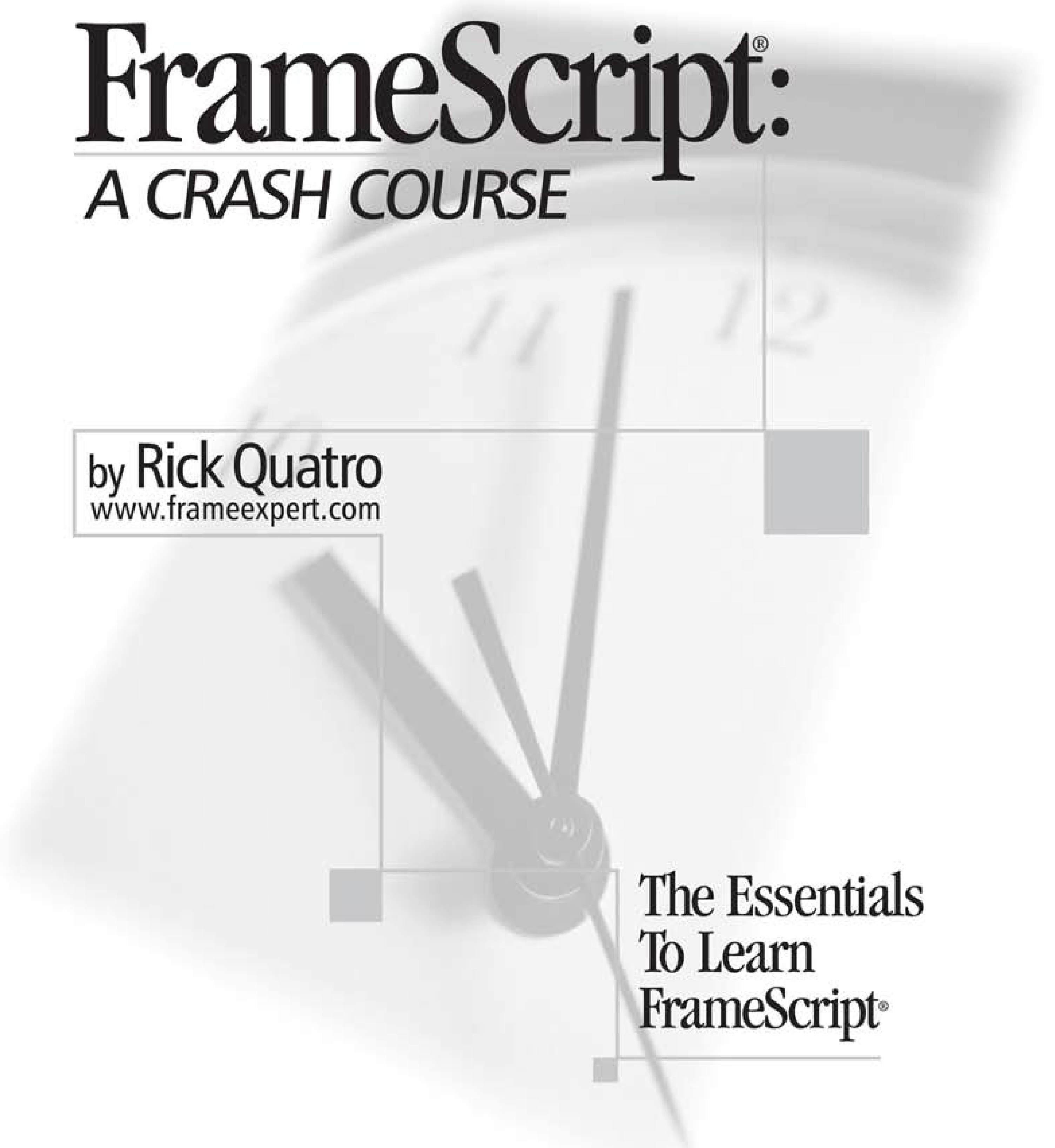
FrameScript®: A CRASH COURSE



by Rick Quatro
www.frameexpert.com

The Essentials To Learn FrameScript®

Edited by Lisa Bronson

FrameScript®: A Crash Course

First Edition published September 2002 in the United States of America.
Revised First Edition published February 2003.

Carmen Publishing is not affiliated in any way with ElmSoft, Inc., the makers of FrameScript, or with Finite Matters, Ltd., distributors of FrameScript.

The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Rick Quatro or Carmen Publishing. Rick Quatro and Carmen Publishing assume no responsibility or liability for any errors or inaccuracies that may appear in this book.

Published by

Carmen
www.frameexpert.com
PUBLISHING

557 County Line Road
Hamlin, New York 14464
rick@frameexpert.com
http://www.frameexpert.com

Foreword

About the Author

Rick Quatro is owner of Carmen Publishing, an upstate New York company providing FrameMaker solutions and support. He has been using FrameMaker since 1993 and FrameScript since its initial release. Rick's focus is on FrameMaker automation with FrameScript. He has written over 1,500 scripts for clients worldwide, including Eastman Kodak, Novell, Harris, and Lockheed Martin.

Rick lives in the Rochester, New York region with his wife, Sherry, and eight sons (James, Daniel, Stephen, Carmen, Jonathan, Jason, Timothy, and Nathan).

About the Editor

Lisa Bronson is an Associate Technical Writer for Evergreen Packaging Equipment in Cedar Rapids, Iowa. She writes, illustrates, and maintains instruction manuals, coordinates translation projects, and is the software guru of the Evergreen publications group.

Outside of technical writing, Ms. Bronson is an aspiring screenwriter and is also the Writers Guild Director for Cedar Rapids Independent Films.

Ms. Bronson lives in Cedar Rapids with her three children. She spends her free time writing, singing, reading, learning, and watching lots of movies.

Acknowledgements

First and foremost, I would like to thank my Lord Jesus Christ for saving me on April 6, 1984 and giving me a new life. I want to thank Him for His love, forgiveness, and promise of eternal life.

Thank you to my beautiful wife Sherry who has been a wonderful partner and friend. Thanks to my eight boys that give purpose and fun to my life.

Thank you to my father, Dr. Richard J. Quatro, for being a great example and role model.

Thank you to Brad Anderson of FrameUsers.com for his encouragement and gentle "prodding" to attend the FrameUsers Workshop. Thanks for being a worthy leader in the FrameMaker community.

Thanks to Frank Elmore, FrameScript's lead developer, for patiently answering my emails as I learned FrameScript. And thanks for designing a terrific program that has captivated me and given my career a new direction.

Thank you to my editor Lisa Bronson, who had to work on very short notice. I take full responsibility for any errors or ommissions that got through.

Thanks to Ron Brancato (http://www.brancatocreative.com) for the cover design.

Introduction

I love to read. I read the Bible through at least once a year. I read Bible commentaries, books on current events (my favorite secular author is Thomas Sowell), and computer books. I even read computer manuals! But writing is another story. Reading books has given me a desire to write something myself, but for me, writing is a slow and awkward process.

It reminds me of childhood days watching professional sports. Being a spectator motivated me to want to get out and *play*. Of course, when I got out there, I was throwing 20-yard floaters instead of 60-yard bombs; instead of dunking from the foul line, I was missing layups. But at least I was involved with the action.

I feel a little bit like that with this book. Because of my vast experience with FrameScript, people have been suggesting that I write a book for the last few years. While I desired to do so, I wasn't sure if I was up to the task. To continue the sports analogy, would I strike out with the bases loaded? But there is definitely a need for FrameScript training materials, so I decided to jump into the game and participate. Hopefully, my writing will not hinder your understanding of the material.

My goal with this book is to contribute some much needed information on FrameScript concepts, combining explanations with tutorials and plenty of code samples. The book doesn't cover *every* FrameScript topic, but it will get you well on your way to writing useful, productive scripts. You will explore the language syntax and learn strategies for solving real-world problems. FrameScript is a tremendous tool that can provide a gold mine of FrameMaker productivity. Hopefully, this book will help you to quickly unlock its treasures.

How To Use This Book

You should read the first three chapters through in order, especially if you are new to FrameScript. They cover foundational material that you will use in every script you write. Chapters 4–9 explore specific topics that you can study in any order, as you need them. Some of the chapters contain short tutorials to help reinforce the material in that chapter.

Full-length tutorials begin with Chapter 10. The scripts you write in the tutorials should be useful in your own production environment. And, they can serve as patterns for other scripts. If you don't understand a particular concept as you work through the examples, you can refer to the relevent chapter earlier in the book.

This book is not designed to replace the FrameScript documentation. You will need the *FrameScript Scriptwriter's Reference* to look up command, object, and property syntax. I suggest that you use Acrobat Catalog to index the FrameScript documentation. Then you can use the Search command to quickly locate the information that you need.

About the Code Listings

Nearly all of code listings in the book are included on the companion CD-ROM. They are organized by chapter and listing number to make them easy to find. To help you learn faster, I suggest that you type out the listings yourself as you experiment with the code. Note that many of the code listings are not complete scripts, but are partial snippets that are being discussed in the book. The complete scripts that are included have descriptive names.

I also suggest that you read the comments in the code listings. They will explain things in the code that may not have been discussed in that particular section of the book. Hopefully, they will encourage you to generously comment your own scripts.

All of the code on the CD is copyrighted; however, that does not mean that you cannot use them in your own scripts. It only means that you cannot copy, distribute, or sell the code as is, without permission of Rick Quatro or Carmen Publishing (http://www.frameexpert.com).

A Word of Caution

FrameScript scripts can radically alter or destroy the contents of your FrameMaker files. Scripts can even delete files on your computer (if you use the right commands). Be careful as you write and test your scripts! If a script will significantly alter your document or book, make sure you test it on copies of your files. FrameMaker's Undo command will not reverse the actions of a script.

That said, Carmen Publishing and Rick Quatro are not responsible for any data loss that may result from the use of the code in this book.

Future Editions

It is almost certain that there will be a second edition of this book (Lord willing). Frankly, there are concepts that were left out for the sake of time. For those of you that purchased this first edition (or received it as part of a training class), you will receive the next edition at a substantially discounted price.

Please email comments and suggestions for future editions to rick@frameexpert.com. Your suggestions will help me to determine what to include in a second edition.

