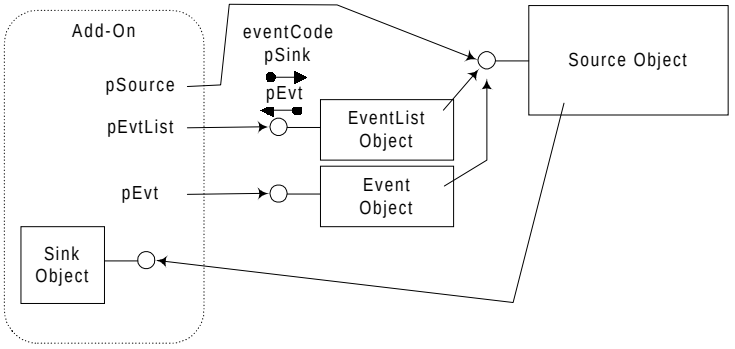
Creating an Event object that sends a notification

An Event object can send a notification to an already running program. Creating this kind of Event object differs from creating one that simply runs an add-on in these ways:

- You define an object in your program—not a Visio object—to receive the notification when it is sent. This kind of object is sometimes called a *notification sink* or *sink object*.
- You write an event procedure in your sink object to handle notifications when they are received.
- Your program creates instances of sink objects and the Event objects in Visio at runtime. Because this kind of Event object uses references, it cannot be stored with a Visio document and must be created each time the program runs.

NOTE If you're writing an external program in Visual Basic, you must use Visual Basic 4.0 or later to receive event notifications from Visio.

The following diagram shows how a program interacts with objects in Visio to receive event notifications.



In this diagram, *pSource* is a reference to the source object in Visio. This is used to get a reference to the source object's EventList collection, which is assigned to *pEvtList*.

The program uses *pEvtList.AddAdvise* to create the Event object, which is assigned to *pEvt*. With **AddAdvise**, the program passes a reference to the sink object to which Visio sends the notification when the Event object fires.

The connection between the source object and the sink object lasts until the program calls *pEvt.Delete,* until the program releases its last reference on the source object, or until Visio terminates. When Visio terminates, it issues a **BeforeQuit** event, which the program should handle by releasing all of its references to source objects. After Visio issues **BeforeQuit**, it releases all of its references to sink objects in the program.

Defining the sink object. A sink object is a non-Visio object you define to receive the notifications that Visio sends. At a minimum, the sink object must be programmable (that is, it must support the OLE **IDispatch** interface) and it must expose an event procedure named **visEventProc**, declared exactly as shown on the next page. (You can give the sink object whatever additional functionality makes sense for your program, but Visio requires only **visEventProc**.) When an Event object fires, Visio calls the **visEventProc** procedure for the corresponding sink object.

Typical events

Here's a partial list of the events supported by Visio 5.0:

AppActivated
BeforeDocumentClose
BeforePageDelete
BeforeQuit
BeforeSelectionDelete
BeforeWindowClose
CellChanged
DocumentOpened
DocumentSaved
PageAdded
ShapeAdded
TextChanged
WindowSelectionDeleted

For a complete list, choose List of Events from the Contents page of the Visio Automation Reference.

To define a sink object in Visual Basic or VBA:

1. Choose Class Module from the Insert menu and give the new object whatever name you want to use.

   Typically, you would set the object's **Public** property to TRUE, but that isn't required. If you wish, you can code predefined methods such as **Initialize** and **Terminate** or add your own methods to the class.

2. Write an event procedure called **visEventProc** to handle notifications when they are received.

   The **visEventProc** procedure must be declared with the following parameters:

   ```
   Public Sub visEventProc( _
     event As Integer, _
     source As Object, _
     id As Long, _
     seq As Long, _
     subject As Object, _
     etc As Variant )
   ```

3. In the **visEventProc** procedure, write code to handle the notifications received from Visio in whatever way makes sense for your program.

For example, the following **visEventProc** procedure uses a Select Case block to check for three events: **DocumentSaved**, **PageAdded**, and **ShapeDeleted**. Other events fall under the default case (Case Else). Each Case block constructs a string (*strDumpMsg*) that contains the name and event code of the event that fired. Finally, the procedure displays the string in a message box.

Designing event handlers

This **visEventProc** example handles multiple events in a single procedure, and uses Select Case to branch according to the event. However, Visio does not require this. Depending on the number and category of events your program will handle, you may prefer to define a different sink object for each event, or use other techniques to branch within the procedure.

visEventProc in \DVS\VB SOLUTIONS\VB EVENT SAMPLE.CLS

```
' visEventProc - Handles Visio events
'
' Parameters:
   ' eventCode    The event code of the event that fired.
   ' sourceObj    A reference to the source object whose EventList contains the Event object.
   ' eventID      The unique ID of the Event object in its EventList collection.
   ' seqNum       The sequence of this event among events fired in the instance of Visio.
   ' subjectObj   A reference to the object that is subject of the event.
   ' moreInfo     A string that contains additional information, defined when the
   '                Event object was created.
'
Public Sub VisEventProc(eventCode As Integer, sourceObj As Object, eventID As Long, _
                      seqNum As Long, subjectObj As Object, moreInfo As Variant)
Dim strDumpMsg As String

   ' Find out which event fired.
   Select Case eventCode
      Case visEvtCodeDocSave
         strDumpMsg = "Save(" & eventCode & ")"

      Case (visEvtPage + visEvtAdd)
         strDumpMsg = "Page Added(" & eventCode & ")"

      Case visEvtCodeShapeDelete
         strDumpMsg = "Shape Deleted(" & eventCode & ")"

      Case Else
         strDumpMsg = "Other(" & eventCode & ")"
   End Select

   ' Display the event name and code
   frmEventDisplay.EventText.Text = strDumpMsg

End Sub
```

Creating the Event object. When your program runs, it should create Event objects when they are needed.

To create an Event object that sends a notification:

1. Create an instance of your sink object.

   You can use the same instance of the sink object for multiple Event objects, or you can use more than one instance of the sink object if you wish.

2. Get a reference to the EventList collection of the source object in Visio.
3. Use the **AddAdvise** method and provide the event code and a reference to the sink object.

   AddAdvise has two additional arguments. The third argument is reserved for future use and should be null (""). The fourth argument can be a string of arguments for the event handler. Visio assigns these to the Event object's **TargetArgs** property. When your program receives the notification, it can get this property to obtain the arguments.

For example, the following code creates an instance of the sink object *CEventSamp* and creates Event objects to send notifications of the following events: **DocumentSaved**, **PageAdded**, and **ShapeDeleted**.

FormLoad in \DVS\VB SOLUTIONS\VB EVENT SAMPLE.FRM

```
' Create an instance of the sink object class CEventSamp, declared in Event Sample.CLS
Dim g_Sink As CEventSamp
Dim docObj As Visio.Document

Private Sub Form_Load()
  Dim eventsObj As Visio.EventList
  ...
  ' Create an instance of the CEventSamp class
  ' g_Sink is global to the form.
  Set g_Sink = New CEventSamp

  ' Create a new drawing
  ' An instance of Visio has already been assigned to g_appVisio
  Set docObj = g_appVisio.Documents.Add("")

  ' Get the EventList collection of this document.
  Set eventsObj = docObj.EventList

  ' Add Event objects that will send notifications.
  ' Add an Event object for the DocumentSaved event.
  eventsObj.AddAdvise visEvtCodeDocSave, g_Sink, "", "Document Saved..."

  ' Add an Event object for the ShapeDeleted event.
  eventsObj.AddAdvise visEvtCodeShapeDelete, g_Sink, "", "Shape Deleted..."

  ' Add an Event object for the PageAdded event.
  eventsObj.AddAdvise (visEvtPage + visEvtAdd), g_Sink, "", "Page Added..."

End Sub
```

What happens when the Event object fires. When an Event object that sends a notification fires, Visio calls the **visEventProc** procedure of the corresponding sink object, passing the following arguments:

- The event code of the event that caused the Event object to fire.
- A reference to the source object whose EventList contains the Event object that fired.
- The unique identifier of the Event object within its EventList collection. Unlike the **Index** property, the identifier does not change as objects are added and removed from the collection. You can access the Event object from within the **visEventProc** procedure by using *source.EventList.ItemFromID(id)*.
- The sequence of the event relative to events that have fired so far in the instance of Visio.
- A reference to the subject of the event, which is the object to which the event occurred.
- Additional information, if any, that accompanies the notification. For most events, this argument will be **Nothing**.

To continue the earlier example, when an event such as **PageAdded** fires, Visio calls **visEventProc** on the sink object *g_sink*, which displays the event name and its event code.

Releasing Event objects. Event objects created with **AddAdvise** persist until

- The Event object is deleted with the **Delete** method.
- All references to the source object are released, including references that are held indirectly through a reference to the source object's EventList collection or to an Event object in the collection.
- The instance of Visio is closed.

Cleaning up before Visio closes

Visio fires a **BeforeQuit** event before releasing references to sink objects. If your program needs to perform cleanup tasks before Visio is closed, handle the **BeforeQuit** event.

If you implement your sink object in a class module in the VBA project of a document, the project will be closed before **BeforeQuit** is fired. In this case, handle the **BeforeDocClose** event to perform cleanup tasks before Visio is closed.

Getting information about events

You can get information about an existing Event object by getting properties such as the following:

Event. The event code of the event or events that causes the Event object to fire. Event codes are prefixed with **visEvt** in the Visio type library, and are listed in event topics in the online Visio Automation Reference.

Action. The action that is triggered when the Event object fires. In the Visio version 5.0 product line, the value of the **Action** property can be **visActCodeRunAddon** or **visActCodeAdvise**.

Target. For an Event object that runs an add-on, the name of the add-on to run. For an Event object that sends a notification, the **Target** property is not available, and attempting to get or set the property will cause an exception.

TargetArgs. Contains the argument string passed with **Add** or **AddAdvise** when the Event object was created.

In addition, the **EventInfo** property of the Application object provides more information about certain events after they occur. For example, if an Event object fires after shapes are deleted, you can get the names of the deleted shapes from the **EventInfo** property.

Because there's only one **EventInfo** property for potentially many events, you must specify the event you're interested in when you get **EventInfo**. To do this, pass the event's sequence number (which Visio passes as the third argument when it calls **visEventProc** on the corresponding sink object), or pass **visEvtIDMostRecent** for the most recent event. If there's no additional information for the event you specify, **EventInfo** returns **Nothing**.

For details about the information passed by a particular event, see that event in the online Visio Automation Reference.

Controlling Event objects

You can control the behavior of an existing Event object from a program in these ways:

- To prevent an Event object from firing temporarily, set its **Enabled** property to FALSE.
- To fire an Event object explicitly without waiting for the event to occur, use its **Trigger** method and specify a context string to send to the target of the action.
- To suspend event processing in an instance of Visio, set the Application object's **EventsEnabled** property to FALSE. No events will fire in that instance until **EventsEnabled** is set to TRUE.

16

Customizing the Visio user interface

If you're writing a program for others to use, you can customize the Visio user interface (UI) to make running your program easier or to simplify Visio for your users. For example, you can add a toolbar button or menu item to the user interface that runs your program. You can also remove items or create your own custom user interface file that contains only menu and toolbar items specific to your business needs.

This chapter discusses the objects in the Visio UI object model, demonstrates how to add and remove items from a user interface, how to create a custom user interface file, and how to restore the built-in Visio user interface.

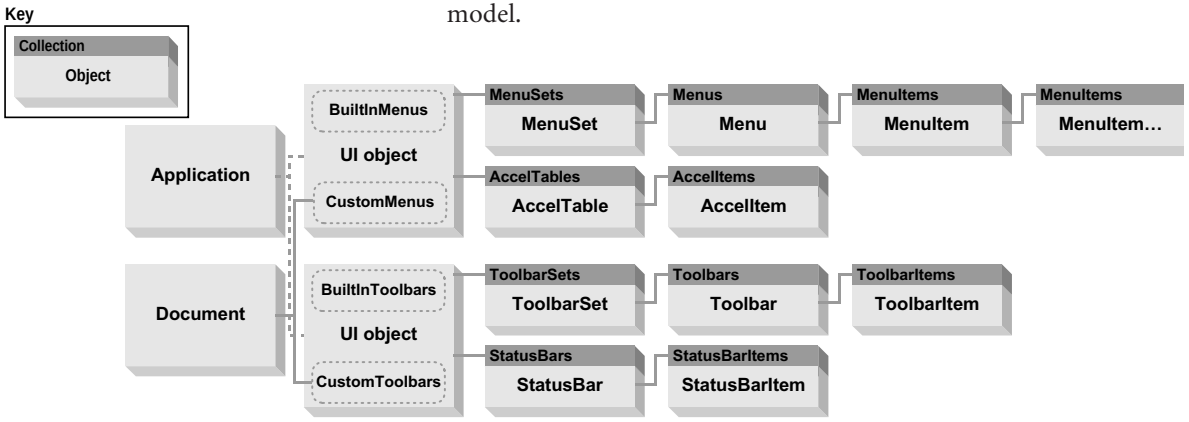
Topics in this chapter

What you can customize

You can help your users work more quickly and easily by letting them launch your programs directly from a custom user interface. You can customize Visio menus and menu items; toolbars and toolbar items; status bars and status bar items; and accelerators from programs written in Visual Basic for Applications (VBA), Visual Basic, C/C++, or other languages that support Automation. For example, you can add your own menu or toolbar items to Visio, or temporarily remove items from the Visio user interface.

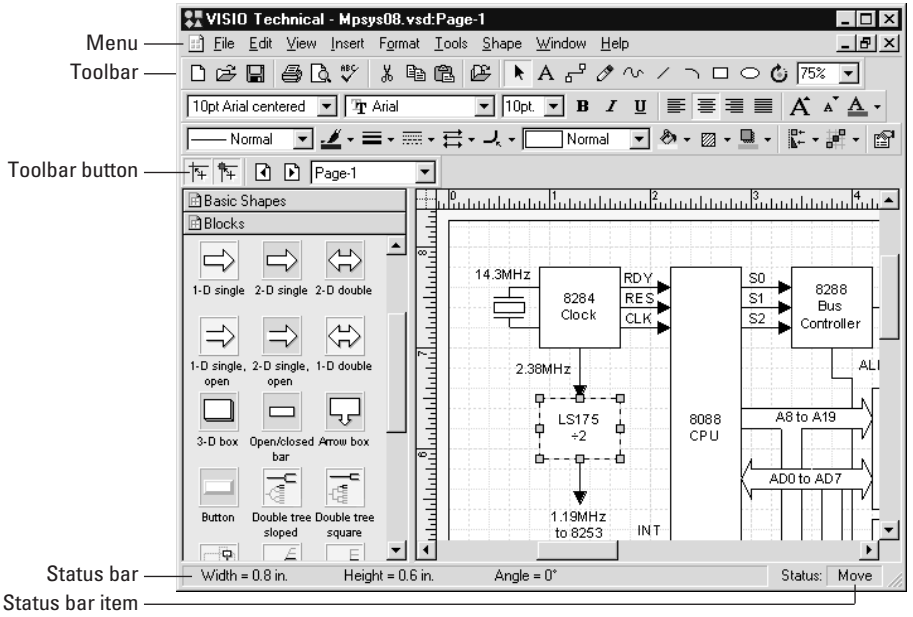
You customize the Visio user interface (UI) by working with UI objects. Just as you get Document objects to work with various open documents in an instance of Visio, you get UI objects to work with the menus, toolbars, status bars, or accelerators of the Visio UI.

The following illustration shows the UI objects in the Visio object model.



UI objects in the Visio object model

Many objects in the Visio UI object model correspond to items you see in Visio. For example, a Menu object can represent the Visio Edit menu, and a MenuItem object can represent the Visio Copy command located on the Edit menu, a custom menu command for a macro or add-on, or an anchor for a hierarchical menu.

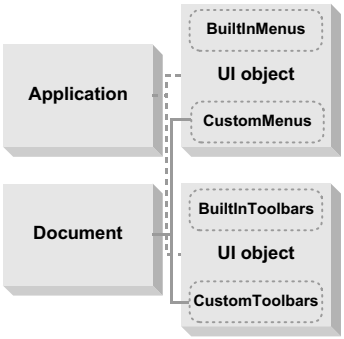


Many Visio UI objects correspond to items you can see in Visio.

Getting a UI object. UI objects differ from other objects in the Visio object model. The properties **BuiltInMenus**, **CustomMenus**, **BuiltInToolbars**, and **CustomToolbars** each return a UI object—there is no UI Object property. You access menus or accelerators by getting the **BuiltInMenus** or **CustomMenus** property, and access toolbars or status bars by getting the **BuiltInToolbars** or **CustomToolbars** property.

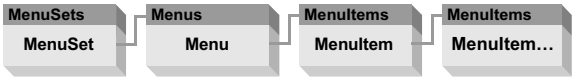
To modify a copy of the built-in Visio user interface, use the **BuiltInMenus** or **BuiltInToolbars** property of the Application object to obtain a UI object. To modify a custom user interface, use the **CustomMenus** or **CustomToolbars** property of an Application or Document object to obtain a UI object. For example, to modify a copy of the built-in Visio menus and obtain a UI object that represents Visio menus and accelerators, start with this code:

```
Dim uiObj as Visio.UIObject
Set uiObj = Visio.Application.BuiltInMenus
```



Four properties each return UI objects

Getting menu objects. Visio displays different sets of menus in different window contexts, such as a drawing window, ShapeSheet window, or stencil window. For example, Visio displays different menu items when the drawing window is active than it does when the stencil window is active.



Menu objects in the Visio UI object model

Here is a list of Menu objects in the Visio UI object model:

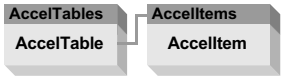
- **MenuSets**. The collection of all possible Visio menu sets. To get a MenuSets collection, get the **MenuSets** property of a UI object.
- **MenuSet**. The set of menus available in a given window context. For example, a MenuSet object could represent the set of menus available when the drawing window is active. To get a MenuSet object, use the **ItemAtID** property of a MenuSets collection and specify the ID of the context you want.
- **Menus**. A collection of Visio menus in a menu set. To get a Menus collection, get the **Menus** property of a MenuSet object.
- **Menu**. A Visio menu. To get a Menu object, use the **Item** property of a Menus collection with the index of the menu you want. Menus are indexed in the order they appear from left to right in Visio. For example, in most window contexts, the File menu has an index of 0. To add a Menu object, use the **Add** or **AddAt** method of a Menus collection.
- **MenuItems**. A collection of menu items in a Visio menu. To get a MenuItems collection, get the **MenuItems** property of a Menu object or MenuItem object if it represents a hierarchical menu.
- **MenuItem**. A menu item, or command, on a Visio menu. To get a MenuItem object, use the **Item** property of the MenuItems collection with the index of the menu item you want. Menu items are indexed in the order they appear from top to bottom on the menu. For example, the Undo command on the Visio Edit menu has an index of 0. To add a MenuItem object, use the **Add** or **AddAt** method of the MenuItems collection.

Shortcut and hierarchical menus

A shortcut menu, sometimes called a right-click menu or context-sensitive menu, is the menu that appears when you right-click something such as a shape, page, or stencil window. All MenuSet objects correspond to a given window context except for a MenuSet object that represents a shortcut menu.

A hierarchical menu, or cascading menu, is a submenu of another menu item. For example, the Visio Macro menu item has a hierarchical menu with menu items such as Macros and Visual Basic Editor. If a Visio menu item has a hierarchical menu, then the MenuItem object that represents the hierarchical menu has a MenuItems collection with MenuItem objects. The **CmdNum** property of the MenuItem object representing a hierarchical menu should be set to **visCmdHierarchical**, and the remaining properties and methods that should be used are: **Caption**, **Index**, **MenuItems**, **Parent**, and **Delete**. All other properties and methods will be ignored.

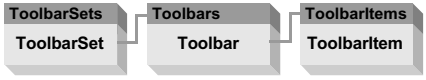
Getting accelerator objects. An accelerator is a combination of keys that, when pressed, execute a command. For example, the accelerator for the Copy menu item is Ctrl+C, and the accelerator for the Paste menu item is Ctrl+V.



Accelerator objects in the Visio UI object model

- **AccelTables**. The collection of all Visio accelerator tables. Visio uses different accelerators in different window contexts. To get an AccelTables collection, get the **AccelTables** property of a UI object.
- **AccelTable**. The table of accelerators available for a given window context. AccelTable objects exist only for window contexts, such as the drawing window, not for shortcut menus. To get an AccelTable object, use the **ItemAtID** property of an AccelTables collection and specify the ID of the context you want.
- **AccelItems**. A collection of accelerators in an accelerator table. To get an AccelItems collection, get the **AccelItems** property of an AccelTable object.
- **AccelItem**. A single accelerator item. Accelerator items such as Ctrl+C (Copy) and Ctrl+V (Paste) are available when a drawing window is active. To get an AccelItem object, use the **Item** property of an AccelItems collection with the index of the menu you want.

Getting toolbar objects. Visio displays different sets of toolbars in different window contexts. For example, when the ShapeSheet window is active, Visio displays different toolbar buttons than it does when the drawing window is active.



Toolbar objects in the Visio UI object model

- **ToolbarSets**. The collection of all possible Visio toolbar sets. To get a ToolbarSets collection, get the **ToolbarSets** property of a UI object.

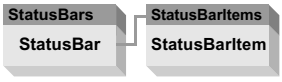
UI object collections

The following collections are indexed starting with 0 rather than 1:

- AccelTables
- AccelItems
- MenuSets
- Menus
- MenuItems
- StatusBars
- StatusBarItems
- ToolbarSets
- Toolbars
- ToolbarItems

- **ToolbarSet**. The set of toolbars available in a given window context. For example, a ToolbarSet object could represent the set of toolbars available when the ShapeSheet window is active. To get a ToolbarSet object, use the **ItemAtID** property of a ToolbarSets collection and specify the ID of the context you want.
- **Toolbars**. A collection of Visio toolbars in a toolbar set. To get a Toolbars collection, get the **Toolbars** property of a ToolbarSet object.
- **Toolbar**. A Visio toolbar. To get a Toolbar object, use the **Item** property of a Toolbars collection with the index of the toolbar you want. Toolbars are indexed in the order they appear from top to bottom in Visio. To add a toolbar, use the **Add** or **AddAt** method of a Toolbars collection.
- **ToolbarItems**. A collection of toolbar items in a Visio toolbar. To get a ToolbarItems collection, get the **ToolbarItems** property of a Toolbar object.
- **ToolbarItem**. A toolbar button on a Visio toolbar. To get a ToolbarItem object, use the **Item** property of the ToolbarItems collection with the index of the toolbar item you want. Toolbar items are indexed in the order they appear from left to right on the toolbar. For most contexts the Blank Drawing toolbar item has an index of 0. To add a toolbar item, use the **Add** or **AddAt** method of a ToolbarItems collection.

Getting status bar objects. A status bar displays status bar items that give users information about the status of a shape, menu, or tool, such as the location of a drawing tool on the page or the angle and length of a line.



Status bar objects in the Visio UI object model

- **StatusBars**. The collection of all possible Visio status bars. To get a StatusBars collection, use the **StatusBars** property of a UI object.
- **StatusBar**. The status bar in a given window context. For example, a StatusBar object could represent the status bar displayed when a shape is selected. To get a StatusBar object, use the **ItemAtID** property of a StatusBars collection and specify the ID of the context you want.

Visio BuiltInToolbars

The **BuiltInToolbars** property in Visio 5.0 has been changed to use the Microsoft Office toolbar set by default. The Lotus toolbar set is no longer supported. The **visToolbarLotusSS** argument is ignored.

The **ShowToolbar** property has been added to control whether Visio shows its toolbar.

Toolbar Caption property

Visio 5.0 now uses the **Caption** property of the Toolbar object. The caption will appear on the hierarchical menu for the Toolbars menu item, available from the View menu in the drawing window.

- **StatusBarItems**. A collection of status bar items on a Visio status bar. To get a StatusBarItems collection, get the **StatusBarItems** property of a StatusBar object.
- **StatusBarItem**. A Visio status bar item on a status bar. To get a StatusBarItem object, use the **Item** property of the StatusBarItems collection with the index of the status bar item you want. Status bar items are indexed in the order they appear from left to right on a status bar. For example, when a 2-D shape is selected the status bar item for its width appears on the left and has an index of 0.

Planning user interface changes

As you begin designing your custom user interface, you need to answer the following questions:

- Will you be customizing a copy of the built-in Visio user interface or an existing custom user interface?
- Should the custom user interface be available on a document or application level? (What is its *scope*?)
- Should the custom user interface be available only when a document is active, throughout a single Visio session, or each time Visio runs? (How long does it *persist*?)

The scope of a user interface defines the context in which your custom user interface is available. Persistence is the length of time in which your user interface is available.

Customizing a copy of the built-in Visio UI versus an existing custom UI

Before you retrieve a user interface, first determine whether it is the built-in Visio user interface or a custom user interface.

When you retrieve the built-in Visio menus or toolbars, you are actually retrieving a *copy*, or *snapshot*, of the built-in Visio user interface that you can manipulate. The original built-in Visio user interface remains untouched so you can restore it later. When you retrieve a custom user interface, you are retrieving the currently active custom user interface, not a copy.

To determine which UI is in use, check the **CustomMenus** and **CustomToolbars** properties of all the Document objects in the Documents collection. Then check the same properties of the Application object. If an object is not using a custom UI, both properties return **Nothing** and you can simply retrieve a copy of the built-in Visio UI.

If a custom UI is in use, you can decide whether you want to replace the custom UI with your own or just add your custom UI items to it.

The following example demonstrates how to retrieve the currently active UI for your document without replacing the application-level custom UI. You then need to write additional code to add your custom UI items.

```
'Check if there are document custom menus
If ThisDocument.Custommenus Is Nothing Then
   'Check if there are Visio custom menus
   If Visio.Application.CustomMenus Is Nothing Then
      'Use the Built-in menus
      Set visUIObj = Visio.Application.BuiltInMenus
   Else
      'Use the Visio custom menus
      Set visUIObj = Visio.Application.CustomMenus
      'Save to a file
      strPath = Visio.Application.Path & _
      "\custUI.vsu"
      visUIObj.SaveToFile (strPath)
      'Set the existing custom UI for the document
      ThisDocument.CustomMenusFile = strPath
      'Grab this document's UI object
      Set visUIObj = ThisDocument.CustomMenus
      'Delete the newly created temp file
      Kill Visio.Application.Path & "\custUI.vsu"
      ThisDocument.ClearCustomMenus
   End If
Else
   'Use the file custom menus
   Set visUIObj = ThisDocument.CustomMenus
End If
```

For further details about how to use the **CustomMenus** and **CustomToolbars** properties, see the online Visio Automation Reference.

Controlling the scope of your UI

Just as you can get a UI object from the Document or Application object, you can also apply your custom user interface changes to the Document or Application object by using its **SetCustomMenus** or **SetCustomToolbars** method. The scope you want determines which object you should apply your changes to.

You can choose to make your custom user interface available on an application or document level. To use a custom user interface for Visio on the application level, that is, regardless of which document is open, apply your custom user interface to the Application object.

To use a custom user interface on a document level, that is, while a document is active, apply your custom user interface to a Document object. This is the way you'll typically work when programming in VBA. This example shows how to set custom menus for the ThisDocument object:

```
Dim uiObj as Visio.UIObject
'Get a copy of the built-in Visio menus
Set uiObj = Visio.Application.BuiltInMenus
... 'Make custom UI changes
'Set custom menus for ThisDocument
ThisDocument.SetCustomMenus uiObj
```

Controlling the persistence of your UI

The approach you use to customize the Visio user interface depends on the extent of the changes you intend to make and the development environment in which you are programming. Depending on the scope of your user interface changes, you may want your changes to persist while a document is active, throughout a single Visio session, or each time Visio runs.

While a document is active. A document can have a custom user interface that takes precedence over the Visio application's user interface (custom or built-in) while the document is active. For example, when a user creates a document from a particular template, you can add a toolbar button that runs a wizard to help the user create a drawing. As soon as the user closes the document, Visio reverts to the built-in user interface, as long as a custom user interface is not set for the application or the next active document.

When you are customizing the Visio user interface from VBA, you usually work on a document level, so set the custom user interface for the ThisDocument object or load a custom user interface (.VSU) file when the **DocumentOpened** event occurs to make a custom user interface persist while a document is active. For details about triggering events, see Chapter 15, "Handling events in Visio."

During a single Visio session. If you want your custom user interface to persist during a single Visio session, set the custom user interface for the Application object from a VBA macro or standalone program.

Each time Visio runs. If you want your user interface changes to replace the Visio user interface on a more permanent basis, you can set the custom user interface for the Application object each time Visio runs or create a custom user interface file. A *custom user interface file* contains your custom user interface data—a snapshot of your custom user interface. To make extensive changes to the Visio user interface, you can code your custom user interface changes in an external development environment and save a custom user interface file.

After you create a custom user interface file, you must tell Visio the file name by setting the **CustomMenusFile** or **CustomToolbarsFile** properties for the Application object. For details, see "Applying a custom user interface" later in this chapter.

Your custom user interface file loads each time Visio runs until you specify a different file or restore the built-in Visio user interface. For details about creating a custom user interface file, see "Applying a custom user interface" later in this chapter.

Making user interface changes

After you decide which UI object you want to work with and the scope and persistence of your custom user interface, you can begin to make the changes themselves. To get to the item that you want to remove or to get the location where you want to add an item, you must navigate the Visio object model. To do this, first get a UI object; then a menu, accelerator, toolbar, or status bar; and then the specific items that you want to change.

NOTE After you make your user interface changes, you must set the customized UI object so your user interface takes effect. For details about setting a UI object, see "Putting custom UI changes into effect" later in this chapter.

Getting a UI object

To access UI objects in an instance of Visio, get one of the following properties:

- **BuiltInMenus**. Get this property for a copy of the built-in Visio menus and accelerators; this is a property of the Application object.
- **BuiltInToolbars**. Get this property for a copy of the built-in Visio toolbars and status bars; this is a property of the Application object. The Visio 5.0 UI uses the Microsoft Office toolbar set.
- **CustomMenus**. Get this property for the custom menus and accelerators currently in effect for that instance or that document; this is a property of the Application or Document object.
- **CustomToolbars**. Get this property for the custom toolbars and status bars currently in effect for that instance or that document; this is a property of the Application or Document object.

For example, to get a UI object that represents a copy of the built-in Visio menus:

```
Dim uiObj As Visio.UIObject
Set uiObj = Visio.Application.BuiltInMenus
```

To get a UI object that represents a copy of the built-in Visio toolbars:

```
Dim uiObj as Visio.UIObject
Set uiObj = Visio.Application.BuiltInToolbars(0)
```

The Visio UI uses the Microsoft Office toolbar set by default.

To get a UI object that represents the custom menus for the ThisDocument object, assuming custom menus are set for the ThisDocument object:

```
Dim uiObj As Visio.UIObject
Set uiObj = ThisDocument.CustomMenus
```

Getting a MenuSet, ToolbarSet, AccelTable, or StatusBar object

To get a MenuSet, ToolbarSet, AccelTable, or StatusBar object, use the **ItemAtID** method of the appropriate collection and specify the ID of the object you want. The object's ID identifies its context.

This example gets a MenuSet object that represents the drawing window menus:

```
Dim uiObj As Visio.UIObject
Dim menuSetObj As Visio.MenuSet

'Get a UI object that represents a copy of the
'built-in Visio menus
Set uiObj = Visio.Application.BuiltInMenus

'Get the drawing window menu set
Set menuSetObj = _
uiObj.MenuSets.ItemAtId(visUIObjSetDrawing)
```

This example gets a ToolbarSet object that represents the ShapeSheet window toolbars:

```
Dim uiObj As Visio.UIObject
Dim toolbarSetObj As Visio.ToolbarSet

'Get a UI object that represents a copy of the
'built-in Visio toolbars
Set uiObj = Visio.Application.BuiltInToolbars(0)

'Get the ShapeSheet window toolbar set
Set toolbarSetObj = _
uiObj.ToolbarSets.ItemAtID(visUIObjSetShapeSheet)
```

Constants for the window contexts you can specify with **ItemAtID** are listed in the following table and defined in the Visio type library. The first seven window contexts are the most commonly used. Not all window contexts are currently in use by Visio, and not all exist in each set; the table lists the window contexts that are available for specific object sets.

Visio contexts for MenuSet, ToolbarSet, AccelTable, and StatusBar objects

ID Constant	Context	Availability
visUIObjSetNoDocument	Visio window, no documents open	● ▲ ◆ ■
visUIObjSetDrawing	Drawing window	● ▲ ◆ ■
visUIObjSetStencil	Stencil window	● ▲ ◆ ■
visUIObjSetShapeSheet	ShapeSheet window	● ▲ ◆ ■
visUIObjSetIcon	Icon editing window	● ▲ ◆ ■
visUIObjSetInPlace	In-place editing window	● ■
visUIObjSetPrintPreview	Print Preview window	● ▲ ◆ ■
visUIObjSetText	Text editing window	☒
visUIObjSetCntx_DrawObjSel	Shortcut menu, selected shape	●
visUIObjSetCntx_DrawOleObjSel	Shortcut menu, selected linked or embedded object	●
visUIObjSetCntx_DrawNoObjSel	Shortcut menu, drawing page (nothing selected)	☒
visUIObjSetCntx_InPlaceNoObj	Shortcut menu, in-place editing window (nothing selected)	☒
visUIObjSetCntx_TextEdit	Shortcut menu, text editing window	●
visUIObjSetCntx_StencilRO	Shortcut menu, read-only stencil	●
visUIObjSetCntx_ShapeSheet	Shortcut menu, ShapeSheet window	●
visUIObjSetCntx_Toolbar	Shortcut menu, toolbar	●
visUIObjSetCntx_FullScreen	Shortcut menu, full screen view	●
visUIObjSetBinderInPlace	Shortcut menu, Microsoft Office Binder	● ■
visUIObjSetCntx_StencilRW	Shortcut menu, stencil opened as an original	●
visUIObjSetCntx_StencilDocked	Shortcut menu, docked stencil	●
visUIObjSetHostingInPlace	In-place editing window	● ■
visUIObjSetCntx_Hyperlink	Shortcut menu, hyperlink	●
visUIObjSetPal_LineColors	Drawing window	▲

(Table continued on next page)

Visio contexts for MenuSet, ToolbarSet, AccelTable, and StatusBar objects (continued)

ID Constant	Context	Availability
visUIObjSetPal_LineWeights	Drawing window	▲
visUIObjSetPal_LinePatterns	Drawing window	▲
visUIObjSetPal_FillColors	Drawing window	▲
visUIObjSetPal_FillPatterns	Drawing window	▲
visUIObjSetPal_TextColors	Drawing window	▲
visUIObjSetPal_AlignShapes	Drawing window	▲
visUIObjSetPal_DistributeShapes	Drawing window	▲
visUIObjSetPal_Shadow	Drawing window	▲
visUIObjSetPal_LineEnds	Drawing window	▲
visUIObjSetPal_CornerRounding	Drawing window	▲

Key:
- ● MenuSets object
- ▲ ToolbarSets object
- ◆ StatusBars object
- ■ AccelTables object
- ☒ Reserved for future use

Adding a menu and menu item

After getting a UI object, you can add or remove items from the user interface. To add items, navigate the UI objects to get the collection that contains the kind of item you want to add and use that collection's **Add** or **AddAt** method.

The following example adds a new menu and menu item available when the Visio drawing window is active.

```
Dim uiObj As Visio.UIObject
Dim menuSetsObj As Visio.MenuSets
Dim menuSetObj As Visio.MenuSet
Dim menusObj as Visio.Menus
Dim menuObj As Visio.Menu
Dim menuItemsObj as Visio.MenuItems
Dim menuItemObj As Visio.MenuItem

'Get a UI object that represents a copy of the
'built-in Visio menus
Set uiObj = Visio.Application.BuiltInMenus
```

Adding a shortcut menu item

For an example of a program that adds a menu item to a shortcut menu, see the **AddShortcutMenuItem** macro in the DVS module in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST on your Visio 5.0 CD.

```
'Get the MenuSets collection
Set menuSetsObj = uiObj.MenuSets

'Get drawing window MenuSet object; Get the context
Set menuSetObj= _
menuSetsObj.ItemAtId(visUIObjSetDrawing)

'Get the Menus collection
Set menusObj = menuSetObj.Menus

'Add a Demo menu before the Window menu
'A menu without a menu item will not appear.
Set menuObj = menusObj.AddAt(7)
menuObj.Caption = "Demo"
```

The first half of this example assumes the Window menu is still in its initial position—seventh from the left on the menu bar. Adding or removing menus can change the position of other menus, however.

The second half of the example, shown below, adds a menu item to the Demo menu and sets the menu item's properties. For details, see "Setting properties of an item" later in this chapter.

The following sample code uses the **Add** method to add one item to the Demo menu that was added in the preceding sample code. When you add an item using the **Add** method, the item is added to the end of a collection. This example adds only one menu item, so its position is not an issue. However, if you were to add another item using the **Add** method, it would appear at the bottom of the menu. To control where a menu item appears, use the **AddAt** method and specify the ordinal position of the item.

```
'Get the MenuItems collection
Set menuItemsObj = menuObj.MenuItems

'Add a MenuItem object to the new Demo menu
Set menuItemObj = menuItemsObj.Add

'Set the properties for the new menu item
menuItemObj.Caption = "Run &Demo Program"
menuItemObj.AddOnName = "Demo.EXE"
menuItemObj.AddOnArgs = "/DVS=Fun"
menuItemObj.MiniHelp = "Run the Demo program"

'Tell Visio to use the new UI object (custom menus)
'while the document is active
ThisDocument.SetCustomMenus uiObj
```

The last statement, *ThisDocument.SetCustomMenus uiObj,* tells Visio to use the custom menus while the document is active. The custom UI changes don't persist after the user closes the document.

Adding a toolbar button

This example demonstrates how to add a toolbar button to a copy of the built-in Microsoft Office toolbar for the drawing window context.

To run this VBA program, open the VBA Samples Template (VBA SAMPLES.VST), choose Macro from the Tools menu, then DVS, and then **AddToolbarButton**. The code for this example is in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST.

AddToolbarButton macro in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub AddToolbarButton ()

   'Object variables to be used in the program.
   Dim uiObj As Visio.UIObject
   Dim toolbarSetObj As Visio.ToolbarSet
   Dim toolbarItemsObj As Visio.ToolbarItems
   Dim objNewToolbarItem As Visio.ToolbarItem

   'Get the UI object for the Microsoft Office toolbars
   Set uiObj = Visio.Application.BuiltInToolbars(0)

   'Get the Drawing Window ToolbarSet object
   Set toolbarSetObj = uiObj.ToolbarSets.ItemAtID(visUIObjSetDrawing)

   'Get the ToolbarItems collection
   Set toolbarItemsObj = toolbarSetObj.Toolbars(0).ToolbarItems

   'Add a new button in the first position
   Set objNewToolbarItem = toolbarItemsObj.AddAt(0)

   'Set the properties for the new toolbar button
   objNewToolbarItem.ActionText = "Run Stencil Report Wizard"
   objNewToolbarItem.AddOnName = "Stndoc.exe"
   objNewToolbarItem.CntrlType = visCtrlTypeBUTTON
   objNewToolbarItem.Priority = 1

   'Set the icon for the new toolbar button
   objNewToolbarItem.IconFileName "dvs.ico"

   'Tell Visio to use the new UI object (custom toolbars) while the document is active
   ThisDocument.SetCustomToolbars uiObj

End Sub
```

Here are some notes on the code:

Set toolbarItemsObj = toolbarSetObj.Toolbars(0).ToolbarItems. Toolbars in Visio are ordered vertically. When specifying a location for a Toolbar object, 0 represents the topmost toolbar. A Toolbars collection can include a maximum of four Toolbar objects.

Set objNewToolbarItem = toolbarItemsObj.AddAt(0). Toolbar items in Visio are ordered horizontally, so this statement adds the ToolbarItem or button at the leftmost location on the toolbar.

objNewToolbarItem.CntrlType = visCtrlTypeBUTTON sets the type of toolbar button to display. The **CntrlType** property is also used for status bar items. Visio includes other constants for the default Visio toolbar buttons, but the only constant you can use for your custom toolbar buttons is **visCtrlTypeBUTTON**.

objNewToolbarItem.Priority = 1 gives this toolbar item the highest priority. Not all toolbar items appear at all resolutions—fewer items appear on low-resolution monitors such as VGA. If a user has a low-resolution monitor, low-priority toolbar items do not appear, but high-priority items do. The higher the monitor resolution, the more toolbar items appear.

objNewToolbarItem.IconFileName "dvs.ico" gets the DVS.ICO file that contains the bitmap for the toolbar to display from a folder along the Visio add-ons path specified on the File Paths tab. The icon file should contain a 32-by-32-pixel icon and a 16-by-16-pixel icon. Visio displays the 16-by-16-pixel icon in "small icon" mode and the 32-by-32-pixel icon in "large icon" mode.

ThisDocument.SetCustomToolbars uiObj uses the custom toolbars while the document is active.

Toolbar button priority

For an example of a program that changes the priority of buttons on a toolbar, see the **ChangeToolbarButtonPriority** macro in the DVS module in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST on your Visio 5.0 CD.

Setting properties of an item

After you've added an item, you can set properties that define it. For example, you can set the **Caption** property of a menu item to define the text that appears on the menu or set the **IconFileName** method of a toolbar item to specify and get the icon to display. You can also change properties of an existing item.

The most significant property of a menu item or toolbar item is **AddOnName**, which specifies the program or macro to run when the user chooses the menu item or clicks the button. If the program takes command line arguments, they can be specified with the **AddOnArgs** property. For details about the properties and methods of a particular item, search the online Visio Automation Reference for that item.

Caption specifies the text that appears on a menu or menu item. If you want to display the accelerator with the menu item, include it as part of the **Caption** property's text and insert two spaces between the "\a" and the accelerator text. For example:

```
"Open...\a Ctrl+O"
```

In this example, *Open...* is the menu item's caption; *Ctrl+O* is the accelerator text; and *\a* left justifies the accelerator text. Adding the accelerator text to the **Caption** property doesn't add an accelerator, it simply displays it as part of the caption. You add accelerators by using the accelerator objects in the Visio UI object model.

You can also specify other properties, such as those in the following example:

```
menuItemObj.ActionText = "Run Demo 1"
menuItemObj.MiniHelp = "Run the Demo 1 application"
accelItemObj.Key = 8  'Backspace key
accelItemObj.Alt = True
```

ActionText specifies the text that appears on the Edit menu with Undo, Redo, and Repeat for a menu item. It also appears in any error messages or toolbar tooltips that might be displayed. **MiniHelp** specifies the prompt that appears in the status bar when the user points to the menu item.

IconFileName method

The **IconFileName** method extracts the bitmap and stores it with your interface. The icon file is no longer needed.

To see an example of a program that changes the icon on a toolbar button, run the **ChangeToolbarButtonIcon** macro in the VBA Samples template located in the \DVS\VBA SOLUTIONS folder on your Visio 5.0 CD.

Identifying a menu or toolbar item

Every built-in Visio menu item and toolbar item represents a Visio command and has a command ID. If you want to remove one of these Visio items, you can identify it by its command ID.

The **CmdNum** property of a custom menu item or toolbar item that runs a program or macro does not correspond to any Visio command ID, so when you want to delete the item, you cannot identify the item using its command ID. Instead, use the **Caption** property string of a custom menu or toolbar item to locate the item.

Key specifies the ASCII key code value for an accelerator. For example, the ASCII key code for the Backspace key is 8, and the ASCII key code for the Esc key is 27. For details about ASCII key codes, see the online documentation for the Microsoft Platform Software Development Kit (SDK). The **Alt**, **Control**, and **Shift** properties modify the **Key** for an accelerator. To set the properties for an accelerator, set any combination of modifiers to TRUE, specify one key code, and set the item's **CmdNum** property. To activate an accelerator command, press the combination of modifiers and the key that corresponds to the key code.

The **CmdNum** property specifies the command ID for an item. Every built-in Visio menu item and toolbar item represents a Visio command and has a command ID. For example, the command ID for Show ShapeSheet is **visCmdWindowShowShapeSheet**; the command ID for Show Master Objects is **visCmdWindowShowMasterObjects**. For a list of valid command IDs, see the Visio type library in the Object Browser and search for "visCmd."

Removing an item from a user interface

You can remove any item from the Visio user interface, whether the item is part of the built-in Visio user interface or a custom item you added. You can't disable, or dim, a menu item or toolbar item.

Removing an item doesn't remove the functionality of that item from Visio, just the access to that functionality. Other avenues, such as accelerators, may still be available. For example, if you remove the Copy command from the Edit menu, but not the accelerator (Ctrl+C), a user can still use the copy functionality by pressing Ctrl+C. You can remove the Show ShapeSheet command from the Window menu, but if the double-click behavior for a shape is to display the ShapeSheet window, that window will still appear when that shape is double-clicked.

Deleting hierarchical menus

For an example that deletes the Visual Basic Editor hierarchical menu item and the corresponding accelerator, see the **DeleteHierarchicalMenuItem** macro in the DVS module in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST on your Visio 5.0 CD.

Hiding the Visio user interface

If you want to completely hide the Visio user interface, use the **ShowToolbar**, **ShowStatusBar**, and **ShowMenus** properties of the Application object.

Use **ShowToolbar** to hide all toolbars. For example:

```
Visio.Application. _
ShowToolbar = False
```

Use **ShowMenus** to hide all menus. For example:

```
Visio.Application. _
ShowMenus = False
```

Use **ShowStatusBar** to hide all status bars. For example:

```
Visio.Application. _
ShowStatusBar = False
```

To remove an item, use the Delete method of that item. For example, the following statements remove the Show ShapeSheet menu item from the Window menu in the drawing window for the running instance of Visio:

```
Dim uiObj as Visio.UIObject
Dim menuSetObj as Visio.MenuSet
Dim menuItemsObj as Visio.MenuItems
Dim i as Integer

Set uiObj = Visio.Application.BuiltInMenus
Set menuSetObj = _
uiObj.MenuSets.ItemAtID(visUIObjSetDrawing)

'Get the Window menu.
Set menuItemsObj = menuSetObj.Menus(7).MenuItems

'Get the Show ShapeSheet menu item by its CmdNum
'property.
'This technique works with localized versions
'of Visio.
For i = 0 to menuItemsObj.Count -1

   If menuItemsObj(i).CmdNum= _
   visCmdWindowShowShapeSheet Then
      menuItemsObj(i).Delete
      Exit For
   End If

Next i

'Replace built-in Visio menus with customized set.
Visio.Application.SetCustomMenus uiObj
```

Removing a toolbar item

This example demonstrates how to delete the Spelling toolbar button from the built-in Microsoft Office version of the Visio toolbar for the drawing window context.

To run this VBA program, open the VBA Samples Template (VBA SAMPLES.VST), choose Macro from the Tools menu, then DVS, and then **DeleteToolbarButton**. The code for this example is in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST.

DeleteToolbarButton macro in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub DeleteToolbarButton ()

  Dim uiObj As Visio.UIObject
  Dim toolbarSetObj As Visio.ToolbarSet
  Dim toolbarItemsObj As Visio.ToolbarItems
  Dim toolbarItemObj As Visio.ToolbarItem
  Dim i As Integer   'Loop variable

  'Get the UI object for the Microsoft Office toolbars
  Set uiObj = Visio.Application.BuiltInToolbars(0)

  'Get the drawing window ToolbarSet object
  Set toolbarSetObj = uiObj.ToolbarSets.ItemAtID(visUIObjSetDrawing)

  'Get the ToolbarItems collection
  Set toolbarItemsObj = toolbarSetObj.Toolbars(0).ToolbarItems

  'Get the Spelling ToolbarItem object
  'Because this code gets the built-in Visio toolbars, you know you'll find the Spelling
  'toolbar item. If code got a custom toolbar, it might not include the Spelling toolbar
  'item.
  For i = 0 To toolbarItemsObj.Count - 1

    'Get the current ToolbarItem object from the collection
    Set toolbarItemObj = toolbarItemsObj(i)

    'Check whether the current toolbar item is the Spelling button by using its constant
    If toolbarItemObj.CmdNum = visCmdToolsSpelling Then
      Exit For
    End If
  Next i

  'Delete the Spelling button
  toolbarItemObj.Delete

  'Tell Visio to use the new UI object (custom toolbars) while the document is active
  ThisDocument.SetCustomToolbars uiObj

End Sub
```

Removing an accelerator

This example demonstrates how to delete the Visual Basic Editor accelerator for the drawing window context.