FrameScript Questions

The best place to get specific FrameScript questions answered is on the FrameScript User Group at http://groups.yahoo.com/group/framescript-users. When you join, you can post questions and use the archives.

Table of Contents

1 Getting Started

2 Objects and Properties

3 *Control Structures*

4 *Documents and Books*

5 *Working With Text*

6 *Working With Graphics*

11 *Autonumber Report – Part 2*

12 *Automatic Callouts*

13 *Applying the Default Paragraph Font*

14 *Code Tester and Extractor*

1 Getting Started

This chapter will introduce you to FrameScript basics and get you on your way to writing productive scripts.

What is a Script?

In its simplest form, a script is one or more FrameScript commands. Here is a one-line script that displays grid lines in the active document.

```
Set ActiveDoc.ViewGrid = 1;
```

Normally, your scripts will be much longer than one line, but a useful script may contain only a dozen lines or so.

Scripts will be made up of FrameScript commands, FrameMaker objects, and object property values. In our one-line example, the word **Set** is a FrameScript command that is used to set a property value. *Commands* are usually verbs that perform an action on an object. **ActiveDoc** is a FrameMaker object that represents the document on the screen that has input focus. **ViewGrid** is one of the many properties that a document has; it is equivalent to the **View** > **Grid Lines** command in the FrameMaker interface. Finally, **1** is the value that we are assigning to this property, which means **True** or "on".

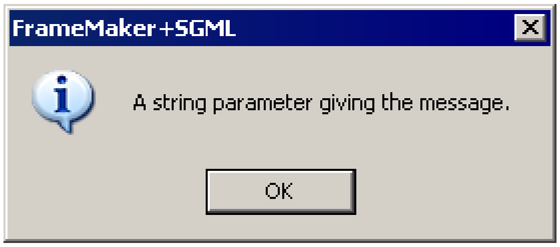
Commands and Parameters

FrameScript commands are one- or two-word statements that describe what the command does. For instance, the **MsgBox** command displays a message box to the user, and the **New AFrame** command creates an anchored frame. All FrameScript commands should end with the semi-colon character (;). The only exceptions are control structure commands **If/Else/EndIf**, **Loop/EndLoop**, **Sub/EndSub**, and **Event/EndEvent**.

Commands contain required or optional parameters. A *parameter* is specific information that a command uses to execute correctly. For example, the **MsgBox** command requires a string parameter that gives the message that it is supposed to display.

Code Listing 1-1

```
MsgBox 'A string parameter giving the message.    ';
```

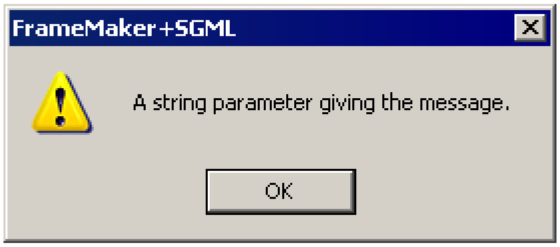


A string parameter giving the message.

In this case, the string is a required parameter because the message dialog box will not display without it. Some parameters are optional; the MsgBox command has an optional **Mode** parameter that alters the style of the dialog box.

Code Listing 1-2

```
MsgBox 'A string parameter giving the message.    ' Mode(Warn);
```



A string parameter giving the message.

The **Warn** value on the **Mode** parameter makes the warning icon appear in the dialog box.

Commands will often use default values when the parameter is not used. The **MsgBox** command uses **Mode(Note)** as the default value when **Mode** is not specified. That means that the following two commands are equivalent, and both dialog boxes will show the note icon.

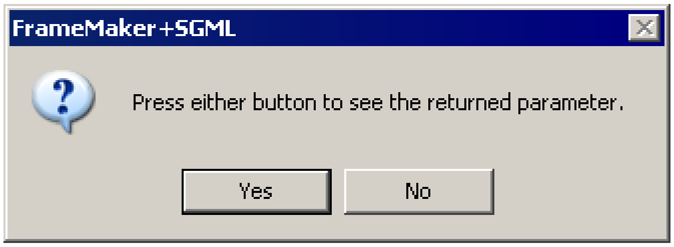
Code Listing 1-3

```
MsgBox 'A string parameter giving the message.    ';
MsgBox 'A string parameter giving the message.    ' Mode(Note);
```

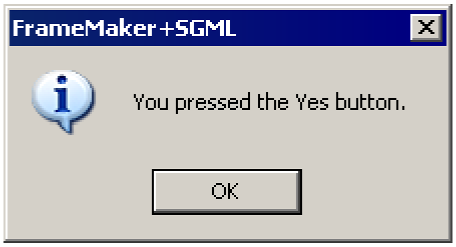
Many commands contain more than one parameter. Some parameters provide information *to* the command; other parameters return information *from* the command. Here is the **MsgBox** command using the **Mode(YesNo)** and **Button** parameters. The **Button** parameter returns information from the **MsgBox** command that tells what button the user pressed.

Code Listing 1-4

```
MsgBox 'Press either button to see the returned parameter.    '
  Mode(YesNo) Button(vButton);
If vButton = YesButton
  MsgBox 'You pressed the Yes button.    ';
Else
  MsgBox 'You pressed the No button.    ';
EndIf
```



Press either button to see the returned parameter.



You pressed the Yes button.

When you use a parameter to return information from a command, you assign a variable to the parameter, in this case **vButton**.

The order of parameters in a command is not significant. For example, the following two commands are equivalent.

Code Listing 1-5

```
MsgBox 'Press either button to see the returned parameter.    '
  Mode(YesNo) Button(vButton);

MsgBox Button(vButton) Mode(YesNo)
  'Press either button to see the returned parameter.    ';
```

When you want to perform a task with FrameScript, you can look in the *FrameScript Quick Reference* to try to find the appropriate commands. FrameScript commands and parameters are listed in detail in the *FrameScript Scriptwriter's Reference.*

White Space and Comments

FrameScript is not particular how scripts are formatted as long as each command ends with a semicolon and the syntax is correct. However, you should strive to make your scripts readable so that they can be easily edited and maintained. You can do this with white space and comments in your scripts.

Here is a script that loops through all of the paragraphs in the active document and writes the text of Heading1 paragraphs to the Console.

Code Listing 1-6

```
If ActiveDoc = 0
MsgBox 'There is no active document.    ';
LeaveSub;
Else
Set vCurrentDoc = ActiveDoc;
EndIf
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
If vPgf.Name = 'Heading1'
Write Console vPgf.Text;
EndIf
EndLoop
```

Although this script is less than a dozen lines, it is hard to follow because it is not formatted at all with white space. Effective white space in a script can take two forms. First, you can separate blocks of code with blank lines. Second, you can indent lines that are inside control structures, such as **If**/**Else**/**EndIf** and **Loop**/**EndLoop** statements.

Comments are helpful because they document your script so that you or others can follow the code if it later needs to be revised. Comments and white space can also be used to make blocks of your code portable so that they can be reused in other scripts.

Here is the previous script enhanced with white space and comments.

Code Listing 1-7

```
// Test for an active document.
If ActiveDoc = 0
  // If no document exists, display a message and exit.
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  // Set a variable for the active document.
  Set vCurrentDoc = ActiveDoc;
EndIf

// Loop through each paragraph in the document.
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  // See if the paragraph is a Heading1 paragraph.
  If vPgf.Name = 'Heading1'
    // Write the text to the Console window.
    Write Console vPgf.Text;
  EndIf
EndLoop
```

There are two kinds of comments that you can use: line comments and block comments. Line comments are used in the previous example and begin with two forward slashes (//). Anything after the two forward slashes is considered a comment, up to the end of the line. Block comments are used in pairs and begin with a forward slash and asterisk (/*) and end with an asterisk and forward slash (*/). Block comments can span multiple lines; everything between them is considered a comment. A typical use for block comments is for multi-line information at the beginning of a script.

Code Listing 1-8

```
/* SaveAsMif.fsl Version 1.2
Copyright 2002, Carmen Publishing. All rights reserved.

This is an event script that saves a copy of the current
document as MIF whenever the document is saved. The script must
be installed, not run. You can use an Initial Script to
install it automatically when FrameMaker starts.

For more information, contact rick@frameexpert.com.
*/
```

You should avoid using block comments for single line comments in your scripts. To see why, consider this example code that uses block comments for single line comments.

```
/* Loop through each paragraph in the document. */
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  /* See if the paragraph is a Heading1 paragraph. */
  If vPgf.Name = 'Heading1'
    /* Write the text to the Console window. */
    Write Console vPgf.Text;
  EndIf
EndLoop
```

As you are developing and testing your script, you may need to comment out a whole block of code. You won't be able to easily do this if you have used block comments for line comments. For example, adding block comments before and after the block above will not work, because the */ at the end of the second line will cancel out any previous /* characters. This means that the whole block will not be commented out.

```
/* Attempt to comment out this block of code. Won't work!
/* Loop through each paragraph in the document. */
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  /* See if the paragraph is a Heading1 paragraph. */
  If vPgf.Name = 'Heading1'
    /* Write the text to the Console window. */
    Write Console vPgf.Text;
  EndIf
EndLoop
*/
```

If you use line comments throughout your code, you can then use block comments to comment whole sections of code, as in the example below.

Code Listing 1-9

```
/* Successfully comment out this block of code.
// Loop through each paragraph in the document.
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  // See if the paragraph is a Heading1 paragraph.
  If vPgf.Name = 'Heading1'
    // Write the text to the Console window.
    Write Console vPgf.Text;
  EndIf
EndLoop
*/
```

Two Kinds of Scripts

Standard Scripts

Scripts can take two major forms. One is a *standard script* that contains one or more commands, which can be followed by one or more subroutines. Standard scripts are read into memory each time they are run and exit from memory when they are finished. The code in a standard script that is not part of a subroutine is called the *body* of the script. The body of a script must come before any subroutines or events that may be in the script.

Code Listing 1-10

```
// The body of the script.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  Set vCurrentDoc = ActiveDoc;
EndIf

// Call a subroutine.
Run ProcessMarkers;

// Body of script ends here. Any subroutines go below the body.

Sub ProcessMarkers
//
...
//
EndSub
```

Event Scripts

The second type of script is an *event script*. An event script consists of one or more events or subroutines. There is no body in an event script. *Event scripts are not run, but are installed*. When you install an event script, it is loaded into memory and "waits" for an event to trigger it. An event can be a menu command, a shortcut key, or a FrameMaker event, such as when a document is opened or saved. An event script stays in memory until the script is uninstalled or FrameMaker exits.