To run this VBA program, open the VBA Samples Template (VBA SAMPLES.VST), choose Macro from the Tools menu, then DVS, and then **DeleteAccelItem**. The code for this example is in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST.

DeleteAccelItem macro in \DVS\VBA SOLUTIONS\VBA SAMPLES.VST\DVS MODULE

```
Sub DeleteAccelItem()

  Dim uiObj As Visio.UIObject
  Dim accelTableObj As Visio.AccelTable
  Dim accelItemsObj As Visio.AccelItems
  Dim accelItemObj As Visio.AccelItem
  Dim i As Integer

  'Retrieve the UIObject for the copy of the BuiltInMenus
  Set uiObj = Visio.Application.BuiltInMenus

  'Set accelTableObj to the Drawing menu set
  Set accelTableObj = uiObj.AccelTables.ItemAtID(visUIObjSetDrawing)

  'Retrieve the accelerator items collection
  Set accelItemsObj = accelTableObj.AccelItems

  'Retrieve the accelerator item for the Visual Basic Editor by iterating
  'through the accelerator items collection and locating the item you want to delete.
  For i = 0 To accelItemsObj.Count - 1
    Set accelItemObj = accelItemsObj.Item(i)
    If accelItemObj.CmdNum = Visio.visCmdToolsRunVBE Then
      Exit For
    End If
  Next i

  'Delete the accelerator
  accelItemObj.Delete

  'Tell Visio to use the new UI
  ThisDocument.SetCustomMenus uiObj

End Sub
```

Applying a custom user interface

No matter how much code you write to customize the Visio user interface, you must finish your code by setting the custom user interface for an object so that your custom user interface changes will take effect. After you code the custom user interface changes, set the appropriate object or save your custom user interface to a file (.VSU).

Putting custom UI changes into effect

To put custom user interface changes into effect, use the **SetCustomMenus** or **SetCustomToolbars** methods of the Document or Application object. For example:

```
ThisDocument.SetCustomMenus uiObj
```

To set custom toolbars for a single Visio session, use this statement:

```
Visio.Application.SetCustomToolbars uiObj
```

If you change a UI object that represents the active custom toolbars or custom menus while Visio is running, use the **UpdateUI** method of the UI object to display your changes. For example:

```
'Get the UI object for the custom menus
Set uiObj = Visio.Application.CustomMenus
...'Code changes to the custom interface
'Update custom interface with changes
uiObj.UpdateUI
```

Creating, saving, and loading a custom user interface file

If you've made extensive user interface changes and intend to save your custom user interface to a (.VSU) file, use the **SaveToFile** method of the UI object. For example:

```
uiObj.SaveToFile("c:\visio\solutions\office\mytools.vsu")
```

You can load a custom user interface file when you run Visio by setting the custom user interface for the Application object. You can also load a custom user interface file (.VSU) when an event occurs, such as opening a document.

Modifying a custom user interface file

If you are programming in an external development environment, such as Visual Basic, you can load a custom user interface (.VSU) file, make changes to it, and then save the changes to the custom user interface file.

To load a custom user interface file, use the **LoadFromFile** method. For example:

```
uiObj.LoadFromFile _
"shortcut.vsu"
...'Make user interface
'changes
uiObj.SaveToFile "c:\visio\ _
solutions\office\mytools.vsu"
```

To load a custom user interface file for an Application object, set the following properties of the object to the name of the custom user interface file:

- **CustomMenusFile.** Set this property for custom menus and accelerators.
- **CustomToolbarsFile.** Set this property for custom toolbars and status bars.

For example, to load a custom user interface file each time Visio runs, use this statement:

```
Visio.Application.CustomToolbarsFile= _
"c:\visio\solutions\office\mytools.vsu"
```

You need to set these properties for the Application object only once. These properties set the value of the **CustomMenusFile** and **CustomToolbarsFile** entries in the VISIO.INI file and tell Visio the location of the corresponding custom interface file. If a path name is not specified, Visio looks in the folders along the Visio add-ons path, specified in the File Paths tab. If Visio can't find the specified file or if the VISIO.INI file is deleted or modified, Visio reverts to the built-in Visio user interface.

To load a custom user interface file when an event occurs, such as opening a document, put the code in the appropriate event for the Document object.

To load a custom user interface file each time a document is opened, use this statement, in the **DocumentOpened** event for the Document object:

```
ThisDocument.CustomMenusFile= _
"c:\visio\solutions\office\mytools.vsu"
```

Restoring the built-in Visio user interface

If your program customizes the Visio user interface, it's a good idea to restore the built-in Visio user interface when your program finishes executing.

To restore the built-in Visio menus and accelerators, use the **ClearCustomMenus** method of the Document (or Application) object. To restore the built-in Visio toolbars and status bars, use the **ClearCustomToolbars** method of the Document (or Application) object. To clear the custom menus for a Document object, use this statement:

```
ThisDocument.ClearCustomMenus
```

To clear custom toolbars for the Application object, use this statement:

```
Visio.Application.ClearCustomToolbars
```

The next time the document is opened or Visio is run, it uses the built-in Visio user interface.

17

Running and distributing a solution

If you're writing a program for others to use, you need to decide which templates, stencils, and drawings to distribute with your program, within which Visio file you should store your Visual Basic for Applications (VBA) program, and where to install the files. You also need to decide how a user will run your program and what arguments might be passed to your program when it is run. You may also want to create a Setup program that installs an external standalone program, its related stencils and templates, and online help in the appropriate folders.

This chapter discusses where to install various files to take advantage of the Visio default paths, some of the different ways a user can run your program, and the things to consider when distributing your program. For details about creating online help files and Setup programs, see the documentation for your development environment. For details about associating online help with particular shapes, see Chapter 9, "Packaging stencils and templates."

Topics in this chapter

Installing a Visio solution

If you're providing your solution as a VBA program or single executable (.EXE) file, you won't need to create a Setup program to install it. However, if your solution includes an .EXE file, stencils, templates, or an online help file, a Setup program can help the user install your solution easily and accurately.

This section describes where to install your solution's files. For details about creating a Setup program, see the documentation for your development environment.

Where to install your files

Visio looks for files in the folders along a specified Visio path. For example, macros and programs in folders along the Visio add-ons path specified on the File Paths tab (choose Options from the Tools menu, then click File Paths) appear in the Macros dialog box; template files in folders along the Visio templates path appear in the Choose A Drawing Template and Open dialog boxes; stencils in folders along the Visio stencils path are listed on the Stencils menu and in the Open Stencil dialog box. You can take advantage of this behavior by installing your solution files in folders along the appropriate Visio paths.

Visio file paths and folders

When installing your solution, install your program and Visio files in folders along the appropriate path as specified on the File Paths tab (choose Options from the Tools menu, then click File Paths). For example, install templates that contain VBA macros in the \SOLUTIONS folder or any of its subfolders. By default, the \SOLUTIONS folder or any of its subfolders is the specified path for templates, stencils, help files, and add-ons.

You can also change and add folders to the file paths to include custom folders you create. To indicate more than one folder, separate individual items in the path string with semicolons. For example, you can change the add-ons path to "SOLUTIONS;LAYOUT;DATABASE."

NOTE If a path is not fully qualified, Visio looks for the folder in the folder that contains the Visio program files. For example, if the Visio executable file is installed in C:\VISIO, and the add-ons path is "ADD-ONS;D:\ADD-ONS," Visio looks for add-ons in both C:\VISIO\ADD-ONS and D:\ADD-ONS and their corresponding subfolders.

The following table lists the paths along which you should install your program's files.

Install this file	Along this Visio path
Program (.EXE)	add-ons path
Visio library (.VSL)	add-ons path
Online help (.HLP) for programs or shapes	help path
Interface file (.VSU) for Visio or a document	add-ons path
Stencil (.VSS)	stencil path
Template (.VST)	template path

You can also find out what paths are in effect on the user's system by checking the following properties of an Application object:

- **AddonPaths**
- **DrawingPaths**
- **FilterPaths**
- **HelpPaths**
- **StartupPaths**
- **StencilPaths**
- **TemplatePaths**

For example, to get the **AddonPaths** property:

```
Dim path as string

'Set path to the AddonPaths string
path = Visio.Application.AddonPaths
```

Controlling when a program runs

You can run a program in a number of ways depending on what type of program you write and where you install external program files.

Your program can run automatically in the following ways:

- When Visio is started
- When a document is opened
- When the user chooses a command from the shortcut menu of a shape
- When the user chooses a command from a menu or toolbar
- When a ShapeSheet formula is evaluated—typically run by a formula in an Action or Event cell

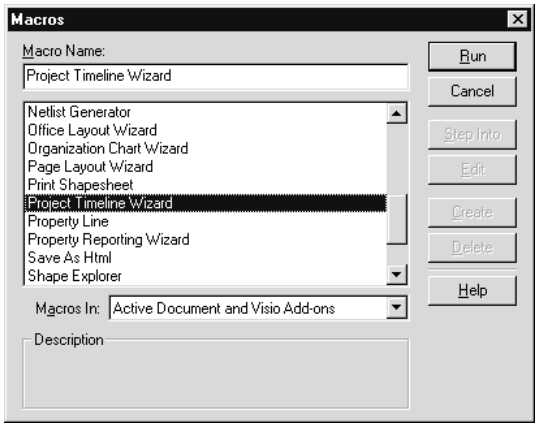
Running a program when Visio is started. To run your program every time Visio itself is started, install your program's .EXE or .VSL file in the Visio startup folder specified on the File Paths tab.

Running a VBA program when an action is taken on a document or when an event occurs. To run a VBA program when a document is opened, enter your code in the ThisDocument class module under the appropriate event procedure. Here are a few common document events used to run programs (grouped by type):

- **BeforeDocumentClose**
  BeforeMasterDelete
  BeforePageDelete
  BeforeSelDelete (before a set of shapes is deleted)
  BeforeStyleDelete
- **DocumentChanged**
  DocumentCreated
  DocumentOpened
  DocumentSaved
  DocumentSavedAs

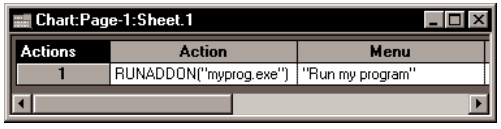
For more information about the ThisDocument object, see Chapter 11, "Using Visio objects." For more information about responding to object events, see Chapter 15, "Handling events in Visio."

Running any program from the Macro submenu or dialog box. To run your program—.EXE or .VSL file, or VBA macro—from the Macros dialog box, install an external program's .EXE or .VSL file along the add-ons path specified on the File Paths tab. All programs along this path appear in the Macros dialog box and on the Macro submenu along with any public VBA macros stored in a Visio file.



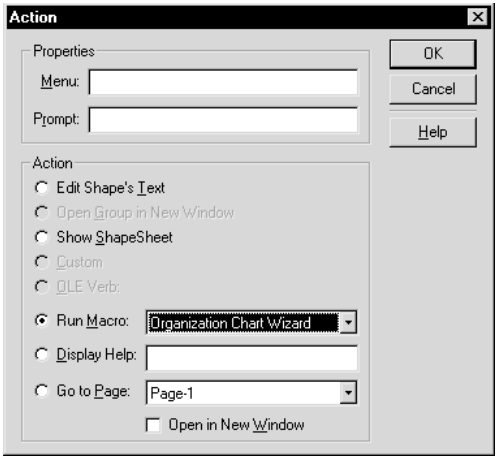
Macros dialog box

Binding a program to an Action or Events cell. You can run a program when a shape event occurs, such as a double-click, or a shape is right-clicked and a menu item is selected. To run any program from a shape's shortcut menu (the menu that appears when a shape or page is right-clicked), enter a formula that uses the Visio RUNADDON function in the Action cell of a row in the shape's Actions section, and enter the text of the menu item in the Menu cell. For example:



Unless you specify a full pathname, Visio looks for your program along the add-ons path specified on the File Paths tab.

You can also set an Action for a particular shape in the Action dialog box (available when the ShapeSheet window is active) by clicking in an Action cell and choosing Action from the Edit menu.

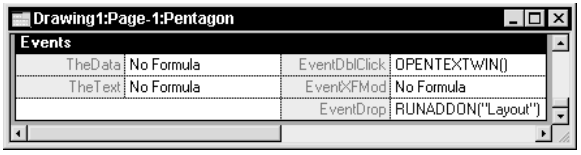


Action dialog box

To pass command-line arguments to your program, use the Visio RUNADDONWARGS function—not used with VBA programs, only add-ons. For example, to run an external program named MYPROG.EXE:

```
= RUNADDONWARGS("myprog.exe", "arguments")
```

To run your program when a particular shape event occurs, put the formula in the Events cell for the event you want to trigger your program.



Events section in the ShapeSheet window

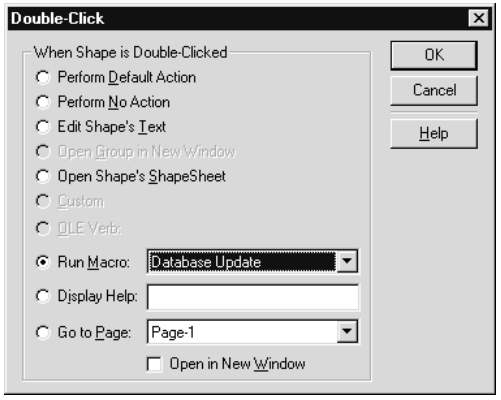
For example, to run a VBA macro when the user drops a particular master in a drawing, put a formula such as the following in the EventDrop cell in the Events section of the master:

```
= RUNADDON("Layout")
```

RUNADDON runs any program

Even though the ShapeSheet entry is RUNADDON, this function runs add-ons or macros—any program you specify in quotation marks. For more information about RUNADDON, search online help for "RUNADDON."

To run a VBA macro when a user double-clicks a particular shape, put the same formula in the EventDblClick cell in the Events section of the shape. Or set the double-click event for a particular shape in the Double-Click dialog box by choosing Double-Click from the Format menu.



Double-Click dialog box

TIP You can exercise finer control over the events that run your programs by using the Visio DEPENDSON function in the formula of a Scratch or User cell. For example, to run an external program when the begin point of a 1-D shape is moved:

```
= RUNADDON("myprog.exe") + DEPENDSON(BeginX)
```

For details about these functions and the Events and Action cells, see online help. For details about actions and events, see Chapter 4, "Enhancing shape behavior."

Binding your program to a menu or toolbar. You can add your own menu command or toolbar item to the Visio user interface and use it to run your program. For details, see Chapter 16, "Customizing the Visio user interface."

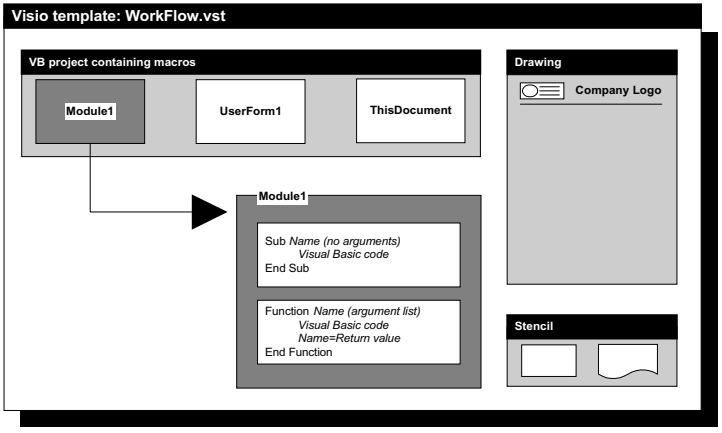
Distributing a program

The files you distribute to your users depend on the type of solution you create. Typically, if you create a VBA program that is stored within a template, you'll distribute only the template and its stencils (and the files the VBA program references, if any). If you create an external program (.EXE), you may need to distribute the .EXE file, a template, and stencils. If you create an add-on, you may need to distribute only the .EXE file or Visio library (.VSL) file. Lastly, if you create only a custom user interface (.VSU), you may need to distribute only that file. You'll also need to be aware of copyright issues (discussed on the next page) if you distribute Visio shapes.

Distributing VBA programs

VBA programs are stored in a Visio template, stencil, or drawing. The only file you typically need to distribute is a template (.VST) or drawing (.VSD) and its stencils (.VSS). If your VBA project references other Visio files, you need to distribute those also. There is no separate program file for a VBA program.

This illustration displays the possible items of a VBA solution.



VBA solution and its elements: template, stencil, and VBA macros

When a user creates a new document from a Visio file, Visio copies the VBA program to the new document and includes references to the same open stencils and other Visio files (if any).

Drawing file size in a VBA solution

Although it's convenient to distribute, a template that contains a lot of VBA code can cause drawings to be much larger than necessary, because the template's code is copied to each drawing created from the template. Such a template can also make a solution more difficult to maintain or upgrade, because each drawing has its own copy of the code.

If the purpose of the code is to help the user create a drawing, and it won't run again after that task is done, the template is probably still the best place for it. However, as an alternative, you can place the bulk of the code in a Visio stencil (.VSS) and call it from the template. This helps conserve drawing file size and improves maintainability, because you can simply distribute a new version of the stencil to upgrade your solution.

If you refer to code in a stencil from another VBA project, Visio automatically opens that stencil and will not permit the user to close it. Such a stencil is opened floating rather than docked, however, which may be less intuitive for users. If your solution also provides masters, consider placing the masters in one stencil and the code in another. Users can then minimize the floating stencil and work with the other stencil in the usual way.

You can minimize a floating stencil from a program by using embedded Windows API calls. For details about such calls, see the Microsoft Platform SDK.

Important licensing information

The stencils, masters, templates, and source code provided with Visio products are copyrighted material, owned by Visio Corporation and protected by United States copyright laws and international treaty provisions.

What this means to you as a solutions developer is this: You cannot distribute any copyrighted master provided with any Visio product, for any purpose other than viewing or modifying a drawing that contains the master, unless your user already has a licensed copy of a Visio product that includes that master. This includes shapes you create by modifying or deriving shapes from copyrighted masters.

For example, you can't legally provide a shape from a Visio Technical stencil to a user who has Visio Professional (and therefore does not have a licensed copy of the Technical stencil that contains the master).

The Visual Basic and C++ files of constants and global functions provided in the DVS (Developing Visio Solutions) folder on your Visio 5.0 CD are also copyrighted. You can include these files in your projects and use them to build executable programs, but you cannot distribute them to another developer unless he or she already has a licensed copy of a Visio product that includes those files.

The source code files for the sample programs provided in the DVS folder on your Visio 5.0 CD are intended to show how Automation works in Visio. You can study these files or use them in your programs.

For complete details about licensing of masters and Visio products, see the Visio Software License Agreement included with this book.

Copyrighting your own shapes

Copyright information for a master is displayed in the Special dialog box for the master or any instance of the master. You can add copyright information to the shapes you create by typing it in the Special dialog box. However, you can do this only once for a shape (unless you choose Undo immediately afterward). Thereafter, the copyright information for that shape cannot be changed.

To open the Special dialog box, select a shape, and then choose Special from the Format menu.

18

Using ActiveX controls in a Visio solution

You can add ActiveX controls directly to Visio 5.0 drawings to make your Visio solution interactive. For example, you might add standard Windows dialog box controls such as single-click buttons, checkboxes, or list boxes. Or, you might add custom controls that you develop or purchase to incorporate more complex functionality, such as animation.

This chapter describes how to add ActiveX controls to a Visio drawing, including how to set tab order and protect controls from inadvertent changes by the user. It describes how to handle a control's events, work with controls at run time, and distribute ActiveX controls in a Visio solution. The chapter ends with an example of controls that interact with shapes on a drawing.

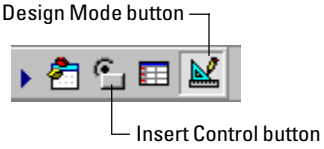
Topics in this chapter

Adding ActiveX controls to a Visio drawing

Using ActiveX controls in your Visio solutions allows you to create a user interface that is consistent with solutions based on other Windows applications. Because the controls are on the drawing page, they're nonmodal—that is, the user can work freely with both controls and Visio shapes without having to display and dismiss a Visual Basic for Applications (VBA) form.

Working in design mode

To work with ActiveX controls in a Visio drawing, you switch between design mode and run mode. In *design mode*, you can insert controls, move and size them, and set their properties. In *run mode*, you can use the controls—click a command button to run its **Click** handler, for example. For other tasks, it doesn't matter whether Visio is in design mode or run mode—all other Visio commands and tools work the same way in either mode.



The document's mode is synchronized with that of its VBA project, so both the document and its project are always in the same mode. While a document is in design mode, none of its objects (including controls) issues events.

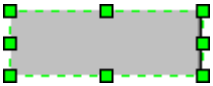
A Visio document opens in run mode by default, unless macro virus protection is set in Visio. To switch to design mode, make sure the Developer toolbar is displayed (if not, choose Toolbars from the View menu, then choose Developer). Then click the Design Mode button, which inverts to indicate that Visio is in design mode.

NOTE If macro virus protection is set in Visio, it prompts the user to enable or disable macros when a document is opened. If the user disables macros, the document opens in design mode and cannot be switched to run mode until the document is closed and reopened with macros enabled. To set macro virus protection, choose Options from the Tools menu, then check or clear Macro Virus Protection on the General tab.

Inserting a control in a drawing

Before you can insert an ActiveX control in a Visio drawing, the control must be installed on your system. Certain controls might also require that you have a design license to use them in applications that you develop.

You insert a control by selecting it in the Control dialog box, which lists all of the ActiveX controls installed on your system, including those installed by other applications. Such applications typically provide a run-time license for the ActiveX controls they contain. The run-time license entitles you to use those controls in the application that contains them, but not to insert the controls in applications that you develop. To insert such controls in your applications, you need a design license for them. For details, see "Distributing controls in a Visio solution" later in this chapter.

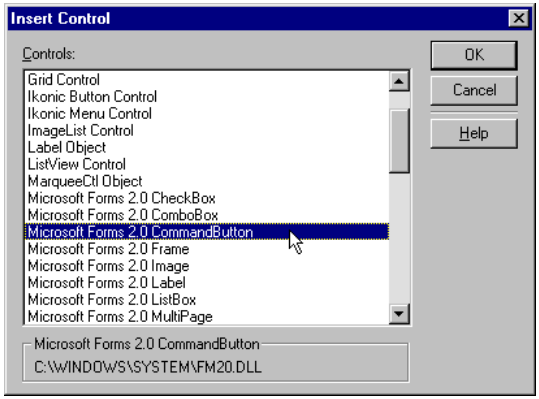


A selected control

A control activated for in-place editing

To insert an ActiveX control in a drawing:

1. Click Insert Control on the Developer toolbar.
2. In the Control dialog box, select a control. For example, Microsoft Forms 2.0 CommandButton.



3. Click OK to insert the control on the drawing page.
4. Move and size the control as needed.

   A selected control has green selection handles, just like a selected shape, and you move and size it in exactly the same way.
5. Edit the control and set its properties as needed.

   To edit a control, double-click it. A control activated for in-place editing in this way looks the same in Visio as in any ActiveX container. To set a control's properties, use the Visual Basic Editor.

After you insert a control in a drawing, you can work with it in much the same way as with a Visio shape—for example, you can cut or copy and paste the control, duplicate it with Ctrl+drag, or make it into a master by dragging it to a stencil.

Setting tabbing order of controls

When Visio is in run mode, pressing the Tab key moves the focus from one control to another on the drawing page. If you add more than one control to a drawing, you'll want the focus to move in a logical order.

The tabbing order of controls corresponds to the stacking order of the controls on the drawing page, starting with the backmost control. Initially, this is the order in which you inserted the controls in the drawing, with the most recently inserted control at the front.

To adjust the tab order, use the Bring Forward, Bring To Front, Send Backward, and Send To Back commands on the Visio Shape menu to change the stacking order of controls relative to each other.

Protecting controls from changes

When you distribute a solution that contains controls, you might want users to be able to edit the shapes in the drawing, but you typically won't want them to be able to edit the controls. You can protect controls from user changes, even in design mode, by locking the shapes and protecting the document.

To protect controls from changes:

1. Select the controls on the drawing.
2. Choose Protection from the Format menu, then check From Selection.
3. Choose Protect Document from the Tools menu, then check Shapes.
4. For added security, define a password in the Protect Document dialog box.

   The user will be able to modify the drawing but not the controls.

Handling a control's events

After you add an ActiveX control to the drawing page, you can handle the various events issued by the control—for example, if you insert a command button, you can handle its **Click** event. You handle a control's events by writing event procedures in the VBA project of the Visio drawing that contains the control, just as you would handle a Visio event.

To write an event procedure for a control:

1. Select the control from the object list in the Visual Basic Editor.
2. Select the event you want to handle from the procedure list.
3. Fill in the event procedure in the code window.

For example, the following event procedure for a command button deletes a shape in the Visio drawing when a user selects the shape's name in a listbox control and clicks the command button:

Printing a drawing without its controls

If you want the user to be able to print a drawing but not its controls, assign all of the controls to the same layer and make the layer nonprinting. For details about layers, search online help for "layers on pages."

```
Private Sub CommandButton1_Click( )
  Dim visShape As Visio.Shape
  If ListBox1.ListIndex >=0 Then
    set visShape = _
      ActivePage.Shapes(ListBox1.Text)
    visShape.Delete
  End If
End Sub
```

Working with controls at run time

An ActiveX control typically exposes properties and methods you can use at run time to work with the control programmatically. For example, a listbox control has a **ListIndex** property that returns the index of the selected item and a **Text** property that returns the text of the item at that index.

About control names

A control has two names: a Visio name and a VBA name. Initially, these names are identical, consisting of the control type plus an integer that makes the name unique. For example, the first listbox control you insert in a drawing is given the name ListBox1, and you can use this name to refer to the control in both VBA code and in Visio. (Visio follows a different naming convention for Visio shapes. For details, see "Getting information from shapes" in Chapter 13, "Getting information from Visio drawings.")

Although initially set to the same value, the two names are programmatically distinct and cannot be used interchangeably:

- You use a control's VBA object name to refer to the control in VBA code. You change this name by setting the control's **(Name)** property in the VBA properties window in the Visual Basic Editor. You cannot use a control's VBA object name to get a Shape object from a Visio collection, such as the Shapes collection; instead, you must use the control's **Shape.Name** property—for example, **ListBox1.Shape.Name**.

Using the Visio ambient properties

If you're developing ActiveX controls for use in Visio, you can take advantage of the *ambient properties* that Visio defines. A control uses an application's ambient properties to maintain a consistent appearance with other controls in a document. For example, the **BackColor** property specifies the color of a control's interior. Ambient properties are read-only.

To list the Visio ambient properties, display the shortcut menu in the Visual Basic Object Browser, choose Show Hidden Members, and then select **IVAmbients**.

- You use a control's Visio name to get the Shape object that represents the control from a Visio collection such as OLEObjects. You change this name by editing it in the Name box in the Special dialog box in Visio (select the control and choose Special from the Format menu to display this dialog box) or by setting the control's **Shape.Name** property in VBA. You cannot use a control's Visio name to refer to the control in VBA code.

For your own convenience, if you change one name from the default value, you might want to change the other name so that the control's VBA and Visio names are identical.

Getting a control from the OLEObjects collection

You can get the Shape object that represents a control from the OLEObjects collection of a Visio Document, Page, or Master object. You can also get a control from the Shapes collection of a document, page, or master, but it's much faster to use the OLEObjects collection because it contains only linked or embedded objects, whereas the Shapes collection also includes all the Visio shapes in that document, page, or master—potentially many more objects.

The OLEObjects collection contains an OLEObject that represents each linked or embedded object in a Visio document, page, or master, plus any ActiveX controls. The **Object** property of an OLEObject returns a reference to the linked or embedded object that you can use to access the object's properties and methods.

You can retrieve a control from the OLEObjects collection by its index within the collection or by the name assigned to the control in Visio. Initially, this name is identical to the value of the control's VBA object name, as described in the previous section, "About control names." For example, the following statements get a Visio shape named ListBox1:

```
Dim g_listbox as Object
Set g_listbox = _
  Document.OLEObjects("ListBox1").Object
```

If you want to perform the same operation on all controls, iterate through the OLEObjects collection and check the **ForeignType** property of each OLEObject object to see whether the **visTypeIsControl** bit is set. If (**ForeignType And visTypeIsControl**) is TRUE, the object is an ActiveX control.

Distributing controls in a Visio solution

VBA in Visio 5.0 includes the Microsoft Forms 2.0 ActiveX controls, which include standard dialog box controls such as buttons, checkboxes, text boxes, and combo boxes. You can distribute these controls most simply with a Visio solution because they are included with Visio—no special installation or additional licensing is required.

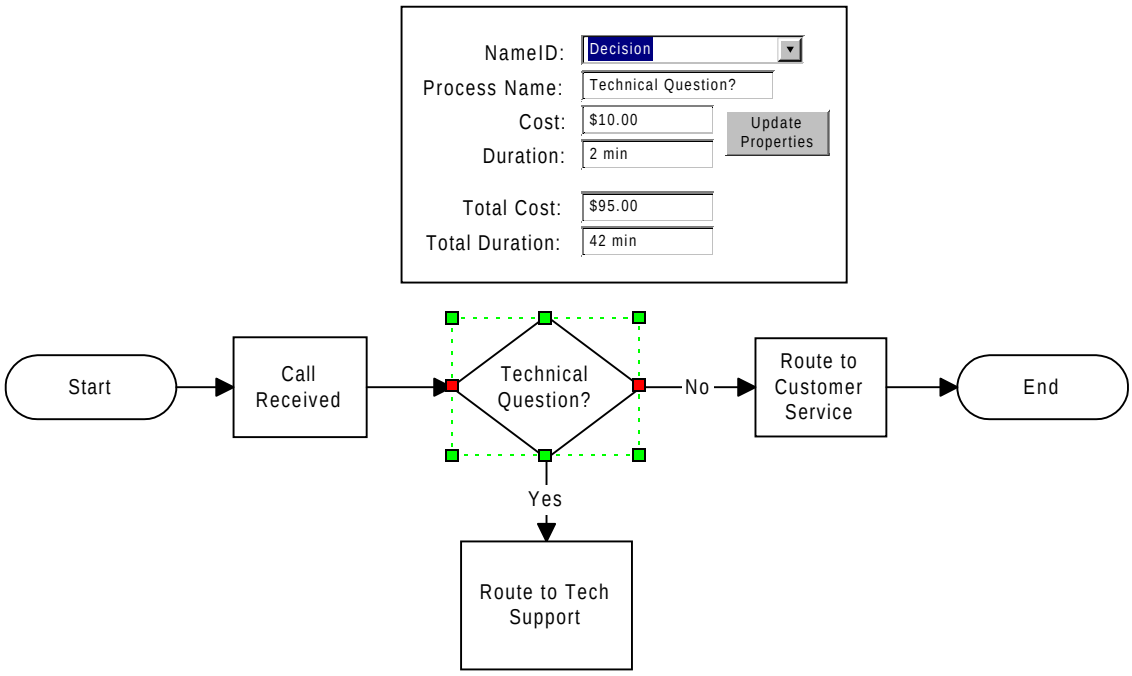
You might acquire other controls by installing Visual Basic or C++, downloading controls from the Internet, or buying third-party packages. Distributing a solution that contains such controls can be a little more complicated:

- Because the controls may or may not already be on the user's system, your solution's Setup program needs to check whether the control is already installed and, if not, install the control and register it on the user's system.
- Such controls typically come with a design-time license so you can use them in your development projects, and might require a run-time license for distribution.

For details about installing, registering, and licensing third-party controls, see the developer documentation provided with the control.

Listing shapes and custom properties in controls: an example

The drawing file VBA ACTIVEX SAMPLE.VSD is an example of a drawing that uses ActiveX controls to interact with shapes. This example contains a combo box control that lists the names of shapes in the drawing, text boxes that display the text and certain custom properties of a selected shape, and a command button that updates a selected shape with new values in the text boxes. The drawing also maintains a running total of cost and duration for all process flowchart shapes on the page, updating the totals as shapes are added, deleted, or changed.



The example uses several event handlers to accomplish these tasks:

- **Document_DocumentOpened.** Initializes controls on the page by clearing the combo box and text boxes, filling the combo box with a list of shapes on the page, and setting a variable, *theWindow*, to the Visio active window.
- **Document_ShapeAdded.** Adds a process flowchart shape to the combo box list when the user adds it to the drawing.
- **Document_BeforeSelectionDelete.** Removes a process flowchart shape from the combo box list when the user deletes it from the drawing.
- **theWindow_SelectionChanged.** Sets the shape name shown in the combo box to the shape that is selected in the drawing.
- **ComboBox1_Change.** Selects a shape in the drawing and displays its custom properties in the text boxes when the user highlights the shape's name in the combo box list.
- **CommandButton1_Click.** Updates the selected shape in the drawing with current values in the text boxes.

The following code shows the **ComboBox1_Change** event handler.

ComboBox1_Change Handler in \DVS\VBA SOLUTIONS\VBA ACTIVEX SAMPLE.VSD

```
Private Sub ComboBox1_Change()
   ' The user has clicked on an item in the list box
   Dim strName As String
   On Error GoTo Ignore:

   If (bInComboBoxChanged) Then
      ' Exit without doing anything; already responding to the initial Change event
      Exit Sub
   End If

   ' Set flag indicating the program is in the Change routine. If an error occurs
   ' after this, it skips to the Ignore label, after which the flag is reset.
   bInComboBoxChanged = True

   ' Calling DeselectAll and Select on the Window object set ComboBox1.Text
   ' (see theWindow_SelectionChanged). Save the current text before calling
   ' DeselectAll so that we know which shape to select.
   strName = ComboBox1.Text

   ' Select the item and get its properties
   ActiveWindow.DeselectAll
   ActiveWindow.Select ActivePage.Shapes(strName), visSelect

   With ActivePage.Shapes(strName)
      TextBox1.Text = .Text
      TextBox2.Text = Format(.Cells("prop.cost").ResultIU, "Currency")
      TextBox3.Text = Format(.Cells("prop.duration").Result(visElapsedMin), _
         "###0 min.")
   End With

Ignore:
   ' Set flag indicating the program is NOT in the Change handler any more
   bInComboBoxChanged = False
   Exit Sub
End Sub
```

This handler does the following:

1. Clears the selection in the drawing.
2. Selects the shape whose name corresponds to the value of the ComboBox1 control's **Text** property. If the drawing doesn't contain such a shape, the handler simply exits.
3. Sets the TextBox1 control's **Text** property to the **Text** property of the shape.
4. Sets the TextBox2 control's **Text** property to the value of the shape's Cost custom property, expressed as currency.
5. Sets the TextBox3 control's **Text** property to the value of the shape's Duration custom property, expressed as minutes.

Note the global variable *bInComboBoxChanged*, which indicates whether the **ComboBox1_Change** handler is being called for the first time. Clearing the selection and selecting a shape trigger **Window_SelectionChanged** events. However, this sample's handler for that event sets the **ComboBox1.Text** property, which triggers a **ComboBox1_Change** event and causes the **ComboBox1_Change** handler to run again. Setting *bInComboBoxChanged* the first time the handler runs allows the handler to skip the selection operations the second time, preventing the sample from going into a recursive loop.

It's also possible to prevent such loops by setting the **EventsEnabled** property of the Application object to disable event handling while the handler performs operations that would otherwise trigger events and cause handlers to run inappropriately. However, this approach is not recommended because it disables *all* events for the instance of Visio, which might interfere with other solutions running on the user's system (especially if an error in your solution prevents it from re-enabling events). Unless you are certain that your solution is the only one that will be handling Visio events, it's recommended that you use the global variable technique shown in this sample.

For the complete listing of this sample, see \DVS\VBA SOLUTIONS\VBA ACTIVEX SAMPLE.VSD on your Visio 5.0 CD.

19

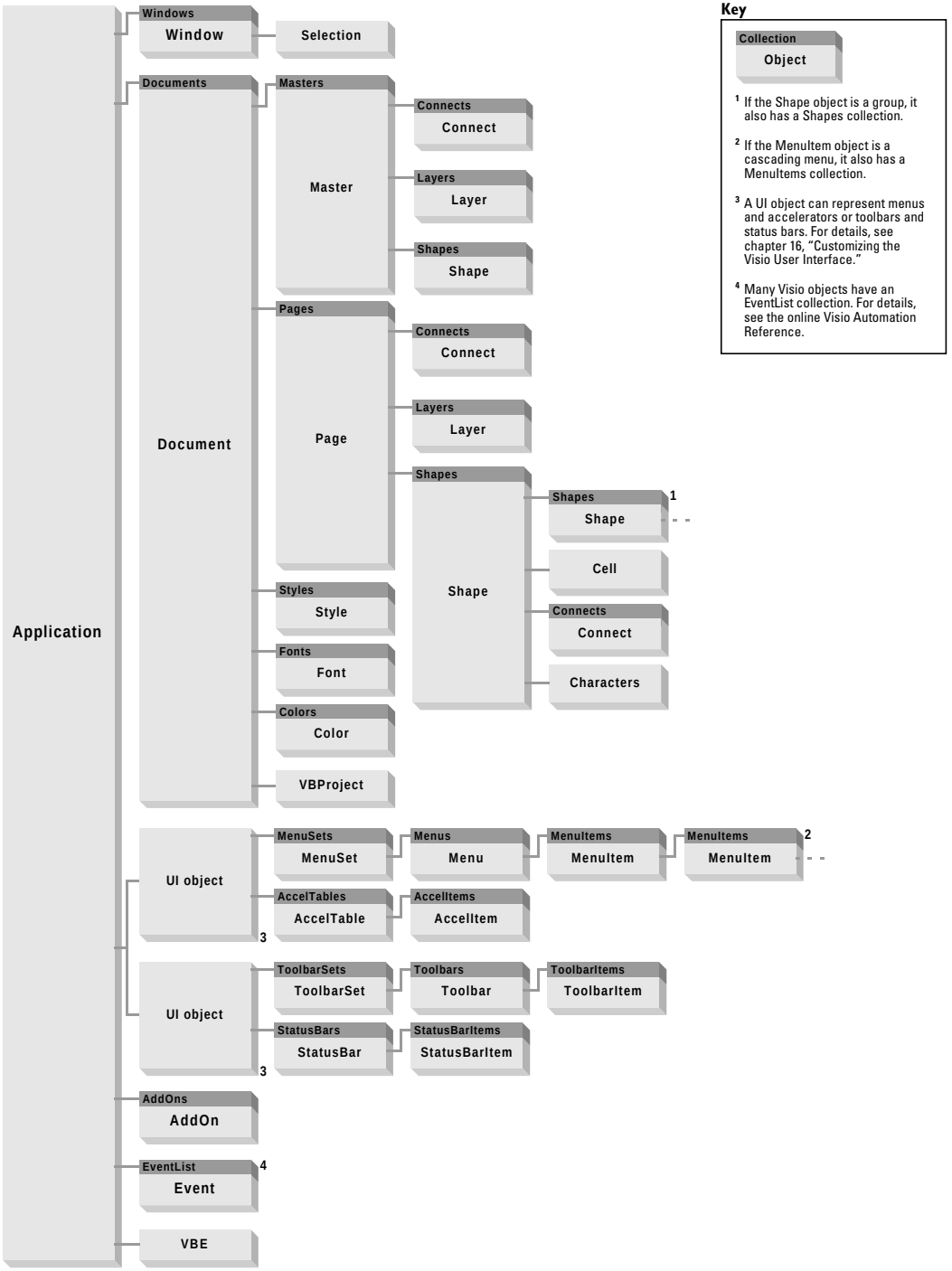
Programming Visio with Visual Basic

In previous chapters, this book has focused on writing programs in the Visual Basic for Applications (VBA) development environment within Visio. This chapter focuses on specific issues related to writing external standalone programs to control Visio using the Visual Basic development environment.

This chapter discusses using the Visio object model from an external development environment and describes how to access the Application object from an external program. It discusses creating a Visio document, handling errors, and using the Visio type library in Visual Basic projects. It also briefly describes the Visual Basic files provided in the DVS (Developing Visio Solutions) folder on your Visio 5.0 CD and the issues that arise when migrating from Visual Basic to VBA.

Topics in this chapter

The Visio object model



The Visio object model represents the objects, properties, and methods that Visio exposes through Automation. More important, it describes how the objects are related to each other. Many of the objects are used primarily to access other objects. For more information about the Visio global object, see Chapter 11, "Using Visio objects."

Getting an instance of Visio

Any external program that controls Visio through Automation must interact with an instance of Visio. Depending on the purpose of your program, you may run a new instance of Visio or use an instance that is already running.

Creating an Application object. After you declare a Visio object variable for an Application object, you can use the **CreateObject** function in a **Set** statement to create the object and assign it to the object variable, which you can then use to control the instance. For example:

```
Dim appVisio as Visio.Application
...
Set appVisio = CreateObject("visio.application")
```

Creating an Application object runs a new instance of Visio, even if other instances are already running.

You can use the global constant **visAPI** defined in VISCONST.BAS instead of *visio.application* in **CreateObject** and **GetObject** statements. For example:

```
Dim appVisio as Visio.Application
...
Set appVisio = CreateObject(visAPI)
```

Getting an Application object. You can use the **GetObject** function to retrieve an Application object for an instance of Visio that is already running. For example:

```
Dim appVisio as Visio.Application
...
Set appVisio = GetObject(, "visio.application")
```

Visio object types

You can define Visio object variables using Visio object types or the more general Object type. Declare object variables using Visio object types like this:

```
Dim appVisio as
Visio.Application
```

instead of:

```
Dim appVisio as Object
```

This increases the speed of your program. All the example code in this book uses Visio object types. To use Visio object types, you must set a reference to the Visio type library. For details, see Chapter 2, "Tools for creating solutions."

Notice the comma, which indicates that the first argument to **GetObject**—a path to a disk file—has been omitted. The comma is required because under some circumstances **GetObject** takes a file name as its first argument. To retrieve an instance of Visio, however, you must omit the file name argument, or an error will occur. For details, see **GetObject** in your Visual Basic documentation.

If more than one instance of Visio is running, **GetObject** returns the active instance. When a program is run as an add-on or by double-clicking a shape, the active instance is the one from which the program was run. Otherwise it is the instance that was most recently run or brought to the front. If no instance of Visio is running, **GetObject** causes an error.