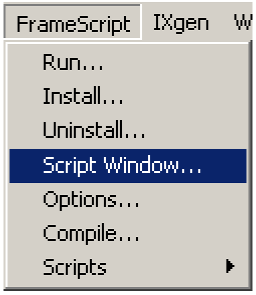
Here is an event script that responds to a FrameMaker event; it is triggered after a document is saved. It automatically saves a copy of the document in MIF format. Notice that the script does not have a body.

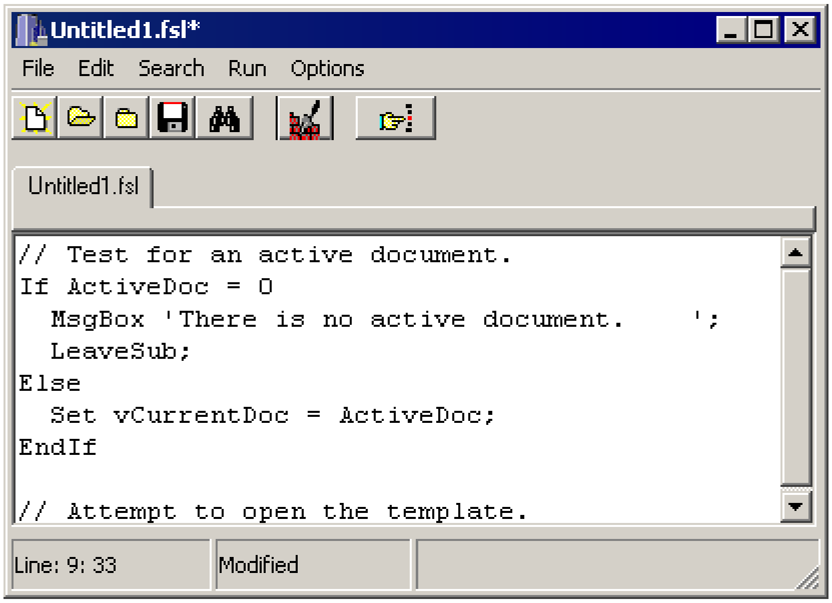
Code Listing 1-11

```
Event NotePostSaveDoc
//
// Find the .fm extension on the file.
Find String('.fm') InString(FileName) Backward NoCase
  ReturnStatus(vFound) ReturnPos(vPos);
If vFound
  // If the extension is found, drop it and add the .mif extension.
  Get String FromString(FileName) EndPos(vPos-1)
    NewVar(vNameNoExt);
  Set vMifName = vNameNoExt+'.mif';
Else
  // If there is no extension, add the .mif extension.
  Set vMifName = FileName+'.mif';
EndIf
// Save the document as a MIF file.
Save Document DocObject(FrameDoc) File(vMifName)
  FileType(SaveFmtInterchange);
//
EndEvent
```

Writing Scripts

You can use the Script Window to write your scripts. You can also run standard scripts from the Script Window. To open the Script Window, choose **FrameScript > Script Window**.





```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  Set vCurrentDoc = ActiveDoc;
EndIf

// Attempt to open the template.
```

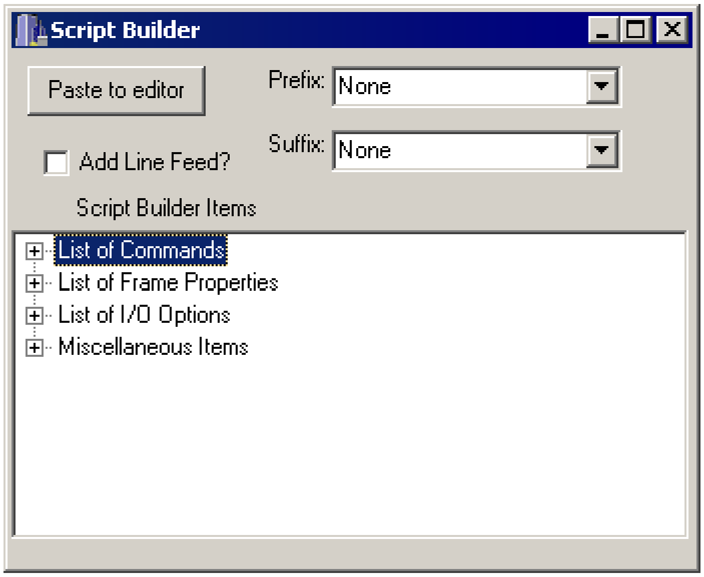
You can use the commands and buttons in the Script Window to save and run your scripts. The following table describes the buttons on the Script Window.

Button	Function	Description	Equivalent Menu Command
🗋	New Script	Opens a new script tab in the Script Window	File > New
📂	Open Script	Prompts to open an existing script	File > Open
📁	Close Script	Closes the front-most script	File > Close (Ctrl+W)
💾	Save Script	Saves the front-most script	File > Save (Ctrl+S)
🔍	Find Text	Finds text in the front-most script	Search > Find (Ctrl+F)

Button	Function	Description	Equivalent Menu Command
	Script Builder	Opens the Script Builder window	Run > Script Builder (Ctrl+B)
	Run Script	Runs the script in the front-most window	Run > Script (Ctrl+R)

The Script Window is an adequate environment for prototyping and troubleshooting small scripts. Follow these guidelines when using the Script Window.

- **Save your scripts often.** Since you can run scripts in the Script Window without saving them, you will lose unsaved changes if the script causes FrameMaker to crash.
- **Save your changes before switching to another tab**. Sometimes the contents of script tabs can get lost as you switch from one tab to another. The contents get replaced with a 0 (zero). If the script has been replaced with a 0 when you switch back to it, close the script without saving changes.
- **Use the Script Builder to help with the script syntax**. Choose **Run > Script Builder** or click the Script Builder button to open it. The Script Builder contains lists of FrameScript commands and properties that you can look up and paste into the Script Window.

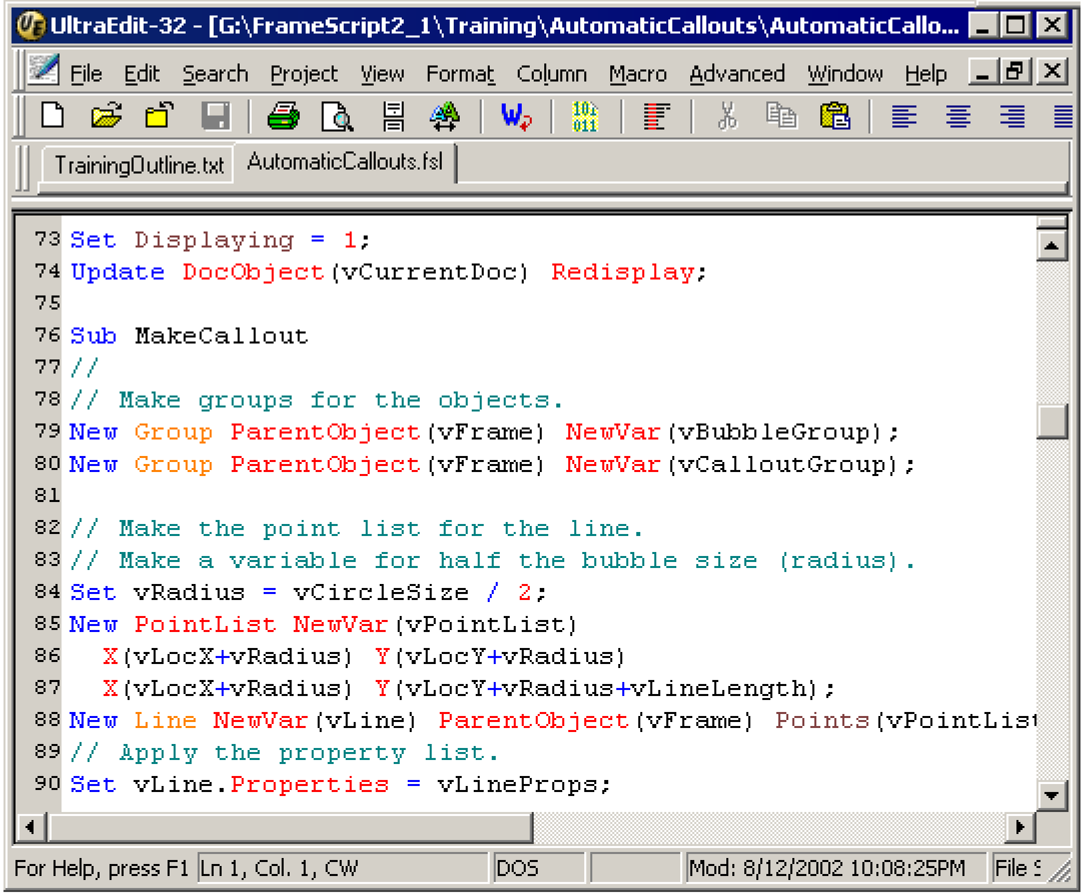


Prefix: None

Suffix: None

□ Add Line Feed?

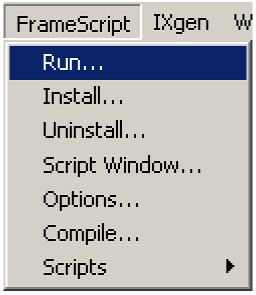
For longer scripts and serious script development, you should consider using a separate text editor to write your scripts in. A good shareware choice for the PC is UltraEdit (http://www.ultraedit.com) which is around $35. It has a lot of useful features for writing scripts, including line numbering and syntax highlighting. With syntax highlighting, different components of the FrameScript language are displayed in different colors. For example, comments appear in green, commands in blue, parameters in red, etc. Below is a screenshot of a script opened in UltraEdit.