Releasing an Application object. An application instance persists until you use the **Quit** method or until a user closes the instance. You may want to include some error handling or use events for the latter situation, which can occur unexpectedly while your program is running.

Shortcuts for working with instances. If **CreateObject** or **GetObject** fails for some reason—for example, if Visio isn't installed on the user's system when **CreateObject** is called, or if **GetObject** can't find a running instance—an error occurs. The **vaoGetObject** function in VISREG.BAS provides a convenient alternative to **CreateObject** and **GetObject** because it includes error handling for these situations:

- If an instance of Visio is already running, **vaoGetObject** assigns that instance to *g_appVisio* and returns **visOK**. (*g_appVisio* is a global variable maintained by VISREG.BAS.)
- If no instance is running, **vaoGetObject** calls **CreateObject** to run a new instance, assigns that instance to *g_appVisio*, and returns **visOK**.
- If Visio is not installed or an error occurs, **vaoGetObject** returns **visError**. (The constants **visOK** and **visError** are defined in VISREG.BAS.)

VISREG.BAS contains a library of functions that streamline working with instances of Visio. The library maintains the global variable *g_appVisio*. To use the library of functions, include VISREG.BAS in your Visual Basic project and use *g_appVisio* to refer to the Application object.

The following example uses **vaoGetObject** to get an instance of Visio. If no instance is running, it runs one. If it cannot run an instance, it displays a message box.

```
Sub appConnect()
   If vaoGetObject() <> visOK Then
      MsgBox ("Unable to run Visio.")
   End If
End Sub
```

Getting an active instance of Visio

The **GetDocName** subroutine gets the active instance of Visio and a document that is open in that instance. The subroutine follows these steps:

1. Gets an Application object that represents the active instance of Visio.
2. Gets the Application object's Documents collection.
3. Gets the first Document object from the collection.
4. Gets the Document object's **Name** property and displays the value returned by the **Name** property in a text box on a user form.

The code for this example is in \DVS\VB SOLUTIONS\DVS.BAS. If you try this example, make sure you run an instance of Visio and open a Visio document before you run the program.

GetDocName in \DVS\VB SOLUTIONS\DVS.BAS

```
Sub GetDocName()
   Dim appVisio As Visio.Application        ' Visio instance
   Dim docObj As Visio.Document             ' A Document
   Dim strDocName As String                 ' String to hold name

   ' Retrieve the current instance of Visio
   Set appVisio = GetObject(, "visio.application")

   ' Get the first document from the Documents collection
   ' The code below uses the more typical short form of the code
   ' Set docObj = Documents.Item(1)
   Set docObj = appVisio.Documents(1)

   ' Get the Document Name property
   strDocName = docObj.Name

   ' Set the Text property of the text box to the document name
   UserForm1.TextBox1 = strDocName
   UserForm1.TextBox1.Enabled = False

   ' Set the label text and show the form
   UserForm1.Label1.Caption = "Document name:"
   UserForm1.Show

End Sub
```

Notice the use of object variables to hold references to the Visio objects used in the program—starting with the Application object. The external program progresses from Application object to Documents collection to Document object. For more information about Visio objects, see Chapter 11, "Using Visio objects."

Creating an application object

The Visual Basic subroutine creates an Application object that runs an instance of Visio and creates a drawing by opening a template and stencil. This program follows these steps:

1. Runs an instance of Visio.
2. Creates a new document based on the VB SOLUTIONS.VST template.

Get an instance's window handle

You can exert more control over an instance by getting its window handle. After you get the window handle, you can manage the instance's frame window as you would manage any other frame window from a Windows application. For example, you might minimize the instance while your program is creating a complex drawing to save time repainting the screen.

The Application object's **WindowHandle32** property returns the window handle (HWND) for the main—or frame—window of an instance. You can use the HWND with standard Windows API calls to obtain other handles. For example, you can pass the window handle to **GetWindowTask** to get the Visio task handle.

For details about using Windows API calls in a Visual Basic program, see your Visual Basic documentation. For details about the calls themselves, see the Windows API Reference (provided with Visual Basic as an online help file).

3. Drops an instance of the Rectangle master from the VB SOLUTIONS.VSS stencil on the drawing page.
4. Sets the text of the rectangle shape on the drawing page to "Hello World!"
5. Saves the document.
6. Closes the instance of Visio.

The code for this example is also in \DVS\VB SOLUTIONS\DVS.BAS. Notice that this example, unlike the previous one, actually creates an Application object that runs a new instance of Visio. It also creates a Visio document. For more information about creating or adding Visio document objects, see "Creating a Visio document" later in this chapter.

If you try this example, make sure the VB Solutions template and stencil are installed in the Visio Templates and Stencils paths respectively. It doesn't matter if an instance of Visio is already running; the program will run a new one.

HelloWorld in \DVS\VB SOLUTIONS\DVS.BAS

```
Sub HelloWorld ()
   'Object variables to be used in the program.
   Dim appVisio As Visio.Application   'Instance of Visio
   Dim docsObj As Visio.Documents      'Documents collection of instance
   Dim docObj As Visio.Document        'Document to work in
   Dim stnObj As Visio.Document        'Stencil that contains master
   Dim mastObj As Visio.Master         'Master to drop
   Dim pagsObj As Visio.Pages          'Pages collection of document
   Dim pagObj As Visio.Page            'Page to work in
   Dim shpObj As Visio.Shape           'Instance of master on page

   'Run an instance of Visio and create a document based on the Basic template.
   'A new document always has one page, whose index in the Pages collection is 1.
   Set appVisio = CreateObject("visio.application")
   Set docsObj = appVisio.Documents
   Set docObj = docsObj.Add("VB Solutions.vst")
   Set pagsObj = appVisio.ActiveDocument.Pages
   Set pagObj = pagsObj.Item(1)
```

```
    'Create a document based on the VB Solutions template which automatically opens
    'the VB Solutions stencil
    Set stnObj = appVisio.Documents("VB Solutions.vss")
    Set mastObj = stnObj.Masters("Rectangle")

    'Drop the rectangle in the approximate middle of the page.
    'Coordinates passed with Drop are always inches.
    Set shpObj = pagObj.Drop(mastObj, 4.25, 5.5)

    'Set the text of the rectangle.
    shpObj.Text = "Hello World!"

    'Save the drawing and quit Visio. The message pauses the program
    'so you can see the Visio drawing before the instance closes.
    docObj.SaveAs "hello.vsd"
    MsgBox "Drawing finished!", , "Hello World!"
    appVisio.Quit

End Sub
```

CreateObject is a Visual Basic function that creates an Automation object—in this example, **CreateObject** runs a new instance of Visio and returns an Application object that represents the instance, which is assigned to the variable *appVisio*. The next six **Set** statements obtain references to the other objects used in this program by getting properties of objects obtained earlier. Notice again the progression through the Visual Basic–specific Visio object model from Application object to Documents collection, to Document object, to Pages collection, to Page object.

Set docObj = docsObj.Add("VB Solutions.vst") uses the **Add** method to open a template and add it to the Documents collection. For more information about adding Document objects, see the next section, "Creating a Visio document."

The statement *appVisio.Quit* uses the **Quit** method to close the instance of Visio assigned to *appVisio*.

Creating a Visio document

After you get an Application object that represents an instance of Visio, the next step is to create or open a document.

To create a new document from a program, you first get the **Documents** property of the Application object to get its Documents collection, then use the **Add** method of the Documents collection to create the document. To base the new document on a template, supply the file name of that template as an argument to **Add**. For example, the following statement creates a new document based on the Basic Diagram template provided with Visio:

```
Dim appVisio as Visio.Application
Dim docObj as Visio.Document
Set appVisio = CreateObject("visio.application")
Set docObj = appVisio.Documents.Add("basic _
diagram.vst")
```

If you don't specify a path with the template file name, Visio searches the folders shown in the Templates box on the File Paths tab. To find out the current path settings, get the Application object's **TemplatePaths** property. For details about displaying and using the File Paths tab, search online help for "file paths tab."

The Application object has a corresponding property for each of the folders shown on the File Paths tab. For example, the **TemplatePaths** property corresponds to the Templates folder on the tab. You can get any of these properties to find the current path, or you can set the property to change the path. For details, see the online Visio Automation Reference.

In the previous example, the new document has the drawing scale, styles, and document settings defined in BASIC DIAGRAM.VST. This template happens to have a stencil—BASIC SHAPES.VSS—in its workspace, so creating the document also opens that stencil as read-only in a stencil window and adds the stencil file to the Documents collection of the instance.

To create a new document without basing it on a template, use a null string ("") in place of the file name argument. A document created in this way has the Visio default drawing scale, styles, and other document settings. No stencils are opened.

Handling errors

When an error occurs during program execution, Visual Basic generates an error message and halts execution. You can prevent many errors by testing assumptions before executing code that will fail if the assumptions aren't valid. You can trap and respond to errors by using the **On Error** statement in your program. For details about **On Error**, see your Visual Basic documentation.

Errors can arise from a variety of situations. This section discusses one common situation specific to running an instance of Visio from an external standalone program. For more information about common situations in which errors can occur, see "Handling errors" in Chapter 11, "Using Visio objects."

Make sure the program is running in the right context

If you've decided which context a program will run in, you can make some assumptions about the environment. For example, if you're writing an add-on, you can probably assume that an instance of Visio is already running.

However, if your program requires a running instance of Visio, it's a good idea to make sure the instance is there. For an example, see the **vaoGetGIO** function in VISREG.BAS. This routine uses **GetObject** to retrieve the active instance of Visio. If there is no active instance, **GetObject** causes an error. The **vaoGetGIO** function uses **OnError** to trap the error and returns FALSE instead of halting execution.

Or, if your program requires an open document, make sure one is available. For example:

```
Dim appVisio as Visio.Application
Dim docObj As Object
...
Set docObj = appVisio.ActiveDocument
If docObj is Nothing Then
   'there are no documents open, handle the error
Else
   'there is a document open, continue processing
End If
```

Visio file extensions

Visio recognizes four different file extensions—.VSD, .VSS, .VST, and .VSW—which identify drawing, stencil, template, and workspace files, respectively. You might assume that these files have different formats, but they don't—the file extension simply determines how Visio opens the document and what it displays. For details about Visio files, see Chapter 2, "Tools for creating solutions."

Interpreting the command string Visio sends to your program

When an executable program (.EXE) is run, it receives a command string from the environment that launched the program. When a Visio library (.VSL) is run, Visio passes the same command string to the .VSL file in an argument structure with the Run message. For details, see Chapter 20, "Programming Visio with C++."

The command string sent by Visio identifies Visio as the environment that launched the program, and may contain values (described below) that you can use to retrieve certain objects in addition to arguments for the program. The values in the string depend on how the program was run—from the Macro submenu or from a ShapeSheet formula, with arguments or without.

Running the program from the Macro submenu. If the program is run from the Macro submenu—the user chooses it from either the Macro submenu or the Macros dialog box—the Visio command string looks like this:

```
"/visio=instanceHandle32"
```

The significant portion of this command string is */visio*, which you can use to confirm that the program was run from Visio and not some inappropriate environment. *instanceHandle* is the Windows handle of the Visio instance from which the program was run—an integer (4-byte value). You usually won't do anything with the instance handle, although you might compare it with the handle of the Application object you're working with to make sure they match. For details about instance handles, see the **InstanceHandle32** properties in the online Visio Automation Reference.

Running the program when a formula is evaluated. If a shape formula uses a RUNADDON function to run the program when that formula is evaluated, the command string Visio sends to the program looks like this:

```
"/visio=instanceHandle32 /doc=docIndex / _
page=pagIndex /shape=NameID"
```

Various parts of the command string identify objects that contain the shape whose formula ran the program. *docIndex* is the index of the Document object, and *pagIndex* the index of the Page object. You can use these values to get the corresponding objects from their respective collections. For example:

Parsing the command string

To retrieve and *parse* a command string, use the functions provided by your development environment for that purpose. In Visual Basic, for example, use **Command** to retrieve the command string and string functions such as **Mid** and **StrComp** to parse it.

Parsing is the process of separating statements into syntactic units—analyzing a character string and breaking it down into a group of more easily processed components.

```
Dim docObj as Visio.Document
docObj = Visio.Application.Documents.Item(docIndex)
```

NameID is the **NameID** property of the shape whose formula was evaluated. You can use this value to get the corresponding Shape object. For example:

```
Dim shpObj as Visio.Shape
shpObj = Visio.Application.Documents(docIndex). _
Pages(pagIndex).Shapes(NameID)
```

If the cell that was evaluated is in a master rather than in a shape on a drawing page, the command string looks like this:

```
"/visio=instanceHandle32 /doc=docIndex /master= _
masterIndex/shape=NameID"
```

In this case, you would get the Shape object as follows:

```
Dim shpObj as Visio.Shape
shpObj = Visio.Application.Documents(docIndex). _
Masters(masterIndex).Shapes(NameID)
```

If the cell is in a style rather than a shape or a master, the command string looks like this:

```
"/visio=instanceHandle32 /doc=docIndex / _
style=NameID"
```

Running the program with arguments. If a cell formula uses a RUNADDONWARGS function to run the program, the command string includes the specified arguments:

```
"/visio=instanceHandle /doc=docIndex /page=pagIndex _
/shape=Sheet.ID arguments"
```

If a custom menu command or toolbar button's **AddOnArgs** property contains arguments, the command string looks like this:

```
"/visio=instanceHandle arguments"
```

The *arguments* string can be anything appropriate for your program.

Note, however, that the entire command string is limited to 127 characters including flags (*/visio=*, */doc=*, */page=*, and */shape*, for example), so in practice the arguments should not exceed 50 characters. If the entire command string exceeds 127 characters, an error occurs and Visio will not run the program.

Running the program from the Startup folder. If the program is run from the Visio Startup folder, the command string also includes the flag */launch*.

Interacting with other programs

While your program is running, you can find out which programs are available to Visio, or install another program, by getting the Addons collection of an Application object. This collection contains an Addon object for each program in the folders specified by the Application object's **AddonPaths** and **StartupPaths** properties or added dynamically by other programs.

The programs represented by Addon objects are listed on the Macro submenu and in the Macros dialog box. You can add a program by using the **Add** method of the Application object's Addons collection. The newly added program remains in the collection until the instance of Visio is closed.

```
Dim addonsObj as Viso.Addons
Dim addonObj as Visio.Addon
Set addonsObj = Visio.Application.Addons
Set addonObj = addonsObj.Add("c:\temp\myprog.exe")
```

NOTE No object is returned if the program is a .VSL file.

Get the **Name** property of an Addon object to find out its name; get its **Enabled** property to find out whether it can be run. An .EXE file is always enabled, but a program in a Visio library may not be. For details, see Chapter 20, "Programming Visio with C++."

To run another program, use the **Run** method of the corresponding Addon object, and include any necessary arguments or a null string.

For more details about Addon objects, their methods, and their properties, see the online Visio Automation Reference.

Using the Visio type library

The Visio type library contains descriptions of the objects, methods, properties, events, and constants that Visio exposes. You use the Visio type library to define Visio object types and constants in your program. Using Visio object types increases the speed of your program.

The Visio type library and VISCONST.BAS

Earlier versions of Visio did not include a type library, so all constants were defined in VISCONST.BAS. Both VISCONST.BAS and the Visio type library contain global constants. When programming with Visio 5.0, you can set a reference to the Visio type library or include VISCONST.BAS in your project. To set a reference to the Visio type library in Visual Basic, choose References from the Tools menu and select the Visio type library in the Available References list.

If you use VISCONST.BAS instead of the Visio type library, you cannot use Visio object types—you must use the Object variable type. For example, you cannot use *Dim docsObj as Visio.Documents* when defining variables. You must use *Dim docsObj as Object*.

IMPORTANT The examples in this book assume that you have a reference set to the Visio type library.

Using global constants

Both VISCONST.BAS and the Visio type library contain global symbolic constants defined for arguments and return values of properties and methods. Most arguments to properties and methods are numeric values. Using these constants can make your code easier to write and to read.

For example, suppose you want to find out what type of window—drawing, stencil, ShapeSheet, or icon editing—a Window object represents. The **Type** property of a Window object returns an integer—1, 2, 3, or 4—that indicates the window's type. If you set a reference to the Visio type library or include VISCONST.BAS in your project, you can use the constants **visDrawing**, **visStencil**, **visSheet**, or **visIcon** instead of 1, 2, 3, or 4 to check the window's type.

The constants in VISCONST.BAS are grouped by usage. For a list of constants used by a particular method or property, see that method or property in the Visio type library or the online Visio Automation Reference.

Using the Visual Basic files provided on your Visio 5.0 CD

The DVS (Developing Visio Solutions) folder on your Visio 5.0 CD contains the code examples discussed in this chapter and several Visual Basic files you can use to develop programs that control Visio. These files are in the \DVS\LIBRARIES\VB folder. You can include these files in your Visual Basic project or just copy the code you need.

NOTE In Visual Basic 5.0, a program that contains event sinks must be set up as an ActiveX EXE project, and its **Instancing** property must be set to **MultiUse**. The DVS folder on your Visio 5.0 CD contains two sets of event sink samples: one for Visual Basic 5.0 and one for Visual Basic 4.0.

Migrating from Visual Basic to VBA

If you are thinking about migrating from Visual Basic to VBA, here are a few issues to keep in mind.

Who will use your solution and which version of Visio do they use? VBA programs are not compatible with earlier versions of Visio. If you open a document created with Visio 4.5 or later in Visio 4.0, Visio opens the drawing, but the VBA programs are not accessible—there is no Macro menu. Users won't be able to run your VBA program.

Visio object types are not compatible with earlier versions of Visio. If your users are using earlier versions of Visio, don't use the Visio type library or Visio object types. To use Visio constants in your program, include VISCONST.BAS in your project.

Remove CreateObject, GetObject, and vaoGetObject references from your program. You don't need these Application object references when programming in the VBA development environment in Visio. If you are programming in another application's VBA development environment, such as Microsoft Excel, you still need these references to get or create an instance of Visio, but when programming with Visio, Visio is already running. The Visio global object represents the active instance of Visio.

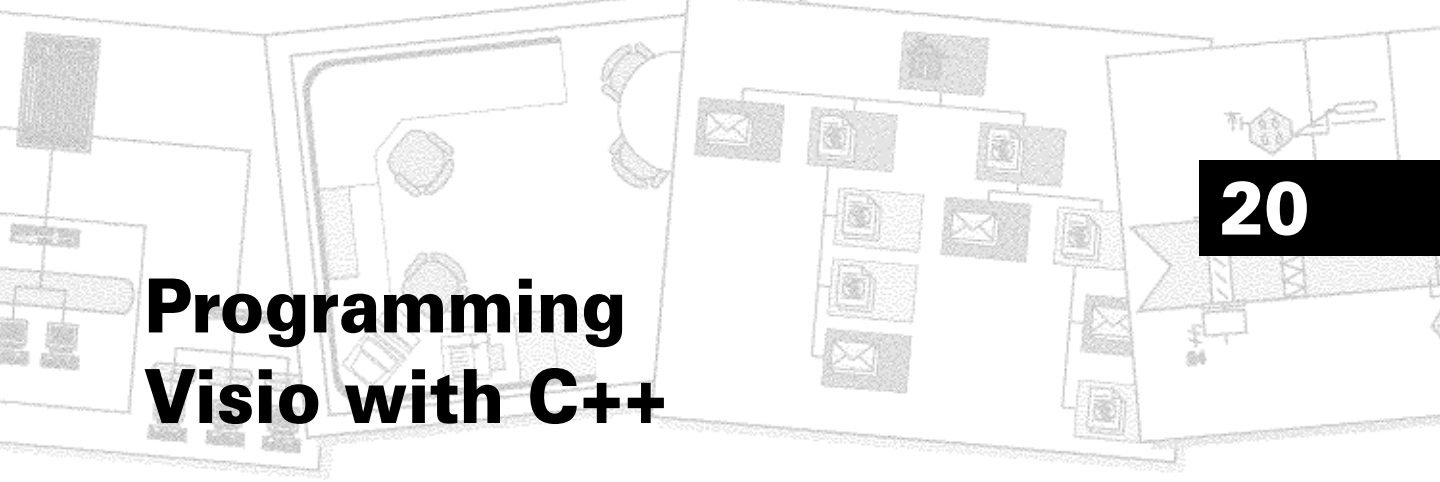
Including files in your project

You can automatically include files such as VISCONST.BAS or VISREG.BAS in all your projects by modifying AUTOLOAD.MAK. For information about modifying AUTOLOAD.MAK, see your Visual Basic documentation.

Alternatively, you can just insert the VISCONST.BAS or VISREG.BAS file in a project by choosing Add File from the File menu.

Transfer code. What components does your code use? Does it use custom controls that aren't included in VBA? Does it use VB forms? Find out if VBA supports the forms and the custom controls. If it does, you can import the forms from your Visual Basic projects into a VBA project and add any custom controls. If it doesn't, you could create a new user form in VBA and copy and paste between Visual Basic and VBA project items. Also, remove any methods in your program that create or get an instance of Visio. These are unnecessary when programming in VBA within Visio.

Store your program in a Visio file. VBA programs are stored in Visio files. If you store your VBA program with a template that opens the stencils containing the shapes you use in your program, you don't have to open the template and stencils in your program—they're already open, just as an instance of Visio is already running.



20

Programming Visio with C++

Any client that supports the OLE Component Object Model (COM) can access and manipulate Visio objects. Several commercially available development environments, such as Visual Basic, conceal the details of COM, which appeals to many developers. But if you are prepared to work more closely with COM, you can use C or C++ to develop programs that control Visio.

This chapter discusses how Visio exposes objects to Automation in terms of COM. It describes some basic support services provided by Visio that ease the task of developing C++ programs that control Visio. It then explains how to develop a Visio library (VSL), a special kind of dynamic-link library that is loaded at run time by Visio. For details about recompiling existing programs to use the new support services or about programming Visio with C, see the file README.TXT in \DVS\LIBRARIES\C-CPP.

NOTE This chapter assumes that you are familiar with OLE programming concepts, including COM, obtaining pointers to interfaces, and calling interface functions. It also assumes that you are familiar with the C++ programming language. For information about OLE, see the OLE documentation in the Microsoft Platform Software Development Kit (SDK). For information about C++, see your C++ documentation.

16-bit development issues

Visio is a 32-bit application that requires Windows 95 or Windows NT. The C and C++ support files provided with Visio 5.0 evolved from versions that support both 16-bit and 32-bit development with earlier versions of Visio. However, these files have been tested only for 32-bit development with Visio 5.0, and their behavior in 16-bit environments is not guaranteed. For more information about 16-bit development issues and Visio, see the file README.TXT in \DVS\LIBRARIES\C-CPP.

Topics in this chapter

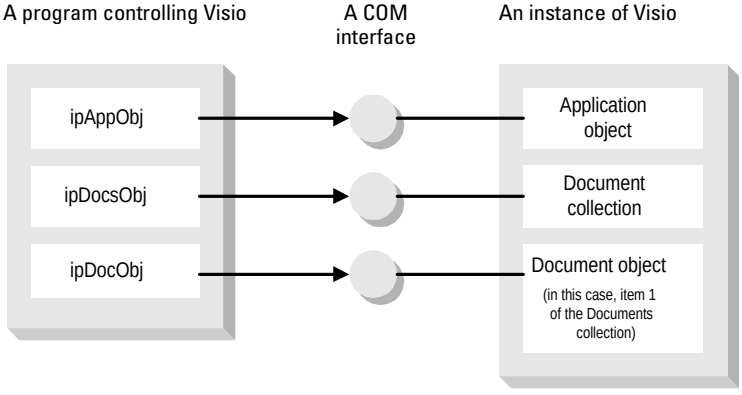
How Visio exposes objects

The objects Visio exposes are Component Object Model (COM) objects. The concepts of an interface on an object and a reference to an interface are fundamental to understanding COM. If you use the C++ files provided with Visio 5.0 and described later in this chapter, you won't need to program at this level. However, it can help to have a general understanding of what's happening behind the scenes.

To illustrate an interface on an object and a reference to an interface, here is a simple example, expressed in pseudocode:

```
ipAppObj = <reference to an interface on a Visio application object>
ipDocsObj = ipAppObj->Documents()       // Get documents collection.
ipDocObj = ipDocsObj->Item(1)           // Get first document.
```

The program state after this code executes is shown in the following illustration, which uses the common conventions for showing COM objects. The controlling program has obtained references to interfaces on three objects exposed by Visio. The arrows are the references, the circles are the interfaces, and the boxes inside the Visio instance are the objects.



The program state after getting a Document object

OLE provides many kinds of interfaces, such as those that support document linking and embedding or persistent data storage. An OLE interface pointer refers to data that represents the object that owns the interface. An interface also refers to an array of functions that perform the actions defined in that interface for that object. Once you have a reference to an interface on an object, you can call the methods defined in that interface for that object.

Methods vs. Visual Basic properties

Note the similarities between the assignments in this example and the object assignments in Visual Basic. You can extrapolate from this example—given a reference to an interface on a Document object, the program can obtain, in like fashion, a reference to an interface on a Page object, then a Shape object, and so on. The properties and methods provided by these objects are exactly the same as those discussed in earlier chapters of this book.

The interfaces that Visio exposes are *dual interfaces*. In a dual interface, the first entries in the interface are identical to the entries in a standard **IDispatch** interface, the principal interface used to implement Automation. The **IDispatch** methods are followed by entries that correspond to the methods and properties exposed by the object. A dual interface is so called because its methods or properties can be called either indirectly through **IDispatch** methods, or directly through the "dual" methods.

IDispatch functions define a protocol that allows late binding—that is, binding that occurs at run time—between Automation controllers and Automation servers. However, if an Automation server provides a type library and implements dual interfaces (as Visio does), it enables early binding—that is, binding that occurs at compile time. This typically results in improved performance on the part of the Automation controller, because the program makes fewer calls at run time to invoke a method. For more information about dual interfaces, see the OLE documentation in the Microsoft Platform SDK.

C++ support in Visio

The Automation interfaces on Visio objects are defined in VISIO.H, which is in \DVS\LIBRARIES\C-CPP\VAO_INC. This file contains a standard OLE interface definition for each Visio object. To control Visio through Automation from a C++ program, include VISIO.H in your project source files.

Visio 5.0 also provides services in the form of wrapper classes that make the job of programming Visio using C++ easier. A *wrapper class* is so called because it encapsulates, or "wraps," the programming involved in certain tasks, such as getting and releasing interface pointers and working with strings. The basic benefit you'll gain by using these classes is that they keep track of **AddRef** and **Release** calls for you, using C++ constructors, destructors, and assignment operators. They also automatically wrap any arguments or return values with a wrapper class when appropriate.

Sample projects

In addition to the files you'll use in your program, the C-CPP folder contains sample projects that illustrate the use of wrapper classes and event sinks.

The README.TXT file in this folder gives more details on the contents of the folder and instructions on how to build the sample projects. You may find it helpful to study these projects before developing your own programs.

Using the wrapper classes

To use the wrapper classes, include VISIWRAP.H in your project source files. This file is provided in the folder \DVS\LIBRARIES\C-CPP\VAO_INC. If you include VISIWRAP.H, you do not need to include VISIO.H. The wrapper classes observe the following conventions:

- Wrapper class names are prefixed with "CVisio." For example, the wrapper class for a Visio Page object is **CVisioPage**.
- Properties are accessed through methods that are prefixed with "get" to read the property or "put" to set the property. For example, the access methods for the **Name** property are **getName** and **putName**. (In VISIO.H, the corresponding methods include an underscore between the prefix and the method name: **get_Name** and **put_Name**.) To find out what these methods do, search the online Automation Reference for "Name."

A program that uses the Visio wrapper classes might look like the following example. This program, from the sample GENERIC.CPP, creates a new document based on SAMPLE.VST, drops two masters, and connects them. The function **vaoGetObjectWrap**, defined in VISIWRAP.H, gets the instance of Visio if one is running and, if not, runs an instance. For conciseness, error handling has been omitted.

This example also uses the helper classes **VBstr** and **VVariant**, which are defined in HELPERS.H.

- **VBstr**. A class that simplifies working with BSTRs, which are strings passed through Automation.
- **VVariant**. A class that simplifies working with VARIANTs, which are the Automation counterpart of C++ unions.

The file VISIWRAP.H includes HELPERS.H, so if you're using the wrapper classes you can use the helper classes also. For more information about the helper classes, see the comments in HELPERS.H.

\DVS\LIBRARIES\C-CPP\SAMPLES\GENERIC.CPP

```
#include "visiwrap.h"
extern “C” int RunDemo(void)
{
  HRESULT              hr=NOERROR
  CVisioApplication    app;
  CVisioDocuments      docs;
  CVisioDocument       doc;
  CVisioPages          pages;
  CVisioPage           page;
  CVisioShape          shape;
  CVisioShape          shape1;
  CVisioMasters        masters;
  CVisioMaster         master;
  CVisioDocument       stencil;
  CVisioCell           cell;
  CVisioCell           cell1;
  ...
  if (VAO_SUCCESS != vaoGetObjectWrap(app))
    goto CU;  //Error handling
  ...
  // Add a new document based on "sample.vst" and get the drawing page
  hr= app.Documents(docs);
  hr= docs.Add(VBstr("sample.vst"), doc);//VBstr is the helper class for type BSTR
  hr= doc.Pages(pages);
  hr= pages.Item(VVariant(1L), page);    //VVariant is the helper class for type VARIANT

  // Get the stencil and the first master to drop.
  hr= docs.Item(VVariant(“sample.vss”), stencil);
  hr= stencil.Masters(masters);
  hr= masters.Item(VVariant(“Executive”), master);
  hr= page.Drop(master, 6.0, 6.0, shape);

  // Get the second master and drop it.
  hr= masters.Item(VVariant(“Position”), master);
  hr= page.Drop(master, 3.0, 3.0, shape1);

  // Connect the two shapes on the drawing page
  hr= shape.Cells(VBstr(“Connections.X4”), cell);
  hr= shape1.Cells(VBstr(“Controls.X1”), cell1);
  hr= cell1.GlueTo(cell);
...
}
```

The interfaces behind the wrappers

The file VISIO.H defines the objects exposed by Visio in standard OLE interface declaration syntax. The wrapper classes defined in VISIWRAP.H call the methods of these interfaces. For example, the **CVisioApplication** wrapper class is declared in VISIWRAP.H as shown here.

CVisioApplication in \DVS\LIBRARIES\C-CPP\VAO_INC\VISIWRAP.H

```
class FAR CVisioApplication : public CVisioUnknown
{
VW_PUBLIC:
  CVisioApplication() : CVisioUnknown() { }
  CVisioApplication(const CVisioApplication& other) : CVisioUnknown(other) { }
  CVisioApplication(const ::IVApplication FAR * other) : CVisioUnknown((LPUNKNOWN)other) { }
  const CVisioApplication FAR & operator=(const CVisioApplication FAR &other)
  {
    if (&other != this)
      CopyIP(other.m_pUnk);
    return *this;
  }
  const CVisioApplication FAR & operator=(const ::IVApplication FAR * other)
  {
    if ((LPUNKNOWN)other != m_pUnk)
      CopyIP((LPUNKNOWN)other);
    return *this;
  }
  virtual ~CVisioApplication() { }
  ::IVApplication FAR * GetIP() const { return (::IVApplication FAR *)m_pUnk; }
  operator ::IVApplication FAR * () { return (::IVApplication FAR *)m_pUnk; }

VW_PUBLIC:
  HRESULT ActiveDocument(CVisioDocument FAR &rWrap);
  HRESULT ActivePage(CVisioPage FAR &rWrap);
  HRESULT ActiveWindow(CVisioWindow FAR &rWrap);
  HRESULT Application(CVisioApplication FAR &rWrap);
  HRESULT Documents(CVisioDocuments FAR &rWrap);
...
};
```

The corresponding Application object interface is declared in VISIO.H as follows.

IVApplication in \DVS\LIBRARIES\C-CPP\VAO_INC\VISIWRAP.H

```
#define INTERFACE IVApplication
DECLARE_INTERFACE_(IVApplication, IDispatch)
{
BEGIN_INTERFACE
#ifndef NO_BASEINTERFACE_FUNCS
  /* IUnknown methods */
  STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
  STDMETHOD_(ULONG, AddRef)(THIS) PURE;
  STDMETHOD_(ULONG, Release)(THIS) PURE;

  /* IDispatch methods */
  STDMETHOD(GetTypeInfoCount)(THIS_ UINT FAR* pctinfo) PURE;

  STDMETHOD(GetTypeInfo)(
  THIS_
  UINT itinfo,
  LCID lcid,
  ITypeInfo FAR* FAR* pptinfo) PURE;

  STDMETHOD(GetIDsOfNames)(
  THIS_
  REFIID riid,
  OLECHAR FAR* FAR* rgszNames,
  UINT cNames,
  LCID lcid,
  DISPID FAR* rgdispid) PURE;

  STDMETHOD(Invoke)(
  THIS_
  DISPID dispidMember,
  REFIID riid,
  LCID lcid,
  WORD wFlags,
  DISPPARAMS FAR* pdispparams,
  VARIANT FAR* pvarResult,
  EXCEPINFO FAR* pexcepinfo,
  UINT FAR* puArgErr) PURE;
#endif
```

(Code sample continued on next page)

IVApplication (continued)

```
/* IVApplication methods */
   STDMETHOD(get_ActiveDocument)(THIS_ IVDocument FAR* FAR* lpdispRet) PURE;
   STDMETHOD(get_ActivePage)(THIS_ IVPage FAR* FAR* lpdispRet) PURE;
   STDMETHOD(get_ActiveWindow)(THIS_ IVWindow FAR* FAR* lpdispRet) PURE;
   STDMETHOD(get_Application)(THIS_ IVApplication FAR* FAR* lpdispRet) PURE;
   STDMETHOD(get_Documents)(THIS_ IVDocuments FAR* FAR* lpdispRet) PURE;
...
};
```

Every object exposed by Visio has a similar declaration in VISIO.H. The various macros in this declaration are common OLE fare that allow VISIO.H to be included in either C or C++ source files. If you include VISIO.H, you'll also need to include WINDOWS.H and OLE2.H.

The first seven methods in this and every Visio interface are **QueryInterface**, **AddRef**, and **Release** (for **IUnknown**), followed by **GetTypeInfoCount**, **GetTypeInfo**, **GetIDsOfNames** and **Invoke** (for **IDispatch**). For details about these standard OLE methods, see your OLE documentation. The remaining methods are those that are exposed by the Visio Application object. These methods correspond to the methods and properties described elsewhere in this book for use with Visual Basic programs.

To learn more about a method, look in the online Automation reference. For example, to find more about the **get_ActiveDocument** method declared above, search the online Automation reference for "ActiveDocument."

Obtaining a Visio Application object

The sample program in GENERIC.CPP (shown earlier in this chapter) begins with the following code:

```
CVisioApplication    app;
if (VAO_SUCCESS != vaoGetObjectWrap(app))
   goto CU;  //Error handling
```

This pebble starts the avalanche. To do anything with Visio you need an Application object, which is what **vaoGetObjectWrap** gets.

vaoGetObjectWrap calls the **vaoGetObject** function, which is declared in IVISREG.H and implemented in IVISREG.CPP. If you're not using the wrapper classes, you can call **vaoGetObject** directly. Look at the source code to see what **vaoGetObject** actually does.

The services defined in IVISREG.H for working with an instance of Visio are equivalent to those provided by the VISREG.BAS file supplied for use with Visual Basic. In particular, this file provides the wherewithal to launch a new Visio instance or establish an Application object for the active instance of Visio.

Values returned by Visio methods

Every method declared in VISIO.H is declared as a STDMETHOD. This means it returns an HRESULT indicating whether the method executed successfully. The HRESULT returned by a method in VISIO.H is passed along by the equivalent method of the corresponding wrapper class defined in VISIWRAP.H.

If a method succeeds, it returns NOERROR. A common practice is to check a method's result by using SUCCEEDED(*hResult*). The sample program does this in a macro called **check_valid** shown later in this section.

Many methods also produce an output value independently of the HRESULT returned by every method. For example, the **ActiveDocument** method of the **CVisioApplication** wrapper class produces a reference to a Document object. By convention, a method's output value is written to the method's last argument. Thus the last argument passed to **ActiveDocument** is a reference to a **CVisioDocument** object where the method can return a reference to the Document object.

Object references. Many methods return an object reference as their output value. This value is really an OLE interface pointer, which, like any interface pointer, must eventually be released.

- If you're using wrapper classes, the value returned is an object of another wrapper class—such as the **CVisioDocument** mentioned above—in which the interface pointer is packaged. When the object goes out of scope, the Visio interface pointer it holds is automatically released.

- If you're not using wrapper classes, the interface pointer is held directly by your program, which must explicitly release the pointer at the appropriate time.

If a method that returns an object reference fails, the output value again depends on whether you're using wrapper classes.

- If you're using wrapper classes, you'll still get an object of the appropriate wrapper class, but the interface pointer held by the object is NULL. Calling the **IsSet** function on that object will return FALSE.
- If you're not using wrapper classes, the interface pointer is NULL, so you can simply check for that.

Even if the method succeeds, you may still need to check the output parameter. For example, if **ActiveDocument** is called when no documents are open, it returns an HRESULT of success and a NULL interface pointer (wrapped or not). The reasoning here is that an error did not occur—having no documents open is a perfectly valid state that the caller should account for. The various **Active*** methods behave in this manner, and you should verify that their output values are not NULL before proceeding. The various **Item** and **Add** methods, however, always return a non-NULL reference if they succeed.

The **check_valid** macro, defined in GENERIC.CPP, checks both possibilities. A function using **check_valid** must provide a *CU* label where it performs cleanup tasks.

```
#define check_valid(hr, obj)              \
  if(!SUCCEEDED(hr) || !((obj).IsSet()))  \
    goto CU;
```

Strings. Several methods return a string to the caller. The Shape object's **Name** method (**getName** of **CVisioShape** or **get_Name** of **IVShape**) is an example. All strings passed to or returned by the Visio methods are of type BSTR, which consists of 16-bit (wide) characters in Win32 programs. Visio allocates the memory for the strings it returns, and the caller is responsible for freeing the memory.

The wrapper classes defined in VISIWRAP.H take care of freeing memory for strings. If you do not use the wrapper classes, however, make sure that you call **SysFreeString** to free any string returned by Visio.

Arguments passed to Visio methods

Passing arguments to Visio methods is straightforward. Integer arguments are declared as short or long, depending on whether they are 2-byte or 4-byte values. Floating-point arguments are declared as double. Boolean values are passed as short integers. Arguments that are object pointers, BSTRs, or VARIANTs merit further discussion.

Object pointers. Some methods take object pointers, and some require a pointer to a specific type of Visio object. The Cell object's **GlueTo** method, for example, takes an argument that must refer to another Cell object.

Other methods that take object pointers are more lenient. For example, the Page object's **Drop** method takes a reference to the object to be dropped, because you might want to drop a master on a page, or you might want to drop a shape on a page.

The simplest way to pass an object pointer to a method is to pass a reference to an object of the appropriate wrapper class. For example, you would pass a reference to a **CVisioCell** as an argument to the **GlueTo** method.

The interfaces defined in VISIO.H declare object pointers as the corresponding interfaces. For example, VISIO.H declares **GlueTo** as taking a pointer to an **IVCell**. Because the **Drop** method is not restricted to a particular object, VISIO.H declares **Drop** to take an **IUnknown**, the OLE way to say that **Drop** takes a reference to any object. Internally, the **Drop** method determines what to drop by querying the object passed to it for an **IDataObject** interface. The interface you pass to **Drop** does not necessarily have to be an interface on a Visio object.

Strings. Any string passed to Visio must be a BSTR. The helper class **VBstr**, defined in HELPERS.H, is a convenient way to pass strings to Visio. **VBstr** takes care of allocating memory for the string when it is created, and frees the memory when the **VBstr** is destroyed. If you don't use **VBstr**, make sure that you call **SysFreeString** to free the memory you have allocated for strings.

For example, the following statement uses a **VBstr** to pass a cell name to the **Cells** method of a **CVisioShape** object. In this statement, *cell* is a variable of type **CVisioCell**:

```
hr = shape.Cells(VBstr("Connections.X4"), cell);
```

VARIANTs. Some Visio methods take arguments that aren't constrained to a single type. For example, if you pass an integer *i* to the **Item** method of a Documents collection, it returns a reference to the *i*th document in the collection. If you pass a string that is a document name to the same method, however, it returns a reference to a document of that name (assuming that the document exists).

OLE defines a data structure known as a **VARIANT** for passing such arguments. The helper class **VVariant**, defined in HELPERS.H, is a convenient way of passing **VARIANTs** to Visio. For example, compare the following two statements:

```
hr = pages.Item(VVariant(1L), page);
hr = masters.Item(VVariant("Position"), master);
```

The first statement passes 1 (an integer) to the **Item** method of a Pages collection. The second statement passes "Position" (a string) to the **Item** method of a Masters collection. In these statements, *page* and *master* are variables of type **CVisioPage** and **CVisioMaster**, respectively.

Handling Visio events in C++ programs

One way to handle Visio events from a C++ program is to use Event objects. An Event object pairs an event with an action—either to run an add-on or to notify another object, called a *sink object,* that the event has occurred. For a discussion of how Event objects work and details about implementing them in Visual Basic programs, see "Handling events with Event objects" in Chapter 15, "Handling events in Visio."

Implementing a sink object

You implement handling of Visio events in a C++ program in much the same way as in a Visual Basic program, with these exceptions:

- The sink object in your C++ program must be a COM object that exposes the **IDispatch** interface.
- The **IDispatch** interface must supply a method called **VisEventProc** that has the following signature:

```
STDMETHOD(VisEventProc) (
     WORD wEvent,              //Event code of the event that is firing
     IUnknown FAR*  ipSource,  //Pointer to IUnknown interface on the object firing the event
     DWORD  dwEventID,         //The ID of the event that is firing
     DWORD  dwSeq,             //The sequence number of the event
     IUnknown FAR* ipSubject,//Pointer to IUnknown interface on the subject of the event
     VARIANTvExtraInfo         //Additional information (usually none)
);
```

When you call **AddAdvise** to create the Event object, you pass a pointer to the **IUnknown** or **IDispatch** interface on the sink object.

Using CVisioAddonSink

Instead of implementing your own sink object, you can use the **CVisioAddonSink** class provided with Visio 5.0. This class is declared in the file ADDSINK.H in \DVS\LIBRARIES\C-CPP\VAO_INC.

To use CVisioAddonSink:

1. Include ADDSINK.H in your project source files. If you're using the wrapper classes defined in VISIWRAP.H, skip this step.
2. Write a callback function to receive the event notifications sent to the sink object.
3. Call **CoCreateAddonSink** with a pointer to your callback function and the address of an IUnknown. **CoCreateAddonSink** creates an instance of a sink object that knows about your callback function, and writes a pointer to an **IUnknown** interface on the sink object to the address you supplied.
4. Get a reference to the EventList collection of the Visio object from which you want to receive notifications.
5. Call the **AddAdvise** method of the EventList collection obtained in step 4 with the **IUnknown** interface obtained in step 3 and the event code of the Visio event you're interested in. When the event occurs, Visio will call your callback function.
6. When you're finished using the sink object, release it.

The sample program GENERIC.CPP uses **CVisioAddonSink** to handle two events: **DocumentCreated** and **ShapeAdded**. The program declares a callback function for each event. The signature of the callback function must conform to **VISEVENTPROC**, which is defined in ADDSINK.H. The following example shows one of the declarations. For the implementation of this function, see GENERIC.CPP.

```
HRESULT STDMETHODCALLTYPE ReceiveNotifyFromVisio (
   IUnknown FAR*     ipSink,
   WORD              wEvent,
   IUnknown FAR*     ipSource,
   DWORD             nEventID,
   DWORD             dwEventSeq,
   IUnknown FAR*     ipSubject,
   VARIANT           eventExtra);
```

To create the sink object, the program gets the EventList collection of the Application object (*app*), calls **CoCreateAddonSink** to create the sink object, and calls **AddAdvise** on the EventList object to create the Event object in Visio. The program sets a flag, *bFirstTime*, to ensure that the Event objects are created only once while the program is running. The ID of the Event object is stored in the static variable *stc_nEventID* for later reference. The **AddAdvise** call creates a second reference on the sink object, so the program can release *pSink*.

```
static long         stc_nEventID = visEvtIDInval;
IUnknown FAR*       pSink = NULL;
IUnknown FAR*       pAnotherSink = NULL;
static BOOL         bFirstTime = TRUE;
CVisioApplication   app;
CVisioEventList     eList;
CVisioEvent         event;
...
if (bFirstTime && (SUCCEEDED(app.EventList(eList))))
{
  bFirstTime= FALSE;

  if (SUCCEEDED(CoCreateAddonSink(ReceiveNotifyFromVisio, &pSink)))
  {
    if (SUCCEEDED(eList.AddAdvise(visEvtCodeDocCreate,
                                  VVariant(pSink), VBstr(""), VBstr(""), event)))
    {
      event.ID(&stc_nEventID);
    }
    // If AddAdvise succeeded, Visio now holds a reference to the sink object
    // via the event object, and pSink can be released.
    pSink->Release();
    pSink= NULL;
  }
...
}
```

Event objects created with **AddAdvise** persist until the Event object is deleted, all references to the source object are released, or the instance of Visio is closed. If your program needs to perform cleanup tasks before Visio is closed, handle the **BeforeQuit** event.