UltraEdit-32 - [G:\FrameScript2_1\Training\AutomaticCallouts\AutomaticCallo...
File Edit Search Project View Format Column Macro Advanced Window Help
TrainingOutline.txt
AutomaticCallouts.fsl
For Help, press F1
Ln 1, Col. 1, CW
DOS
Mod: 8/12/2002 10:08:25PM
File S

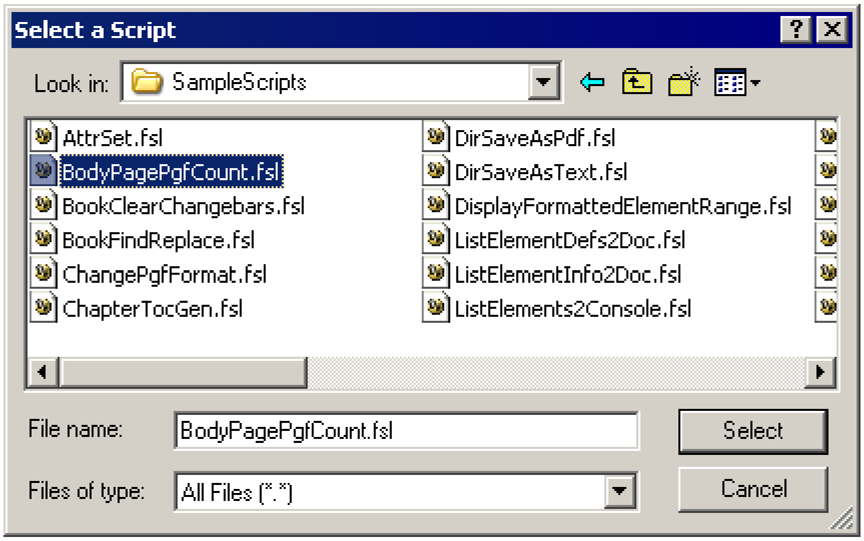
```
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;

Sub MakeCallout
//
// Make groups for the objects.
New Group ParentObject(vFrame) NewVar(vBubbleGroup);
New Group ParentObject(vFrame) NewVar(vCalloutGroup);

// Make the point list for the line.
// Make a variable for half the bubble size (radius).
Set vRadius = vCircleSize / 2;
New PointList NewVar(vPointList)
   X(vLocX+vRadius) Y(vLocY+vRadius)
   X(vLocX+vRadius) Y(vLocY+vRadius+vLineLength);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointLis
// Apply the property list.
Set vLine.Properties = vLineProps;
```

Running Scripts

When you write a script, you save it in a plain ASCII text file, typically with an .FSL extension. To run a standard script that has already been saved to disk, you use the **FrameScript** > **Run** command.



Navigate to the script that you want to run and click **Select**.

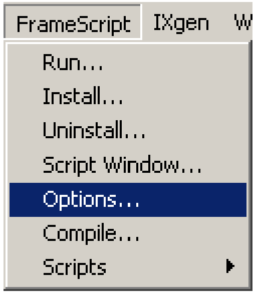


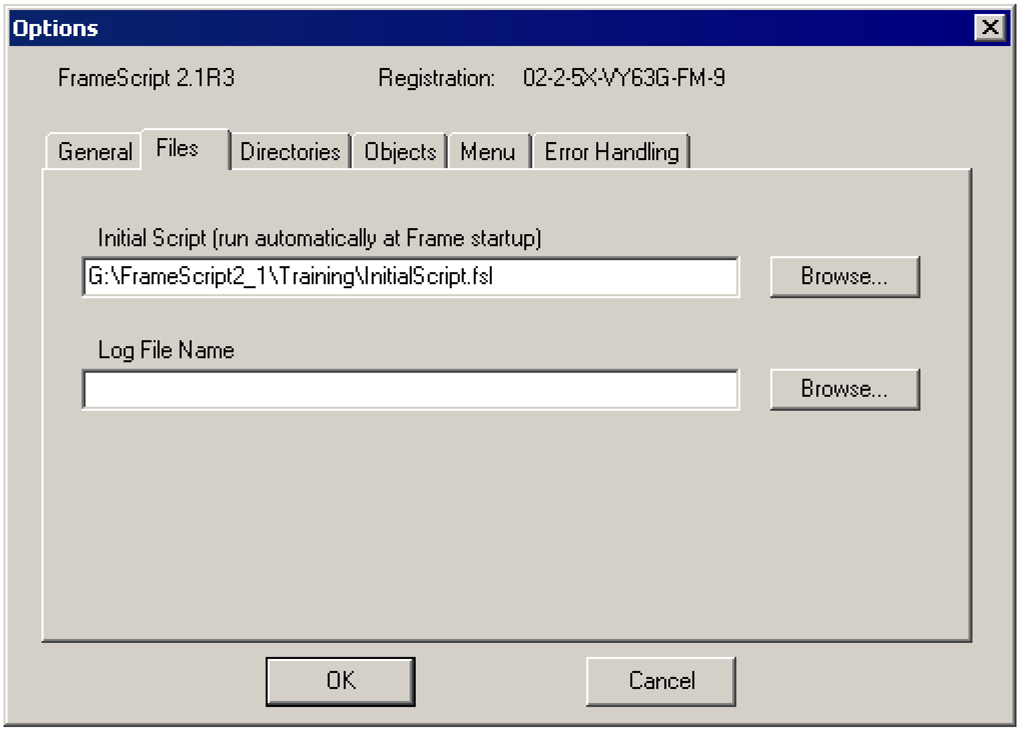
You can also run standard scripts that are open in the Script Window by choosing **Run** > **Script** or by clicking the Run button.

Running Scripts Automatically

FrameScript can run a script automatically when FrameMaker launches. This is called an *Initial Script*. An Initial Script can be any standard script; it is used most often to install other scripts when FrameMaker starts. See “Installing Scripts Automatically” on page 1-15. Any variables set in an Initial Script are global variables and stay in memory as long as FrameMaker is running.

To make a script an Initial Script, choose **FrameScript** > **Options** and click on the **Files** tab.





FrameScript 2.1R3 Registration: 02-2-5X-VY63G-FM-9

Initial Script (run automatically at Frame startup): G:\FrameScript2_1\Training\InitialScript.fsl

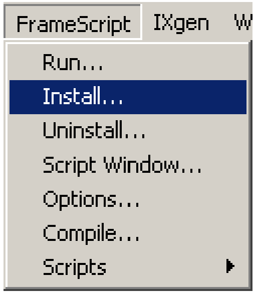
Log File Name:

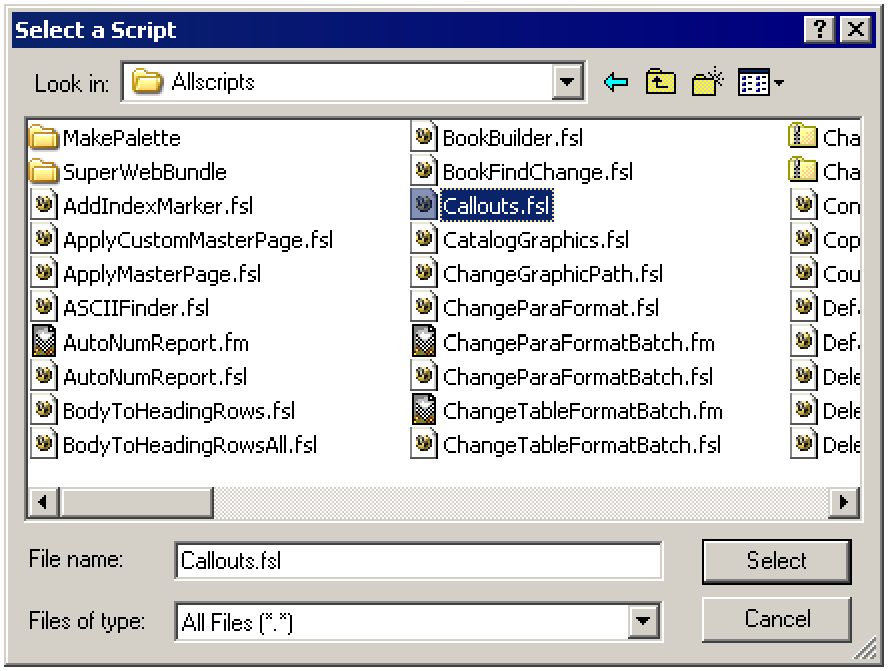
Click the Browse button next to the Initial Script field and navigate to the script that you want to use for your Initial Script. Click OK to dismiss the Options dialog. The Initial Script will run the next time you start FrameMaker.

Installing and Uninstalling Scripts

Standard Scripts

You can install standard scripts so that they can be run from the **FrameScript** > **Scripts** submenu. Choose **FrameScript** > **Install** and navigate to the script that you want to install.

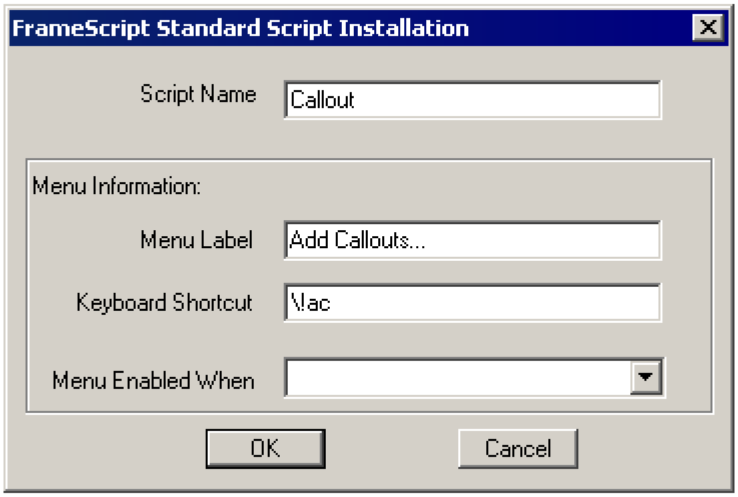




File name: Callouts.fsl

Files of type: All Files (*.*)

You will be prompted with the **Standard Script Installation** dialog.



Script Name: Callout

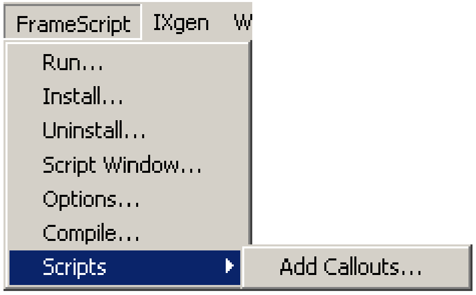
Menu Label: Add Callouts...