Visio libraries (VSLs)

A Visio library (VSL) is a special dynamic-link library (DLL) that is loaded by Visio at run time and that can implement one or more Visio add-ons (programs that use Automation to control Visio).

An add-on implemented by a Visio library can interact with Visio objects in exactly the same fashion as an add-on implemented by an executable (.EXE) file or code in a document's VBA project, and a user can do exactly the same things. Add-ons implemented in a VSL have performance and integration advantages over those implemented in executable programs—for example, because a VSL runs in the same process as Visio. However, you cannot run a Visio library from Windows Explorer as you can an executable program.

Visio recognizes as a Visio library any file with a .VSL extension in the Add-ons or Startup paths. Installing a VSL is simply a matter of copying the file to one of the directories specified in the Visio Add-ons or Startup paths. The next time you run Visio, the add-ons implemented by that VSL are available to Visio.

Advantages of Visio libraries

All else being equal, a Visio library runs faster than an executable program. Because a Visio library is a DLL, it is loaded into the process space of the Visio instance that is using the library. Calls from a Visio library to Visio do not cross a process boundary, as is the case when an executable program calls Visio.

In addition, because a Visio library runs in the same process as Visio, it is much easier for it to open a dialog box that is modal with respect to the Visio process. When two executable files (an add-on and Visio) are running, it is difficult for one of them to display a dialog box that is modal with respect to the other. An add-on executable program can display a dialog box, but the user can click the Visio window and change the Visio state while the dialog box is open.

Files for developing VSLs

The files you'll need to develop Visio libraries are in \DVS\LIBRARIES\C-CPP. This folder also contains source and .MAK files for a simple but functional Visio library. See README.TXT in the C-CPP folder for details.

In addition, the file MYADDON.CPP in \DVS\LIBRARIES\C-CPP\SAMPLES\MYADDON contains a shell for writing your own VSL.

The architecture of a Visio library

A VSL is nothing more than a standard dynamic-link library that exports an entry point with the prescribed name **VisioLibMain**.

Visio loads a VSL using **LoadLibrary** and frees it using **FreeLibrary**. Unless your VSL is installed in a Visio Startup folder, your VSL should not make assumptions about when it will get loaded. Visio loads non-startup VSLs only when it needs to do so. If an instance of Visio does load a VSL, it does not call **FreeLibrary** on the VSL until the instance shuts down.

The file VDLLMAIN.C provides a default implementation for **DllMain**, which is the standard DLL entry point that Windows calls when it loads and unloads a DLL. The file VAO.C implements several other functions that you may find useful; some of these are mentioned in the paragraphs that follow.

Once Visio has loaded a VSL, it makes occasional calls to the VSL's **VisioLibMain** procedure. One of the arguments Visio passes to **VisioLibMain** is a message code that tells the VSL why it is being called. All messages that Visio sends are defined in VAO.H.

The prescribed prototype for **VisioLibMain** can be found in VAO.H:

```
typedef WORD VAORC, FAR* LPVAORC;          // Visio add-on return code.
typedef WORD VAOMSG, FAR* LPVAOMSG;        // Visio add-on message code.

#if defined(_WIN32)                        // Visio add-on call back proc.
  #define VAOCB __cdecl
#else
  #define VAOCB LOADDS PASCAL FAR
#endif

// The prototype of VisioLibMain should conform to VAOFUNC.
typedef VAORC (VAOCB VAOFUNC) (VAOMSG,WORD,LPVOID);
```

A typical **VisioLibMain** will thus look something like:

```
// Make sure your DLL exports VisioLibMain.
#include "vao.h"
VAORC VAOCB VisioLibMain (VAOMSG wMsg, WORD wParam, LPVOID lpParam)
  {
  VAORC result = VAORC_SUCCESS;
  switch (wMsg)
    {
    case V2LMSG_ENUMADDONS:
      // Code to register this VSL's add-ons goes here.
      break;
    case V2LMSG_RUN:
      // Code to run add-on with ordinal wParam goes here.
      break;
    default:
      // Trigger generic response to wMsg.
      // VAOUtil_DefVisMainProc and VLIBUTL_hModule
      // are helper procedures implemented in vao.c.

      result = VAOUtil_DefVisMainProc(wMsg, wParam, lpParam, VLIBUTL_hModule());
      break;
    };
  return result;
  }
```

This **VisioLibMain** specifically handles the V2LMSG_RUN and V2LMSG_ENUMADDONS messages. Other messages are deferred to **VAOUtil_DefVisMainProc**, a function that implements generic message responses. **VLIBUTL_hModule** evaluates to the module handle of the VSL.

Declaring and registering add-ons

When Visio sends the V2LMSG_ENUMADDONS message to a VSL's **VisioLibMain**, it is asking for descriptions of the add-ons implemented by the VSL.

The file LIB.C implements a sample VSL. In it you can see source code demonstrating how a VSL registers add-ons. Two aspects are involved: First, LIB.C defines a data structure describing its add-ons. Second, in response to V2LMSG_ENUMADDONS, it passes this data structure to Visio.

LIB.C implements one add-on. Near the top of the file is the following code:

```
#define DEMO_ADDON_ORDINAL 1
PRIVATE VAOREGSTRUCT stc_myAddons[] =
{
   {
   DEMO_ADDON_ORDINAL,            // Ordinal of this add-on
   VAO_AOATTS_ISACTION,           // This add-on does things to Visio.
   VAO_ENABLEALWAYS,              // This add-on is always enabled.
   0,                             // Invoke on mask.
   0,                             // Reserved for future use.
   "VSL Automation Demo",         // The name of this add-on.
   },
};
```

VAOREGSTRUCT is declared in VAO.H. You'll find comments and declarations there that give more information on the various fields in the structure.

When Visio tells a VSL to run an add-on, it identifies which add-on by specifying the add-on's ordinal, a unique value that identifies the add-on within the file. The **stc_myAddons** array declares one add-on whose ordinal is 1 (DEMO_ADDON_ORDINAL). If LIB.C implemented two add-ons instead of one, **stc_myAddons** would have two entries instead of one and each entry would designate a unique ordinal.

The declared add-on is presented in the Visio user interface as "VSL Automation Demo." If you intend to localize your add-on, you wouldn't declare its name in the code as is shown here. Rather, you'd read the name from a string resource and dynamically initialize the **VAOREGSTRUCT**.

VAO_ENABLEALWAYS tells Visio this add-on should be considered enabled at all times. Other enabling policies can be declared. There are many add-ons, for example, that it makes sense to run only when a document is open. Such add-ons can declare an enabling policy of VAO_NEEDSDOC. Visio makes such add-ons unavailable when no documents are open. When such an add-on is run, it can assert that a document is open. Several static enabling policies similar to VAO_NEEDSDOC are declared in VAO.H.

VAO.H also contains a policy called VAO_ENABLEDYNAMIC. When Visio wants to determine whether the add-on is enabled, it sends V2LMSG_ISAOENABLED to a dynamically enabled add-on. The add-on can claim to be enabled or disabled based on its own criteria.

The last aspect of **VAOREGSTRUCT** involves making an add-on run automatically when an instance of Visio launches. To make an add-on implemented by an executable program run on startup, you simply place the executable file in one of the directories specified by the Visio Startup paths setting.

For add-ons implemented in a VSL, those to be run on startup must also specify VAO_INVOKE_LAUNCH in the invokeOnMask field of their **VAOREGSTRUCT**. This constant allows a single .VSL file to implement some add-ons that run automatically when Visio launches, and some that don't.

By itself, **VAOREGSTRUCT** is just a data structure whose mere existence doesn't tell Visio anything. When Visio sends V2LMSG_ENUMADDONS to a VSL, the library should respond by passing Visio a pointer to the array of **VAOREGSTRUCTs** discussed earlier, so the data they contain is available to Visio. To do this, LIB.C makes use of a utility implemented in VAO.C. The code is as follows:

```
result = VAOUtil_RegisterAddons(
         ((LPVAOV2LSTRUCT)lpParam)->wSessID,
         stc_myAddons,
         sizeof(stc_myAddons)/sizeof(VAOREGSTRUCT));
```

For details about what this code does, look at the source in VAO.C.

Running an add-on

Visio sends V2LMSG_RUN to a VSL when the VSL is to run one of its add-ons. The ordinal of the add-on to run is passed in *wParam*.

Visio sends V2LMSG_RUN only if it has determined that the designated add-on is enabled, according to the enabling policy declared in the add-on's registration structure. If the add-on's enabling policy is VAO_ENABLEDYNAMIC, the VSL will already have responded with VAORC_L2V_ENABLED to the V2LMSG_ISAOENABLED message it received from Visio.

In addition to the ordinal of the add-on to run, Visio passes a pointer to a **VAOV2LSTRUCT** with the V2LMSG_RUN message. From VAO.H:

```
VAO_EMBEDDABLE_STRUCT
  {
  HINSTANCE hVisInst;                  // Handle of running Visio instance.
  LPVAOFUNC lpfunc;                    // Callback address in Visio.
  WORD   wSessID;                      // ID of session.
  LPVOID lpArgs;                       // Reserved for future use.
  LPSTR    lpCmdLineArgs;              // Command line arguments.
  } VAOV2LSTRUCT, FAR* LPVAOV2LSTRUCT;
```

This structure gives the instance handle of the Visio instance sending the message, which is sometimes useful. (*lpfunc* and *lpArgs* are used by other functions in VAO.C). In *lpCmdLineArgs*, Visio passes an argument string to the add-on. This is the same string Visio would pass to an analogous add-on implemented as an executable program.

You'll sometimes be interested in *wSessID*, which is the ID Visio has assigned to the "session" it associated with the V2LMSG_RUN it just sent. For example, you might use *wSessID* if your add-on initiates a modeless activity.

Most add-ons will perform a modal action in response to V2LMSG_RUN: They receive the message, do something, then return control to Visio. Unless the add-on says otherwise, Visio considers the session finished when it regains control from the add-on.

Pseudocode for this typical case would be:

```
case V2LMSG_RUN:
  wParam is ordinal of add-on to run.

  Execute code to do whatever it is the add-on with ordinal wParam does.
    This will probably involve instantiating Visio objects and invoking methods and
    properties of those objects. You can use the C++ support services discussed in the
    previous section just as if this code were in an .EXE file.

  if (operation was successful)
    return VAORC_SUCCESS;
  else
    return VAORC_XXX;              // See vao.h.
```

Sometimes, in response to V2LMSG_RUN, an add-on may initiate an activity that doesn't terminate when the add-on returns control to Visio. Such activities are called *modeless*. An add-on may, for example, open a window that will stay open indefinitely.

If your add-on implements a modeless activity, it should remember the session ID passed with V2LMSG_RUN. Pseudocode for such an add-on would be:

```
case V2LMSG_RUN:
  wParam is ordinal of add-on to run.

  Execute code to initiate modeless activity.
    For example, open a window and stash its handle.

  if (operation was successful)
    {
    stash lParam->wSessID where it can be looked up later.
    return VAORC_L2V_MODELESS;
    }
  else
    return VAORC_XXX;            // See vao.h.
```

Note the return value of VAORC_L2V_MODELESS. This tells Visio the session still persists, even though the VSL has completed handling the V2LMSG_RUN message.

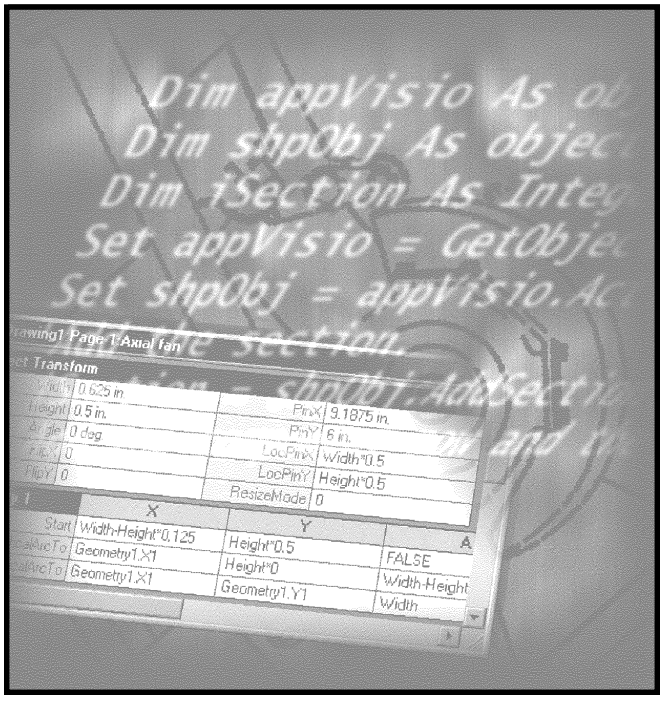
A modeless session initiated in this fashion persists until either the VSL ends the session or the Visio instance associated with the session terminates.

If the VSL ends the session (for example, perhaps the window it opened has been closed), it does so with this function call:

```
VAOUtil_SendEndSession(wSessID);// wSessID: ID of
                                // terminating session.
```

When Visio terminates, it sends V2LMSG_KILLSESSION to all extant sessions. With V2LMSG_KILLSESSION, Visio passes a **VAOV2LSTRUCT** whose *wSessID* field identifies the ID of the session to terminate. The VSL should respond by terminating and cleaning up after the identified session.

PART IV



Appendixes

Appendix A

Arcs and splines in Visio

This appendix provides technical details about circular and elliptical arcs in Visio. It also discusses how Visio represents splines, including how to create a spline by entering control points and spline knots in the ShapeSheet window.

Topics in this appendix

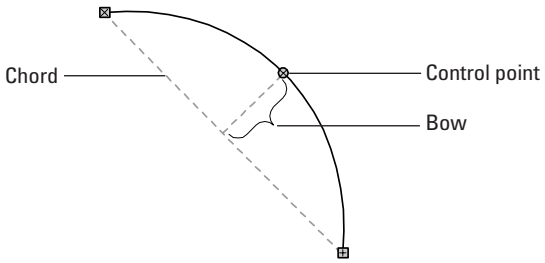
About arcs

A *circular arc* is a portion of a circle. An *elliptical arc* is a portion of an ellipse. An elliptical arc may appear to be circular, because a circle is simply a special case of an ellipse. The Geometry section shows the difference: A circular arc is always defined by an ArcTo row, and an elliptical arc is defined by an EllipticalArcTo row.

Circular arcs

In a circular arc, the magnitude of the bow is the distance from the midpoint of the chord to the midpoint of the arc, as the following figure shows. The bow's value is positive if the arc is drawn in the counterclockwise direction; otherwise, it is negative. A selected arc has a control point at the arc's midpoint, which is always located along the perpendicular bisector of the chord. If you try to move the control point with the pencil tool, the point moves freely, but it always snaps back to a position along the perpendicular bisector.

When you resize a circular arc, you change the radius of the circle of which the arc is a portion. The arc may flatten or bulge—appropriate resizing behavior for a circle, but perhaps not the expected behavior for your shapes. For example, resizing a full circle in one direction turns it into an ellipse. If you want an arc that resizes proportionately, use an elliptical arc.



A circular arc

To create a circular arc:

1. Select a shape, open its ShapeSheet window, then select a LineTo or EllipticalArcTo row in the Geometry section.
2. From the Edit menu, choose Change Row Type, check ArcTo, then click OK.

The following table shows what the cells of an ArcTo row represent.

Circular arc representation in the Geometry section

Row	Cell	Value
The row that precedes the ArcTo row*	X	The *x*-coordinate of the begin point
	Y	The *y*-coordinate of the begin point
ArcTo	X	The *x*-coordinate of the end point
	Y	The *y*-coordinate of the end point
	A	Size of the arc's bow

* The begin point of the arc is determined by the X and Y cells of the previous row in the Geometry section.

Elliptical arcs

The arcs you draw with the arc tool are always a quarter of an ellipse, and those drawn with the pencil tool are a portion of a circle, but both are represented in a Geometry section as elliptical arcs.

When stretched, an elliptical arc's eccentricity changes in proportion to the stretching to maintain a smooth curve. *Eccentricity* controls how asymmetrical the arc appears. In most cases, you'll probably want to use an elliptical arc in your shapes rather than a circular arc, whose resizing behavior is constrained by the fact that it must remain circular.

To create an elliptical arc, do one of the following:

- Draw an arc using the pencil or arc tool.
- In the ShapeSheet window, change the row type of a LineTo or an ArcTo row to an EllipticalArcTo row.
- On the drawing page, use the pencil tool to drag the control point of a straight line. This transforms the line into an elliptical arc.

An elliptical arc's geometry is described in an EllipticalArcTo row, as the following table shows.

Elliptical arc representation in the Geometry section

Row	Cell	Value
The row that precedes EllipticalArcTo*	X	The *x*-coordinate of the begin point
	Y	The *y*-coordinate of the begin point
EllipticalArcTo	X	The *x*-position of the end point
	Y	The *y*-position of the end point
	A	The *x*-position of the control point
	B	The *y*-position of the control point
	C	Angle of the arc
	D	Eccentricity of the arc

* The elliptical arc's begin point is determined by the X and Y cells of the previous row in the Geometry section.

You can move the control point of an elliptical arc to change the arc's eccentricity. An eccentricity of 1 represents a circular arc, and a value greater or less than 1 represents an arc with more or less eccentricity. For example, in an ellipse that is 2 inches wide and 1 inch tall, each elliptical arc has an eccentricity of 2. In an ellipse that is 1 inch wide and 2 inches tall, each elliptical arc has an eccentricity of 1/2.

To change an elliptical arc's eccentricity:

- Select the pencil tool, and then press Ctrl as you drag the control point to display the eccentricity handles, which you can stretch and rotate.

When you move an arc's eccentricity handles, Visio generates formulas in the C and D cells of the EllipticalArcTo row to record the current orientation and shape of the elliptical arc. If a shape with elliptical arcs is stretched, Visio changes the eccentricity and angle of the arcs if necessary so that the arcs resize consistently with the rest of the shape.

Useful arc formulas

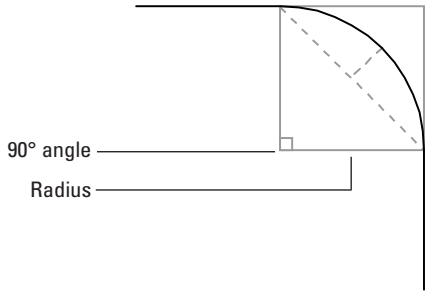
You can control the resizing behavior of circular arcs using ShapeSheet formulas that calculate the bow and radius.

To find the bow when the arc's radius and angle are known. If you know the radius of an arc and the angle that an ArcTo will subtend, you can calculate the bow with the following general equation:

```
|Bow| = radius * (1 - COS(angle/2))
```

If the bow is zero, the arc is a straight line. You can use this equation for any shape, open or closed, to create rounded corners that span a set angle, as shown in the following figure. The advantage of using circular arcs is that the corners resize smoothly. For example, if you know that the radius is 2 inches and the angle is 45 degrees, in an ArcTo row of the Geometry section, you would enter this formula:

```
Geometryn.An = 2 in. * (1 - COS(45 deg. /2))
```



Using a circular arc segment for a rounded corner

In a shape such as a rectangle where the value of *angle* won't change (it's always 90 degrees), you can reduce part of the formula to a constant. If *angle* is always 90 degrees, (1–COS(angle/2)) = 0.2929. So you can enter the formula as:

```
Geometryn.An = radius * 0.2929
```

Using this constant may speed up processing, but it limits flexibility if you later decide that the angle won't always be 90 degrees. For details about creating rounded corners, see "Creating curved shapes that resize smoothly" in Chapter 3, "Controlling shape size and position."

To find the radius when the bow is known. If you know the bow of an arc, you can calculate its radius. To do this, find the magnitude of the chord—the distance between the arc's begin point and end point. In the following formula, X1, Y1 represent the arc's begin point and X2, Y2 represent the arc's end point. The length of the chord, then, is:

```
Chord length = SQRT( (Y2 – Y1) ^2 + (X2 - X1)^2 )
```

And the radius is:

```
Radius = (4 * Bow² + Chord²) / 8Bow
```

Working with splines

Freeform curves are represented internally as B-splines. (Bézier curves are a special case of a spline.) To create a spline on the drawing page, use the freeform tool. To adjust the spline, drag the handles that appear when you select the spline with the freeform or pencil tool.

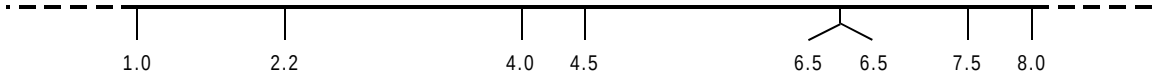
You don't need to know any of the technical details to draw and modify a spline with the mouse. However, if you want to create a spline in Visio based on a specific set of control points and knots, you may prefer to work in the ShapeSheet window. This section describes how to create a spline in the ShapeSheet window by drawing with the line tool, then converting that shape's Geometry rows to spline rows and adding degree and knot values to the appropriate cells.

To create a spline from a program, you work with the sections, rows, and cells in the spline, using the rows and cell formulas described below. For details about adding rows and changing row types, see "Modifying a shape's sections and rows" in Chapter 14, "Working with drawings and shapes." For constants and cell names to use when working with spline rows from a program, see Appendix B, "ShapeSheet sections, cell references, and index constants."

Splines: the basics

If you're unfamiliar with splines, you may want to consult a standard college textbook on curve and surface geometry or a comparable textbook on computer-aided design. Until you do, here's a brief introduction.

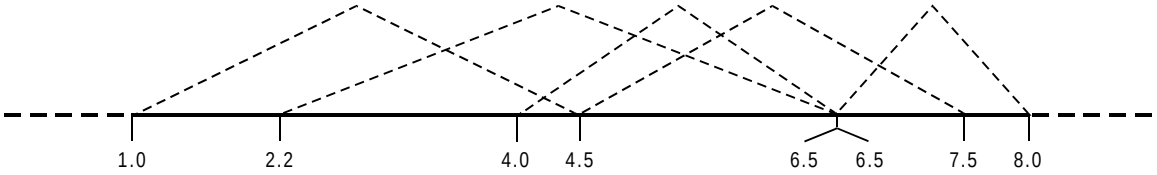
Think of a spline as a thin, flexible ruler whose marks may be unevenly spaced. The marks are called *knots*, and a knot may be repeated more than once. The *multiplicity* of a knot is the number of times it is repeated.



Knots on a spline

The spline's *degree* is a positive integer (between 1 and 9 in Visio) that is the degree of the polynomial equations used to calculate the pieces of the spline.

The *control points* of the spline influence its curvature. Each control point is located at some distance from the visible curve of the spline; taken together, all of a spline's control points are sometimes called the spline's *control polygon*. Each control point has a *domain of influence* of degree+2 consecutive knots. For example, if a spline has a degree of 2, each control point has a domain of influence of four consecutive knots on the spline. The control point influences the part of the spline between the first and last knot of its domain.

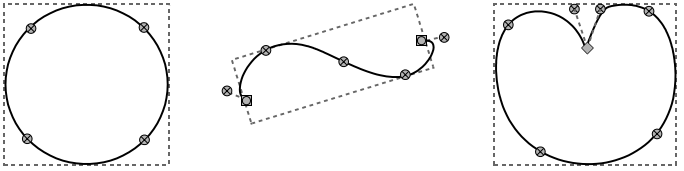


Control points and their domains of influence on a spline with a degree of 2

The visible curve of the spline is "attracted" to each control point, which exerts the strongest pull at about the middle of its domain of influence. A control point has no effect outside its domain of influence, but the domains of adjacent control points may overlap, so every point on the curve is affected by degree+2 control points. The degree of a spline, the number of knots it has, and the locations of its control points relative to each other all influence the appearance of the spline on the drawing page.

About periodic and nonperiodic splines

A spline can be *periodic* or *nonperiodic*. A nonperiodic spline has beginning and ending points, and it can be open or closed (that is, its beginning and ending points can meet). A periodic spline is a closed, seamless shape that has no beginning or ending point.



Periodic spline | Open nonperiodic spline | Closed nonperiodic spline

By mathematical convention, a nonperiodic spline has degree+1 knots of the same multiplicity at the end of the spline, as well as the beginning. In Visio, however, if a spline begins with degree+1 knots, Visio assumes that the spline is nonperiodic and that the multiple ending knots are implied.

For a periodic spline, the multiplicity of the first knot can be less than or equal to the degree of the spline. For example, if the spline has a degree of 3, the first one, two, or three knots can have the same value, but the fourth knot must be greater.

How spline data is organized in the ShapeSheet window

In the ShapeSheet window, Visio displays the definition of the spline in a Geometry section that contains a SplineStart row followed by one or more SplineKnot rows. The SplineStart row must be preceded by another kind of row, such as a Start row, to indicate the first control point of the spline. The preceding row can be a LineTo, ArcTo, or EllipticalArcTo row if the spline follows a segment of that type.

The following table describes the values in spline row cells. All coordinates are local. Spline knots are specified as described in "Splines: the basics" earlier in this appendix.

Spline representation in the Geometry section

Row	Cell	Value
The row that precedes SplineStart	X	The *x*-coordinate of the spline's first control point
	Y	The *y*-coordinate of the spline's first control point
SplineStart	X	The *x*-coordinate of the spline's second control point
	Y	The *y*-coordinate of the spline's second control point
	A	The position of the second knot on the spline
	B	The position of the first knot on the spline
	C	The position of the last knot on the spline
	D	The degree of the spline (an integer from 1 to 9)
SplineKnot	X	The *x*-coordinate of a control point
	Y	The *y*-coordinate of a control point
	A	The position of the third or greater knot on the spline

For a valid spline, the values of these cells must follow certain rules:

- The knot values must not decrease from the first knot to the last.
- The multiplicity of the first knot (the number of times the same knot value occurs) may not exceed the degree of the spline plus 1. For example, if the spline has a degree of 3, the first four knots can have the same value.
- The multiplicity of all knots after the first must not exceed the degree of the spline.
- If you're creating splines in Visio from a data set that has degree+1 ending knots, you need to create a SplineKnot row for only one ending knot, not degree+1 knots.
- The degree of the spline must be an integer from 1 to 9. A value of 1 draws the spline as straight segments between control points (which is essentially identical to its control polygon). The higher the degree, the flatter the spline's curves will be.
- Unless the degree of the spline is 1, the SplineStart row must be followed by at least one SplineKnot row.

If you delete a SplineKnot row of a nonperiodic spline so that it has fewer than degree+1 knots at the beginning, Visio assumes that you want a periodic spline and converts the spline.

If the spline definition becomes invalid, Visio draws the spline as if the last SplineKnot row is a LineTo row. All other SplineKnot rows are ignored.

Creating a spline in the ShapeSheet window: an example

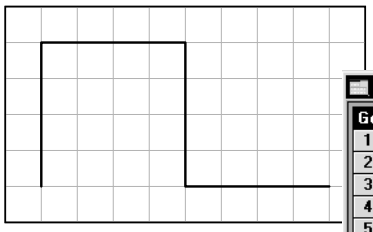
The following example shows how to create a spline in the ShapeSheet window. Suppose you want to create a spline based on the following definition:

Degree = 3
Control points = {3,3}, {3,5}, {5,5}, {5,3}, {7,3}
Knots = 0, 0, 0, 0, 1, 2

Using conventional mathematical notation, the knots of the same spline would be specified as 0, 0, 0, 0, 1, 2, 2, 2, 2.

To create the example spline in the ShapeSheet window:

1. Use the line tool to draw the spline's control polygon as a series of line segments connecting its control points. In this example, the control polygon and its Geometry section would look like this.



splines:Page-1:Sheet.1

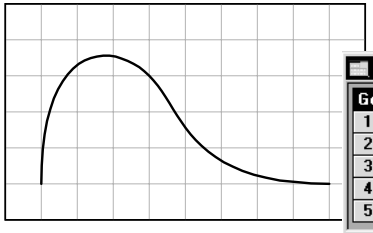
Geometry 1		X	Y	A	B	C	D
1	Start	0.0000 in.	0.0000 in.	1	0		
2	SplineStart	0.0000 in.	1.0000 in.	0.0000	0.0000	2.0000	3
3	SplineKnot	1.0000 in.	1.0000 in.	0.0000			
4	SplineKnot	1.0000 in.	0.0000 in.	0.0000			
5	SplineKnot	2.0000 in.	0.0000 in.	1.0000			

A spline's control polygon drawn as a Visio shape

2. In the ShapeSheet window, change the row type of the first LineTo row to SplineStart.
3. Change the row type of all subsequent LineTo rows to SplineKnot.
4. Set formulas as shown in the following table.

Row	Cell	Formula	Meaning
SplineStart	Geometry1.A2	=0	Second knot
	Geometry1.B2	=0	First knot
	Geometry1.C2	=2	Last knot
	Geometry1.D2	=3	Degree of the spline
SplineKnot	Geometry1.A3	=0	Third knot
SplineKnot	Geometry1.A4	=0	Fourth knot
SplineKnot	Geometry1.A5	=1	Fifth knot

The resulting spline and its Geometry section look like this.



splines:Page-1:Sheet.1

Geometry 1		X	Y	A	B	C	D
1	Start	0.0000 in.	0.0000 in.	1	0		
2	SplineStart	0.0000 in.	1.0000 in.	0.0000	0.0000	2.0000	3
3	SplineKnot	1.0000 in.	1.0000 in.	0.0000			
4	SplineKnot	1.0000 in.	0.0000 in.	0.0000			
5	SplineKnot	2.0000 in.	0.0000 in.	1.0000			

The control polygon converted to a spline

Appendix B

ShapeSheet sections, cell references, and index constants

This appendix lists cell references for cells that appear in the ShapeSheet window for shapes, pages, and masters. Cell references are grouped alphabetically by section name. This appendix also lists the corresponding index constants that you can use in a program to access sections, rows, and cells with Automation, plus index constants you can use to access tab settings.

For details about using cell references in formulas, search online help for "formulas: cell references." For details about accessing cells from a program, see "Getting cells from shapes" in Chapter 13, "Getting information from Visio drawings," and "Working with formulas" in Chapter 14, "Working with drawings and shapes."

Using contants in VISCONST.BAS

Many of the constants in VISCONST.BAS represent arguments for various methods. Constants that pertain to a particular method are listed with the method in the Automation Reference in online help. Certain other constants in VISCONST.BAS are reserved for future use or for internal use by Visio, and their values may change in future versions of Visio. Unless a constant is listed in this appendix or in the Automation Reference, you should not use it in your programs.

Topics in this appendix

Sections, cells, and indexes for shapes

This table lists sections and cells that are displayed in the ShapeSheet window, with constants for the corresponding section, row, and cell indexes.

In sections that have a variable number of rows, such as **visSectionAction**, rows are indexed using the row constant as a base. To refer to a particular row, add an integer offset to the row index constant, starting with 0 for the first row.

TIP To find the value of a constant, use the Object Browser in the Visual Basic Editor. When a constant is selected in the Members Of list, the details pane displays the value of the constant.

Section	Cell reference	Section index	Row index	Cell index
1-D Endpoints[1]	BeginX BeginY EndX EndY	visSectionObject	visRowXForm1D	vis1DBeginX vis1DBeginY vis1DEndX vis1DEndY
Actions	Actions.Action[*n*] Actions.Menu[*n*] Actions.Prompt[*n*] Actions.C*n* Actions.D*n*	visSectionAction	visRowAction + *n*	visActionAction visActionMenu visActionPrompt visActionChecked visActionDisabled
Alignment[2]	AlignLeft AlignCenter AlignRight AlignTop AlignMiddle AlignBottom	visSectionObject	visRowAlign	visAlignLeft visAlignCenter visAlignRight visAlignTop visAlignMiddle visAlignBottom
Character Format	Char.Font[*n*] Char.Color[*n*] Char.Style[*n*] Char.Case[*n*] Char.Pos[*n*] Char.Size[*n*]	visSectionCharacter	visRowCharacter + *n*	visCharacterFont visCharacterColor visCharacterStyle visCharacterCase visCharacterPos visCharacterSize
Connection Points	Connections.X*n* Connections.Y*n*	visSectionExport	visRowExport + *n*	visX visY

[1] Valid only for 1-D shapes.
[2] Valid only for shapes that are glued to one or more shapes or guides.

Section	Cell reference	Section index	Row index	Cell index
Controls	Controls.X*n*	visSectionControls	visRowControl + *n*	visCtlX
	Controls.Y*n*			visCtlY
	Controls.XDyn[*n*]			visCtlXDyn
	Controls.YDyn[*n*]			visCtlYDyn
	Controls.XCon[*n*]			visCtlXCon
	Controls.YCon[*n*]			visCtlYCon
	Controls.CanGlue[*n*]			visCtlGlue
	Controls.Prompt[*n*]			visCtlTip
Custom Properties	Prop.*Name*.Label	visSectionProp	visRowProp + *n*	visCustPropsLabel
	Prop.*Name*.Prompt			visCustPropsPrompt
	Prop.*Name*.SortKey			visCustPropsSortKey
	Prop.*Name*.Type			visCustPropsType
	Prop.*Name*.Format			visCustPropsFormat
	Prop.*Name*.Value[3]			visCustPropsValue
	Prop.*Name*.Invisible			visCustPropsInvis
	Prop.*Name*.Verify			visCustPropsAsk
Events	TheData	visSectionObject	visRowEvent	visEvtCellTheData
	TheText			visEvtCellTheText
	EventDblClick			visEvtCellDblClick
	EventXFMod			visEvtCellXFMod
	EventDrop			visEvtCellDrop
Fill Format	FillBkgnd	visSectionObject	visRowFill	visFillBkgnd
	FillPattern			visFillPattern
	FillForegnd			visFillForegnd
	ShdwBkgnd			visFillShdwBkgnd
	ShdwPattern			visFillShdwPattern
	ShdwForegnd			visFillShdwForegnd

[3] The Value cell is the default and can be omitted from the cell reference.

Section	Cell reference	Section index	Row index	Cell index
Geometry*n*				
(Start row)[4]	Geometry*n*.NoFill Geometry*n*.NoShow	visSectionFirstComponent + *n*	visRowComponent	visCompNoFill visCompNoShow
	Geometry*n*.X1 Geometry*n*.Y1		visRowVertex + 0	visX visY
(LineTo row)	Geometry*n*.X2...*n* Geometry*n*.Y2...*n*		visRowVertex + *n*	visX visY
(ArcTo row)	Geometry*n*.X2...*n* Geometry*n*.Y2...*n* Geometry*n*.A2...*n*		visRowVertex + *n*	visX visY visBow
(EllipticalArcTo row)	Geometry*n*.X2...*n* Geometry*n*.Y2...*n* Geometry*n*.A2...*n* Geometry*n*.B2...*n* Geometry*n*.C2...*n*		visRowVertex + *n*	visX visY visControlX visControlY
visEccentricityAngle	Geometry*n*.D2...*n*			visAspectRatio
(SplineStart row)	Geometry*n*.X2...*n* Geometry*n*.Y2...*n* Geometry*n*.A2...*n* Geometry*n*.B2...*n* Geometry*n*.C2...*n* Geometry*n*.D2...*n*		visRowVertex + *n*	visX visY visSplineKnot visSplineKnot2 visSplineKnot3 visSplineDegree
(SplineKnot row)	Geometry*n*.X2...*n* Geometry*n*.Y2...*n* Geometry*n*.A2...*n*		visRowVertex + *n*	visX visY visSplineKnot
Guide Info[5]	PinX PinY Angle	visSectionObject	visRowGuide	visX visY visGuideFlags
Image Info[6]	ImgOffsetX ImgOffsetY ImgWidth ImgHeight	visSectionObject	visRowForeign	visFrgnImgOffsetX visFrgnImgOffsetY visFrgnImgWidth visFrgnImgHeight

[4] Although the Start row of a Geometry section appears as one row in the ShapeSheet window, it is represented internally by two row indexes: **visRowComponent** and **visRowVertex + 0**. Rows that follow the Start row are **visRowVertex + 1**, **visRowVertex + 2**, and so forth.

[5] Valid only for guides and guide points.

[6] Valid only for linked or embedded objects.

Section	Cell reference	Section index	Row index	Cell index
HyperLink	Description	visSectionObject	visRowHyperlink	visHLinkDescription
	ExtraInfo			visHLinkExtraInfo
	Frame			visHLinkFrame
	Address			visHLinkAddress
	NewWindow			visHLinkNewWin
	SubAddress			visHLinkSubAddress
Layer Membership	LayerMember	visSectionObject	visRowLayerMem	visLayerMember
Line Format	LineWeight	visSectionObject	visRowLine	visLineWeight
	LineColor			visLineColor
	LinePattern			visLinePattern
	Rounding			visLineRounding
	BeginArrow			visLineBeginArrow
	EndArrow			visLineEndArrow
	ArrowSize			visLineArrowSize
	EndCap			visLineEndCap
Miscellaneous	NoObjHandles	visSectionObject	visRowMisc	visNoObjHandles
	NonPrinting			visNonPrinting
	NoCtlHandles			visNoCtlHandles
	NoAlignBox			visNoAlignBox
	UpdateAlignBox			visUpdateAlignBox
	HideText			visHideText
	ObjType			visLOFlags
	ObjInteract			visLOInteraction
	ObjBehavior			visLOBehavior
Paragraph Format	Para.IndFirst[*n*]	visSectionParagraph	visRowParagraph + *n*	visIndentFirst
	Para.IndLeft[*n*]			visIndentLeft
	Para.IndRight[*n*]			visIndentRight
	Para.SpLine[*n*]			visSpaceLine
	Para.SpBefore[*n*]			visSpaceBefore
	Para.SpAfter[*n*]			visSpaceAfter
	Para.HorzAlign[*n*]			visHorzAlign

Section	Cell reference	Section index	Row index	Cell index
Protection	LockWidth	visSectionObject	visRowLock	visLockWidth
	LockHeight			visLockHeight
	LockMoveX			visLockMoveX
	LockMoveY			visLockMoveY
	LockAspect			visLockAspect
	LockDelete			visLockDelete
	LockBegin			visLockBegin
	LockEnd			visLockEnd
	LockRotate			visLockRotate
	LockCrop			visLockCrop
	LockVtxEdit			visLockVtxEdit
	LockTextEdit			visLockTextEdit
	LockFormat			visLockFormat
	LockGroup			visLockGroup
	LockCalcWH			visLockCalcWH
	LockSelect			visLockSelect
Scratch	Scratch.X*n*	visSectionScratch	visRowScratch + *n*	visScratchX
	Scratch.Y*n*			visScratchY
	Scratch.A*n*			visScratchA
	Scratch.B*n*			visScratchB
	Scratch.C*n*			visScratchC
	Scratch.D*n*			visScratchD
Shape Transform[7]	PinX	visSectionObject	visRowXFormOut	visXFormPinX
	PinY			visXFormPinY
	Width			visXFormWidth
	Height			visXFormHeight
	LocPinX			visXFormLocPinX
	LocPinY			visXFormLocPinY
	Angle			visXFormAngle
	FlipX			visXFormFlipX
	FlipY			visXFormFlipY
	ResizeMode			visXFormResizeMode
Text Block Format	VerticalAlign	visSectionObject	visRowText	visTxtBlkVerticalAlign
	TopMargin			visTxtBlkTopMargin
	BottomMargin			visTxtBlkBottomMargin
	LeftMargin			visTxtBlkLeftMargin
	RightMargin			visTxtBlkRightMargin
	TextBkgnd			visTxtBlkBkgnd

[7] In Visio 5.0, a guide can have a Shape Transform section. If so, only the PinX, PinY, and Angle cells in that section are valid. The formulas in these cells override those in PinX and PinY in the Guide Info row. However, if the document is saved in pre-5.0 format, the guide reverts to the position defined by PinX and PinY in the Guide Info row.

Section	Cell reference	Section index	Row index	Cell index
Text Fields[8]	Fields.Value[*n*]	visSectionTextField	visRowField + *n*	visFieldCell
Text Transform	TxtPinX TxtPinY TxtWidth TxtHeight TxtLocPinX TxtLocPinY TxtAngle	visSectionObject	visRowTextXForm	visXFormPinX visXFormPinY visXFormWidth visXFormHeight visXFormLocPinX visXFormLocPinY visXFormAngle
User-Defined Cells	User.*Name*.Value[9] User.*Name*.Prompt	visSectionUser	visRowUser + *n*	visUserValue visUserPrompt

[8] Each Text Fields row has one cell that contains the custom formula of the corresponding text field.
[9] Value is the default cell for this row and can be omitted from the cell reference.

Sections, cells, and indexes for pages

This table lists sections and cells that are displayed in the ShapeSheet window for a drawing page or a master editing page, with constants for the corresponding section, row, and cell indexes.

Section	Cell reference	Section index	Row index	Cell index
Actions	Actions.Action[*n*] Actions.Menu[*n*] Actions.Prompt[*n*] Actions.C*n* Actions.D*n*	visSectionAction	visRowAction + *n*	visActionAction visActionMenu visActionPrompt visActionChecked visActionDisabled
Custom Properties	Prop.*Name*.Label Prop.*Name*.Prompt Prop.*Name*.SortKey Prop.*Name*.Type Prop.*Name*.Format Prop.*Name*.Value[1] Prop.*Name*.Invisible Prop.*Name*.Verify	visSectionProp	visRowProp + *n*	visCustPropsLabel visCustPropsPrompt visCustPropsSortKey visCustPropsType visCustPropsFormat visCustPropsValue visCustPropsInvis visCustPropsAsk

[1] Value is the default cell for this row and can be omitted from the cell reference.

Section	Cell reference	Section index	Row index	Cell index
HyperLink	Description	visSectionObject	visRowHyperlink	visHLinkDescription
	ExtraInfo			visHLinkExtraInfo
	Frame			visHLinkFrame
	Address			visHLinkAddress
	NewWindow			visHLinkNewWin
	SubAddress			visHLinkSubAddress
Layers	Layers.Name[*n*]	visSectionLayer	visRowLayer + *n*	visLayerName
	Layers.Visible[*n*]			visLayerVisible
	Layers.Print[*n*]			visLayerPrint
	Layers.Active[*n*]			visLayerActive
	Layers.Locked[*n*]			visLayerLock
	Layers.Snap[*n*]			visLayerSnap
	Layers.Glue[*n*]			visLayerGlue
	Layers.Color[*n*]			visLayerColor
	Layers.Status[*n*]			visLayerStatus
Page Properties	PageWidth	visSectionObject	visRowPage	visPageWidth
	PageHeight			visPageHeight
	PageScale			visPageScale
	DrawingScale			visPageDrawingScale
	ShdwOffsetX			visPageShdwOffsetX
	ShdwOffsetY			visPageShdwOffsetY
	DrawingSizeType			visPageDrawSizeType
	DrawingScaleType			visPageDrawScaleType
Ruler & Grid	XRulerOrigin	visSectionObject	visRowRulerGrid	visXRulerOrigin
	YRulerOrigin			visYRulerOrigin
	XRulerDensity			visXRulerDensity
	YRulerDensity			visYRulerDensity
	XGridOrigin			visXGridOrigin
	YGridOrigin			visYGridOrigin
	XGridDensity			visXGridDensity
	YGridDensity			visYGridDensity
	XGridSpacing			visXGridSpacing
	YGridSpacing			visYGridSpacing
Shape Transform[2]	Angle	visSectionObject	visRowXFormOut	visXFormAngle
User-Defined Cells	User.*Name*.Value[3]	visSectionUser	visRowUser + *n*	visUserValue
	User.*Name*.Prompt			visUserPrompt

[2] In Visio 5.0, a page can have a Shape Transform section. If it does, only the Angle cell is valid, although the section contains all of the Shape Transform cells shown in "Sections, Cells, and Indexes for Shapes."

[3] Value is the default cell for this row and can be omitted from the cell reference.

Tab cells and row types

The tab settings for a shape's text are accessible from a program only by section, row, and cell index. In Visio, tab settings can be displayed and changed on the Tabs tab (choose Text from the Format menu, then click Tabs).

Section index	Row index	Cell index
visSectionTab	visRowTab + *n*	0 ... 30

The section contains a row for each set of tabs defined for the shape. Each row contains three cells for each tab defined in that row, up to ten tabs. Cells for the entire row are indexed starting with 0.

Index	Determines
0	Number of active tabs in the row
1	Position of the first tab
2	Alignment code for the first tab
3	Reserved
4	Position of the second tab
5	Alignment code for the second tab
6	Reserved
…	…
28	Position of the tenth tab
29	Alignment code for the tenth tab
30	Reserved

The number of tabs that can be set depends on the tab row type. The row type can be changed by setting the **RowType** property of a tab section row in a shape with one of the following row tag constants:

- **visTagTab0**. Zero tabs. Text defaults to tabs every 0.5 inches.
- **visTagTab2**. Zero, one, or two tabs.
- **visTagTab10**. Zero to 10 tabs.

General-purpose index constants

You may find the following constants useful when iterating through all of the sections and rows for a shape, or checking for errors when retrieving sections and rows.

Logical position constants

The following constants allow you to access sections and rows by logical position. Use logical position constants when you want to iterate through all of the sections of a shape or rows of a section, but the order of traversal is not important.

Constant	Represents
visSectionFirst	The first section for a shape
visSectionLast	The last section for a shape
visRowFirst	The first row in any section
visRowLast	The last row in any section

Error constants

The following constants are returned when a program is unable to retrieve a section or row:

- **visSectionNone**. The requested section could not be retrieved.
- **visRowNone**. The requested row could not be retrieved.

Index

B

C

D

E

F

G

H

I

M

N

O

P

S

T

U

V

W

X

Z