Keyboard Shortcut: \!ac

Menu Enabled When:

Choose a name for the script; if you leave the **Script Name** field blank, FrameScript will name the script for you (StandardScriptName1, StandardScriptName2, etc.). The **Menu Label** field is the name that will appear in the **FrameScript > Scripts** submenu. You can also choose a keyboard shortcut for the script and determine when the menu command is available.

When you click OK, your script will appear in the **FrameScript > Scripts** submenu.



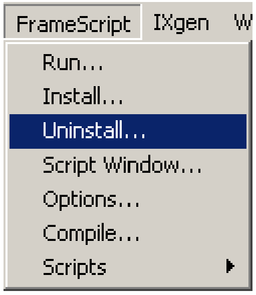
If you develop your scripts in a separate text editor, you can edit and test your scripts interactively by following the steps below. These steps will only work with standard scripts, not event scripts.

1. Open your script in the text editor. Make sure there is at least one command in the body of the script, and save the script.
2. Leave the script open in the text editor and switch to FrameMaker. Choose **FrameScript** > **Install** and choose the script you have open in the text editor.
3. When prompted with the **Standard Script Installation** dialog, enter a **Menu Label** and, if you like, a **Keyboard Shortcut** for the script.

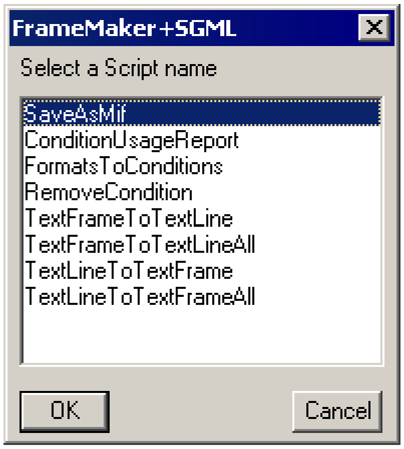
You will see the script installed in the **FrameScript** > **Scripts** submenu.

You can now switch back and forth between FrameMaker and the text editor. When you make changes to the script, make sure you save the script in the text editor before running the script in FrameMaker. The last saved version of the script will be run when you choose it from the **FrameScript** > **Scripts** submenu.

To uninstall a script, choose **FrameScript** > **Uninstall**.



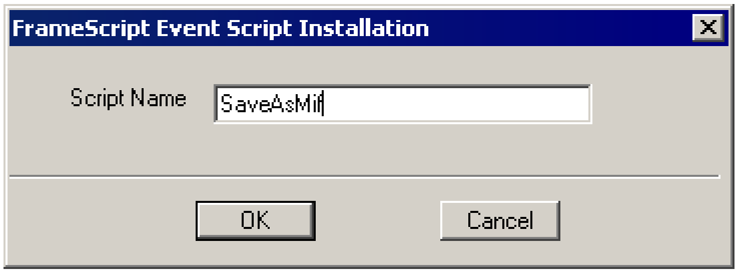
Select the script name in the dialog box and click OK.



Select a Script name

Event Scripts

You must install an event script before you can test it. To install an event script, choose **FrameScript** > **Install** and navigate to the script that you want to install. You will be prompted to name the event script.



Script Name: SaveAsMif

If you are prompted with the **Standard Script Installation** dialog instead, then you chose a standard script instead of an event script. If you are certain that you chose an event script, check the script to make sure that there is no code outside of events or subroutines in the script.

When an event script is installed, it is loaded in memory and waits for an event to occur. It stays in memory until it is uninstalled or until FrameMaker exits. As a result, when you make changes to an event script, you must uninstall it and reinstall it for the changes to take effect. To uninstall a script, choose **FrameScript** > **Uninstall**. Select the script name in the dialog box and click OK.

Installing and Uninstalling with FrameScript Commands

FrameScript has commands for installing and uninstalling scripts. One use for these commands is to make it easier to test event scripts. You can make a “Reset” script that uninstalls and installs the event script. When you make a change to the event script, you save the changes, and run the Reset script, which quickly uninstalls and reinstalls the event script so that the latest version is in memory. Here is an example that works with the SaveAsMif.fsl event script shown earlier. We are assuming that the script is located in the same folder as the FrameScript .DLL.

Code Listing 1-12

```
// Reset script for SaveAsMif.fsl script.
// Uninstall the script.
Uninstall Script Name(SaveAsMif);
// Reinstall the script. ClientDir is the folder containing the
// FrameScript DLL or Module.
Install Script File(ClientDir+DIRSEP+'SaveAsMif.fsl')
  Name('SaveAsMif');
```

The **Name** parameter value must be the same in both commands. The **File** parameter on the **Install Script** command must be the path to the event script file. In this example, we are using the FrameScript **ClientDir** variable which gives the path where the FrameScript .DLL or module is installed. **DIRSEP** is a FrameScript variable that gives the folder separator character (backslash on the PC, colon on the Mac). Note that if the script is not found, FrameScript will give an error on the **Install Script** command. In constrast, the **Uninstall Script** command does not return an error if you try to uninstall a script that is not installed.

To use the Reset script, save it and install it as a standard script in the **FrameScript > Scripts** submenu. When you make changes to the SaveAsMif event script, save the changes, and run the Reset script. It will uninstall the old version from memory and install the last saved version from the disk.

Installing Scripts Automatically

FrameScript can install scripts automatically each time FrameMaker starts by using an Initial Script. Include your install commands in a standard script and designate that script as an Initial Script. See "Running Scripts Automatically" on page 1-10.

Here is a script that installs the SaveAsMif event script and its corresponding Reset script. If this script is saved and designated as an Initial Script, it will install the SaveAsMif scripts automatically when FrameMaker is started. This example requires that the Initial Script be saved in the same folder as the other scripts.

Code Listing 1-13

```
// Initial Script for loading SaveAsMif scripts.
// Find the folder where this script is located.
Find String(DIRSEP) InString(ThisScript) Backward ReturnPos(gPos);
Get String FromString(ThisScript) EndPos(gPos)
  NewVar(gScriptsPath);

// Install the SaveAsMif event script.
Install Script File(gScriptsPath+'SaveAsMif.fsl')
  Name('SaveAsMif');

// Install the Reset standard script.
Install Script File(gScriptsPath+'ResetSaveAsMif.fsl')
  Name('ResetSaveAsMif') Label('Reset SaveAsMif Script');
```

Note the following points regarding this script.

- The **Find String** command uses the FrameScript **ThisScript** variable. This variable returns the full path of the script that contains it. We use this to get the path for the other scripts. This means that the Initial Script can be installed anywhere as long as it is in the same folder as the other scripts.

- Variables created in this script are named with a "**g**" prefix to distinguish them as global variables. This is a reminder that variables created in an Initial Script are global in scope and will persist in memory while FrameMaker is running. Global variables cannot be changed by other scripts. To avoid conflicts, you should avoid using the the global variable names in other scripts.

Strategies for Writing Scripts

Divide and Conquer

When you decide to write a script, you will often think in terms of the overall task that you want to perform. For example you might say, "I want a script that will center-align all of the anchored frames in the document. And, I want it to add a half-point border around each frame." The best way to begin is to break up the overall task into small steps. It will be easier if you make a list.

- Center-align an anchored frame
- Add a half-point border to an anchored frame
- Loop through all of the anchored frames in a document

You should write separate scripts for each task. When you are certain that each one works the way you want it to, you can combine them into a larger script that will perform your overall task.

Start With Single Items

If you want to modify anchored frames, you should pick a single anchored frame to experiment with. The basic code will be the same whether you are modifying one frame or a thousand, but it is certainly safer to experiment with one! FrameMaker has objects that allow you to work with selected items in your document. For example, there is a **FirstSelectedGraphicInDoc** property that returns a selected object in your document. If you select an anchored frame, you can use this property to change its properties.

Code Listing 1-14

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected graphic.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Display the anchored frame’s properties.
Display vAFrame.Properties;
```

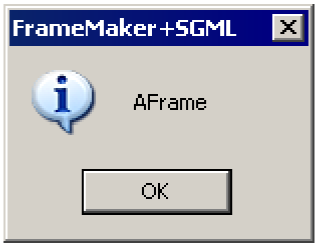
Use the Documentation

Now that you have an anchored frame object, you can experiment with changing its properties. In this case, you should look up the **AFrame** object in the *FrameScript Scriptwriter's Reference*. If you are not sure what an object is called, you can use this code to display its **ObjectName**.

Code Listing 1-15

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Set a variable for the selected graphic.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Display the anchored frame’s ObjectName property.
Display vAFrame.ObjectName;
```



FrameMaker+SGML

AFrame

Chapter 2 of the *FrameScript Scriptwriter's Reference* gives the properties for each kind of FrameMaker object. If you look up the **AFrame** object, you will find an **Alignment** property and a list of possible values. You can now try this with the selected anchored frame.

Code Listing 1-16

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected graphic.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Center-align the anchored frame.
Set vAFrame.Alignment = AlignCenter;
```

You should see your selected frame become center-aligned. Now you can find the property that will set a border on the frame. Actually, two properties are required to do this; **BorderWidth** sets the width of the border, and **Pen** sets the border style. You can add the code to your script to modify the border.

Code Listing 1-17

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected graphic.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Center-align the anchored frame.
Set vAFrame.Alignment = AlignCenter;

// Add the black border to the anchored frame.
Set vAFrame.BorderWidth = .5pt;
Set vAFrame.Pen = FillBlack;
```

Mark Your Trail

You should save each portion of your script when you get it working as expected. It may seem silly to save small snippets like this, but it will give you a place to go back to if you have problems with the larger script. As you gain more experience with FrameScript, you can do this less often.

Move to the Next Step in the List

Now you can move on to the next step in your list. In our example, we want to loop through all of the anchored frames in the document. You can look up the **Doc** (document) object in the *FrameScript Scriptwriter's Reference* to find a property to start with. See “Objects and Properties” and “Control Structures” for more information.

Here is a simple script that loops through all of the graphics in a document.

Code Listing 1-18

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first graphic in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
// Loop through the graphics in the document.
Loop While(vGraphic)
  // Write the graphic to the Console window.
  Write Console vGraphic;
  // Move to the next graphic.
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```

We are writing each graphic's object to the FrameMaker Console window. This is a way to test the code before it actually does something useful. We want the script to only process anchored frames, so we need to add a test inside the loop.

Code Listing 1-19

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first graphic in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
// Loop through the graphics in the document.
Loop While(vGraphic)
  // Test to see if the graphic is an anchored frame.
  If vGraphic.ObjectName = 'AFrame'
    // Write the graphic to the Console window.
    Write Console vGraphic;
  EndIf
  // Move to the next graphic.
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```

Make a Library of Reusable Code

If you run this script, you will see that it only lists anchored frames (**AFrame**) to the Console. At this point, this task is complete and you can save this as a separate script. Save it with a descriptive name, such as ProcessAllAnchoredFramesInDoc.fsl. When you want to write another script that processes all of the anchored frames in a document, you can pull this script from your “code library” and use it as a starting point.

Combine the Code

Now you are ready to combine the two scripts so that they perform your overall task. You should set up a sample document with some anchored frames so that you can safely test your script. In our example, we want to change each anchored frame in the document, so parts of the first script will go inside the loop of the second script. Make sure you preserve the first two scripts and combine the code in a third. Below is the combined script. Look at the script and see if you can identify the components from each of the individual scripts. You will also see that some minor changes were made in variable names to make the code work together.

Code Listing 1-20

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first graphic in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
// Loop through the graphics in the document.
Loop While(vGraphic)
  // Test to see if the graphic is an anchored frame.
  If vGraphic.ObjectName = 'AFrame'
    // Center-align the anchored frame.
    Set vGraphic.Alignment = AlignCenter;
    // Add the black border to the anchored frame.
    Set vGraphic.BorderWidth = .5pt;
    Set vGraphic.Pen = FillBlack;
  EndIf
  // Move to the next graphic.
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```

Test and Refine the Script

As you develop your scripts, you will find other things that need to be done. In our example, two immediate tasks come to mind.

- Only modify anchored frames that are anchored below the current line. In other words, skip anchored frames that are anchored at the insertion point, outside of the column, etc.
- Test to make sure a document is active before running the script.

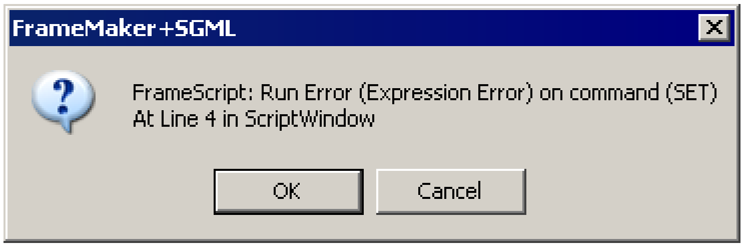
You may discover the first task the hard way; you will run your script and find out it isn’t particular enough about which items to modify. Try to anticipate what the commands in your script will do in a wide variety of circumstances. In general, you should make the scope of your commands as narrow as possible. Here is how we accomplish this for our example script.

Code Listing 1-21

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Set a variable for the first graphic in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
// Loop through the graphics in the document.
Loop While(vGraphic)
  // Test to see if the graphic is an anchored frame.
  If vGraphic.ObjectName = 'AFrame'
    // Test to see if the frame is anchored below the current line.
    If vGraphic.AnchorType = AnchorBelow
      // Center-align the anchored frame.
      Set vGraphic.Alignment = AlignCenter;
      // Add the black border to the anchored frame.
      Set vGraphic.BorderWidth = .5pt;
      Set vGraphic.Pen = FillBlack;
    EndIf
  EndIf
  // Move to the next graphic.
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```

You will discover the second task when you try to run your script with no document open and you get the following error.



FrameScript: Run Error (Expression Error) on command (SET)
At Line 4 in ScriptWindow

Your scripts should include sufficient *error-checking* to anticipate conditions that will cause an error. If your code requires a certain condition, you need to test for it in your script. Here is how we can test to make sure a document is opened.

Code Listing 1-22

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the active document.
  Set vCurrentDoc = ActiveDoc;
EndIf
```

Troubleshooting Scripts

When FrameScript encounters an error in a script, it will usually give you the line number in the script that is causing the error. You can examine that line to see what the error is. That is why it is helpful to use a text editor that shows line numbers. If you are using the Script Window, the lower left hand corner displays the line number of the cursor location. You can use **Search** > **Go To Line** (Ctrl+L) to go to a particular line number in the Script Window.

Many script errors will come from misspelled code. Remember that programming languages are very precise; one incorrect letter can cause an error. Other errors can be caused by using incorrect data types on properties. For example, if you try to set a string property without using single quotes around the string, you may get an error.

You may get compile errors when you try to run a script. These are usually caused by improperly paired control statements; for instance, you have an **If** statement without a corresponding **EndIf** statement. The line number reported on a compile error may not be exact, but the error usually occurs somewhere before the line indicated on the error message.

One special kind of error is a “runaway script.” This occurs when an you accidently make an endless loop. With a runaway script, the script doesn’t stop until FrameMaker crashes. You can try to stop a runaway script by pressing the Escape key on the PC or Command+. on the Mac. Here is an example of a loop that will cause a runaway script.

```
// Don’t run this code!
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first graphic in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
// Loop through the graphics.
Loop While(vGraphic)
  // Write the graphic to the Console window.
  Write Console vGraphic;
  // ***Something is missing here!***
EndLoop
```

There is nothing inside of the loop to move to the next graphic in the document, so the loop will never end. Here is the missing line.

```
  Set vGraphic = vGraphic.NextGraphicInDoc;
```

See “For another method of testing multiple conditions, See “Using a Loop to Handle Multiple Conditions” on page 3-13.” on page 3-3 for more information about loops.

2 Objects and Properties

A majority of your FrameScript code will manipulate FrameMaker objects and properties. An *object* is any FrameMaker component that you may need to work with, including documents, paragraphs, paragraph formats, and anchored frames.

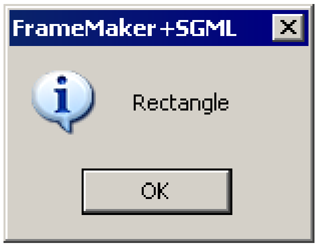
Objects

Each FrameMaker object has an **ObjectName** property that identifies what kind of object it is. **ObjectNames** include **Doc** (document), **Pgf** (paragraph), **PgfFmt** (paragraph format), and **AFrame** (anchored frame).

To illustrate this, use the retangle tool to draw a small rectangle. Leave the rectangle selected and run the following code.

Code Listing 2-1

```
// Set a variable for the first selected graphic in the document.
Set vRectangle = FirstSelectedGraphicInDoc;
// Display the object name of the graphic.
Display vRectangle.ObjectName;
```

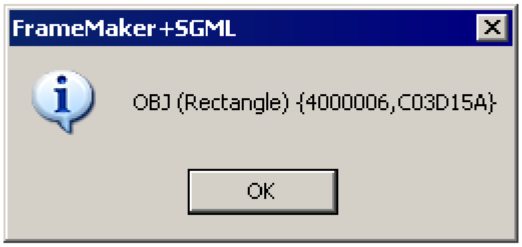


Testing for the Existence of an Object

Each object has a unique ID number. The ID number itself is not normally used in scripts, but it can tell you if an object *exists*. You need to make sure an object exists before trying to read or change its properties. Make sure the rectangle is still selected and run the following script.

Code Listing 2-2

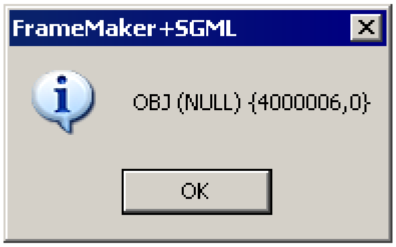
```
// Set a variable for the first selected graphic in the document.
Set vRectangle = FirstSelectedGraphicInDoc;
// Display the object.
Display vRectangle;
```


FrameMaker+SGML

OBJ (Rectangle) {4000006,C03D15A}

OBJ identifies this as an object with the **ObjectName** in parenthesis. The last number in the brackets is the unique ID of the object, and the first number is the “container” object’s unique ID; in this case, the **Doc** object that contains the **Rectangle**.

Deselect the rectangle and rerun the previous code.


FrameMaker+SGML

OBJ (NULL) {4000006,0}

Since there is no selected graphic in the document, **FirstSelectedGraphicInDoc** is **NULL** or **0** (zero). You should test for the existence of an object before attempting to work with it. Here is a simple example that tests for the existence of a selected graphic before displaying its **ObjectName**.

Code Listing 2-3

```
// Test to see if an object is selected.
If FirstSelectedGraphicInDoc = 0
  MsgBox 'Please select a graphic and rerun the script.    ';
  LeaveSub; // Exit the script.
Else
  // Display the graphic's object name.
  Display FirstSelectedGraphicInDoc.ObjectName;
EndIf
```

Objects Have Properties

Each FrameMaker object has a list of properties. A *property* is a particular characteristic of an object. For example, a **Pgf** object has a **SpaceAbove** property that determines how much space is above the paragraph. This corresponds to the Space Above Pgf field on the Basic tab of the Paragraph Designer. Almost any property that can be set in FrameMaker’s interface can be changed with a script. In addition, you can use FrameScript code to change certain properties that are not accessible in the interface.

NOTE: Some properties are read-only, which means that they can be read but not changed.

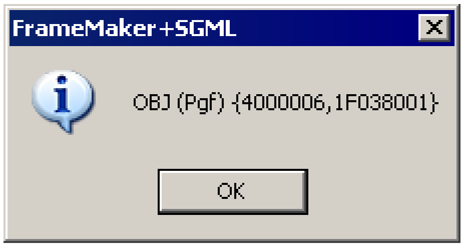
An object’s property list can contain a mix of properties and other objects. For instance, a **Doc** object contains properties like **ViewBorders** and **ViewTextSymbols** that determine if borders and text symbols are displayed in the document. But a **Doc** object also contains other objects, such as **FirstPgfInDoc** and **FirstGraphicInDoc**.

And these objects will have properties and objects of their own. To see this, run the following code with a document open.

```
Display ActiveDoc.ViewBorders;
```

You will see a **1** or a **0** (zero); 1 means **True** (or "on") and 0 means **False** (or "off"). What you are displaying here is the *value* of a property of the **ActiveDoc** (active document) object. The following code demonstrates that an object can contain other objects as well as properties.

```
Display ActiveDoc.FirstPgfInDoc;
```



OBJ (Pgf) {4000006,1F038001}

We see that the **ActiveDoc** object contains a **FirstPgfInDoc** object, which itself contains properties (and perhaps objects).

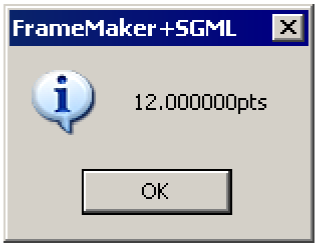
A Hierarchy of Objects

When you launch FrameMaker, a **Session** object is created. The **Session** object is the top-level object that contains all other objects. The **Session** object also has properties, such as **AutoSave** and **ProductName**. These properties affect FrameMaker as a whole and are independent of each document's properties.

A FrameMaker **Session** is made up of a hierarchy of objects and these objects can be nested inside of other objects. In order to manipulate an object by changing its properties, you have to learn how to navigate the hierarchy in order to find the object that you want. Understanding how to do this is one of the keys to learning FrameScript.

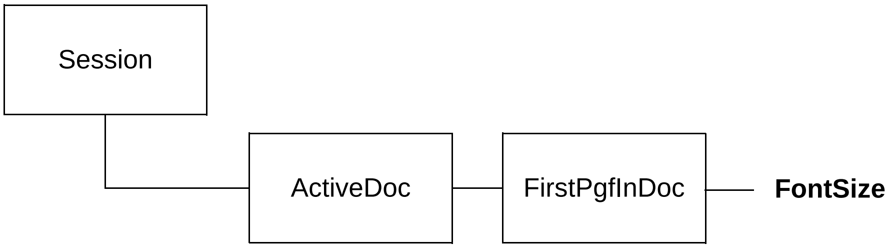
FrameScript uses *dot-notation* to work with objects and properties. Dot-notation is a way of navigating the hierarchy of objects and properties; you have seen this in the brief examples in this chapter. Each dot moves to an object or property that is directly related to the previous one. Here is an example to try with an open document.

```
Display Session.ActiveDoc.FirstPgfInDoc.FontSize;
```



12.000000pts

You have to start with a "known" object to get to the objects and properties that you want. There is only one **Session** object and it is always the top-level object, so we started with it. The **Session** object contains an **ActiveDoc** object, which represents the active document in FrameMaker (the one that has input focus). Every document has a **FirstPgfInDoc** object, which in turn has a **FontSize** property. Here is a simple diagram of how each dot in the code moves to another step in the hierarchy.



In the diagram, objects are represented by rectangles and properties are represented by bold text. Remember that each object in the hierarchy contains properties as well as other objects.

It is possible to leave some objects off when using dot-notation. Since there is only one **Session** object and it is the top-level object, you can usually leave it off. The following code works fine.

```
Display ActiveDoc.FirstPgfInDoc.FontSize;
```

Indeed, it is possible to leave off the ActiveDoc object and the script will still work.

```
Display FirstPgfInDoc.FontSize; // Not recommended!
```

Be careful of doing this, though, because you may want your code to work on a document that is *not* the **ActiveDoc**. For example, when processing all of the documents in a book, you may want to modify the documents without making them the active document. In this case, you will have to use a variable for each document object.

Using the **ActiveDoc** object on your code can cause its own difficulties. The **ActiveDoc** object is a specific **Doc** object representing the document having input focus. You may test your code on the **ActiveDoc**, but it should be portable enough to work on any document. The solution is to set a variable for the ActiveDoc and use the variable in your code.

Code Listing 2-4

```
Set vCurrentDoc = ActiveDoc;
Display vCurrentDoc.FirstPgfInDoc.FontSize;
```

In this script, **vCurrentDoc** represents the **ActiveDoc** object, but it could represent a non-active document in another script.

In the previous examples, we used dot-notation to move “down” the object hierarchy, to the particular property we were looking for. You can also move “up” the hierarchy. Draw a rectangle on the page, leave it selected, and run the following code.

Code Listing 2-5

```
Set vCurrentDoc = ActiveDoc;
Set vRectangle = vCurrentDoc.FirstSelectedGraphicInDoc;
Display vRectangle.FrameParent.PageFramePage.Doc;
```

In this example, we start with a selected **Rectangle** object, move to the unanchored frame (**UFrame**) object containing the **Rectangle**, to the **BodyPage** object containing the page frame, and finally to the **Doc** object containing the page.

Lists of Related Objects

Many FrameMaker objects are the "First" in a list of objects. For instance, a **Doc** object has **FirstPgfInDoc**, **FirstGraphicInDoc**, and **FirstMarkerInDoc** properties. Each of these is an object that represents the first member in a list of related objects. These object lists are known as *linked lists*.

Consider the list of paragraphs in a document. The **FirstPgfInDoc** object is the first member in the linked list of **Pgf** (paragraph) objects. The **Doc** object does not give you direct access to each **Pgf** object in the list, only to the first one in the list (**FirstPgfInDoc**). Each **Pgf** object has a **NextPgfInDoc** property that gives you the next **Pgf** object in the list. You use each paragraph's **NextPgfInDoc** property to move from paragraph-to-paragraph in the list.

Suppose we want to turn off hyphenation for all of the paragraphs in the active document. We can use the following piece of code to do this.

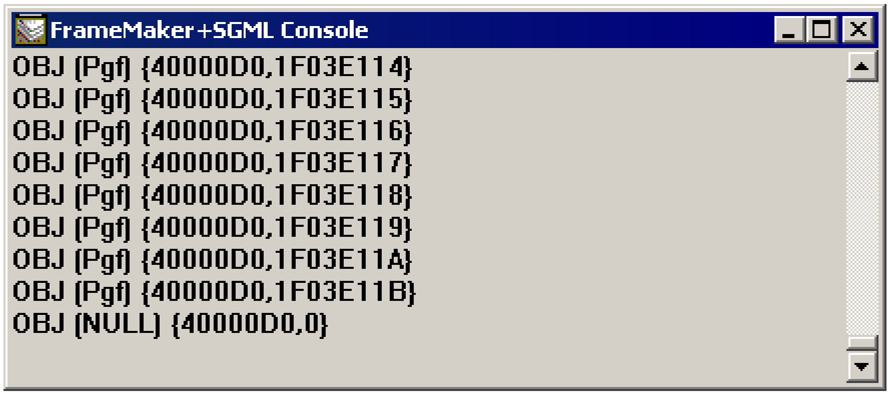
Code Listing 2-6

```
Set vCurrentDoc = ActiveDoc;
// Go to the first pararagraph in the document.
Set vPgf = vCurrentDoc.FirstPgfInDoc;
// Loop through all of the paragraphs in the document.
Loop While(vPgf)
  // Turn off hyphenation for the current paragraph.
  Set vPgf.Hyphenate = 0;
  // Move to the next paragraph in the list.
  Set vPgf = vPgf.NextPgfInDoc;
EndLoop
```

When we have the document object, we have access to the first paragraph in the document. Then we can move from paragraph to paragraph using each **Pgf** object's **NextPgfInDoc** property. When we get to the last paragraph in the document, its **NextPgfInDoc** property is **0** (zero), causing the loop to exit. To further illustrate this, make a new FrameMaker document and run the following script.

Code Listing 2-7

```
Set vCurrentDoc = ActiveDoc;
// Go to the first pararagraph in the document.
Set vPgf = vCurrentDoc.FirstPgfInDoc;
// Loop through all of the paragraphs in the document.
Loop While(vPgf)
  // Write the next Pgf object to the Console.
  Write Console vPgf.NextPgfInDoc;
  // Move to the next paragraph in the list.
  Set vPgf = vPgf.NextPgfInDoc;
EndLoop
```



```
OBJ [Pgf] {40000D0,1F03E114}
OBJ [Pgf] {40000D0,1F03E115}
OBJ [Pgf] {40000D0,1F03E116}
OBJ [Pgf] {40000D0,1F03E117}
OBJ [Pgf] {40000D0,1F03E118}
OBJ [Pgf] {40000D0,1F03E119}
OBJ [Pgf] {40000D0,1F03E11A}
OBJ [Pgf] {40000D0,1F03E11B}
OBJ [NULL] {40000D0,0}
```

We are writing the **NextPgfInDoc** property of each **Pgf** to the Console. When the script reaches the last **Pgf** in the list, its **NextPgfInDoc** object is **0** (or **NULL**). This is what causes the **Loop While(vPgf)** loop to exit.

Note that the paragraphs in this list are *unordered*, that is, they are not necessarily in document order. Most FrameMaker linked lists are unordered, but some are ordered.

Getting the Correct Objects

One of the main challenges in writing scripts is finding the correct FrameMaker object or objects to work with. A good place to start is to find out which objects are required for a particular command or task. Then you can look up the objects in the *FrameScript Scriptwriter's Reference* and find out the best way to access them using dot-notation.

One example is the necessity to loop through all of the paragraphs in the main document flow in order. You can start by looking up the **Doc** (Document) object in the *FrameScript Scriptwriter's Reference*. You will see a **FirstPgfInDoc** property, but this is not necessarily the first paragraph in the main flow. In addition, the linked list of **Pgf** objects includes all of the paragraphs in the document, including those on master and reference pages.

As you look further, you will see the **MainFlowInDoc** property, which is a **Flow** object. To see if this will help, look up the **Flow** object. Perhaps there is an object that you can use. When you look up the **Flow** object, you find a **FirstPgfInFlow** object, which is just what you need.

Now you can assemble the object hierarchy to see if it works. First, set a variable for the active document so you can test the code. Then, add each object, using dot-notation.

```
Set vCurrentDoc = ActiveDoc;
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
```

You can test your code by using the **Display** command. For example, if your code is correct, the last line should display a **Pgf** object.

Code Listing 2-8

```
Set vCurrentDoc = ActiveDoc;
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Display vPgf;
```

If you check the **Pgf** object in the *FrameScript Scriptwriter's Reference*, you will see that there is a **NextPgfInFlow** property that you can use to move from paragraph to paragraph in the flow. This list of paragraphs in the flow is an ordered list, so now you have the basis for your loop.

Code Listing 2-9

```
Set vCurrentDoc = ActiveDoc;
// Find the first paragraph in the document's main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  // Do something here to each paragraph.
  // ...
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Here is another example. We have a series of tables in a document that are tagged with a **Price** table format.

Item No.	Description	Price	
		Wholesale	Retail
12-3875	Delphi carburetor rebuild kit	27.95	49.95
12-90225	Delphi gasket set	12.75	19.95
12-42325	Delphi carburetor float	2.35	3.95

We want to straddle the heading table cells so that the table look like this.

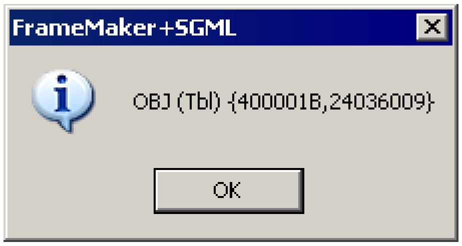
Item No.	Description	Price	
		Wholesale	Retail
12-3875	Delphi carburetor rebuild kit	27.95	49.95
12-90225	Delphi gasket set	12.75	19.95
12-42325	Delphi carburetor float	2.35	3.95

The first thing to do is look up the FrameScript command that performs the straddle, **Straddle TableCells**. This tells you that you need the **Cell** object of the first cell in the straddle. To get the **Cell** object, you will need to find the **Tbl** (table) object of the **Price** tables in the document. You should see this as two distinct tasks: one, is to loop through all of the tables and find those tagged with the **Price** table format; two is to actually perform the straddles on each table. It is best to work with each task separately and then combine them in your final script.

Let's start by figuring out how to do the straddle on a single table. A document has a **SelectedTbl** property that returns a **Tbl** object if your cursor is in a table or there are table cells selected. You can use this to test your code on the selected table in a document. To see this, click in a table and run this code.

Code Listing 2-10

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table in the document.
Set vTbl = vCurrentDoc.SelectedTbl;
// Display the table object.
Display vTbl;
```

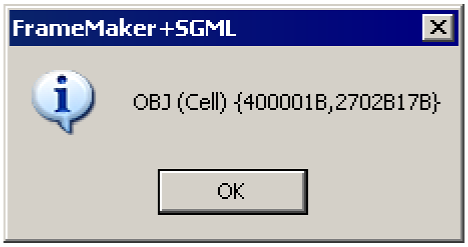


OBJ (Tbl) {400001B,24036009}

If your cursor is in a table, **vTbl** will represent the **Tbl** object. Now that we have a **Tbl** object, we need to find the appropriate **Cell** objects to use in the **Straddle TableCells** command. Look up the **Tbl** (Table) object in the *FrameScript Scriptwriter's Reference* and see if you can find any **Cell** objects. There are no **Cell** objects, but you should notice the **FirstRowInTbl** property, which represents the first row in the table. If you look up **Row** properties, you will see a **FirstCellInRow** property. Here is the code that moves from the **Tbl** object to the **Cell** object.

Code Listing 2-11

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table in the document.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for the first cell in the first row of the table.
Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
// Display the cell object.
Display vCell;
```



OBJ (Cell) {400001B,2702B17B}

Now, try straddling the first cell and see if it works.

Code Listing 2-12

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table in the document.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for the first cell in the first row of the table.
Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
// Straddle the first two rows in the first column.
Straddle TableCells CellObject(vCell) NumCols(1) NumRows(2) On;
// Update hyphenation to correctly refresh the screen.
Update DocObject(vCurrentDoc) Hyphenating;
```

The last line in the script makes sure that the document display updates correctly after the **Straddle TableCells** command. Unfortunately, this is one of those things that you may have to learn by trial and error.

Now that we know that it works for the first cell, we can move to the next cells and straddle them. Each **Cell** has a **NextCellInRow** property that you can use to move from cell to cell. Here is the completed script.

Code Listing 2-13

```
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;

// Set a variable for the first cell in the first row of the table.
Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
// Straddle the two heading rows.
Straddle TableCells CellObject(vCell) NumCols(1) NumRows(2) On;
// Move to the next cell in the row.
Set vCell = vCell.NextCellInRow;
// Straddle the two heading rows.
Straddle TableCells CellObject(vCell) NumCols(1) NumRows(2) On;
// Move to the next cell in the row.
Set vCell = vCell.NextCellInRow;
// Straddle the two columns.
Straddle TableCells CellObject(vCell) NumCols(2) NumRows(1) On;

// Refresh the document display so you can see the straddles.
Update DocObject(vCurrentDoc) Hyphenating;
```

Now we can concern ourselves with moving from table to table in the document. We want to test for **Price** tables and then perform the straddles.

Code Listing 2-14

```
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first table in the document.
Set vTbl = vCurrentDoc.FirstTblInDoc;
// Loop through the tables in the document.
Loop While(vTbl)
  // See if the current table is a Price table.
  If vTbl.TblTag = 'Price'
    // Do the straddles with our existing code.
    // Go to the first cell of the table.
    Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
    // Straddle the two heading rows.
    Straddle TableCells CellObject(vCell) NumCols(1) NumRows(2) On;
    // Move to the next cell in the row.
    Set vCell = vCell.NextCellInRow;
    // Straddle the two heading rows.
    Straddle TableCells CellObject(vCell) NumCols(1) NumRows(2) On;
    // Move to the next cell in the row.
    Set vCell = vCell.NextCellInRow;
    // Straddle the two columns.
    Straddle TableCells CellObject(vCell) NumCols(2) NumRows(1) On;
  EndIf
  // Go to the next table in the document.
  Set vTbl = vTbl.NextTblInDoc;
EndLoop

// Refresh the document display so you can see the straddles.
Update DocObject(vCurrentDoc) Hyphenating;
```

These examples illustrate how dot-notation is used to move from object to object in your scripts. As you become more familiar with FrameMaker objects, it will be easier to construct the correct dot-notation syntax.

Getting Named Objects

Some FrameMaker objects have unique names. These are called *named objects* and include **PgfFmt**s, **Color**s, **XRefFmt**s, and **VarFmt**s. These objects can be uniquely identified by their names because you cannot have two of them with the same name. For instance, it is not possible to have two paragraph formats with the same name. You can access these objects by using the **Get Object** command. Here is a script that gets the Heading1 paragraph format and the Blue color and sets the paragraph format to use the Blue color.

Code Listing 2-15

```
Set vCurrentDoc = ActiveDoc;
// Get the Heading1 paragraph format.
Get Object Type(PgfFmt) Name('Heading1') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
// Make sure the paragraph format exists.
If vPgfFmt
  // Get the Blue color object.
  Get Object Type(Color) Name('Blue') DocObject(vCurrentDoc)
    NewVar(vColor);
  // Set Heading1 paragraph format to Blue.
  Set vPgfFmt.Color = vColor;
EndIf
```

You may notice that we test to see if the Heading1 paragraph format exists. This is done because it *may not* exist, and we do not want to cause an error by setting a property on a non-existent format. We don't do this on the Blue color object, because every FrameMaker document has Blue in it.

For a complete list of object types that you can use the **Get Object** command with, see the *FrameScript Scriptwriter's Reference*.

Working With Properties

You will work with properties in your scripts by reading and writing property values. Another way of saying this is that you will *get* and *set* properties values in your scripts. In our script that straddled table cells, we used each table's **TblTag** property to see what table format was applied to the table. We wanted to restrict our changes to **Price** tables. This is an example of getting a property value and using it in a script.

Property Types

When working with a particular property, you will need to know what *data type* the property represents. The following table lists some FrameMaker property value data types.

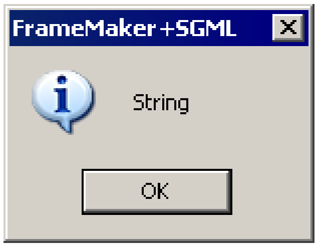
Property Data Type	Description	Example Property
Boolean	A True or False value. Can also use 1 for True and 0 for False.	Doc.ViewBorders
Integer	A whole number (negative, positive, or zero).	Tbl.TableTitlePosition
IntList	A list of whole numbers.	
Metric	A real number representing a measurement. Default unit is points.	Pgf.SpaceAbove
MetricList	A list of measurements.	Tbl.TblColWidths
String	Characters or numbers enclosed with single quotes ('). Strings are case-sensitive	PgfFmt.Name
StringList	A list of strings.	FontFamilyNames
Real	Numbers that can include decimal points.	
Object	A FrameMaker object.	Doc

There are also property types for working with text and structured elements and attributes. For information on working with text properties, See Chapter 5, "Working With Text."

It is important that you know the data types of the properties that you are working with in your scripts. You can find out the data type of a property by testing its **ObjectName** property. For example, click in a table and run this code.

Code Listing 2-16

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;
// Display the table tag's data type.
Display vTbl.TblTag.ObjectName;
```



String

This shows that the **TblTag** property is a **String** value. Strings must be enclosed in single quotes in your scripts. For instance, if you are testing for the existence of a particular table format, you must use single quotes around the name.

```
If vTbl.TblTag = 'Price' // single quotes around a string
  ...
EndIf
```

You also use the quotes when you are setting a string property value.

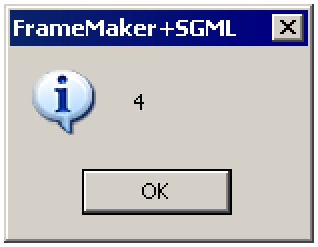
```
If vPgf.Name = 'Head1'
  If vPgf.Start = PgfTopOfPage
    Set vPgf.Name = 'Head1Top';
  EndIf
EndIf
```

Integer Constants

Some FrameMaker properties are represented by constants. A *constant* is a text string that represents an integer value. A constant is used in place of an integer because it is more descriptive and easier to remember. For example, the **PgfAlignment** property on a **Pgf** or **PgfFmt** is an integer value. If you look in the *FrameScript Scriptwriter's Reference*, you will see constants for each value (**PgfLeft**, **PgfCenter**, **PgfRight**, **PgfJustified**). However, when you get the property using FrameScript, it will display the integer, not the constant. Here is a script that illustrates this.

Code Listing 2-17

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
// If the paragraph is left-aligned, justify it.
If vPgf.PgfAlignment = PgfLeft
  Set vPgf.PgfAlignment = PgfJustified;
  Display vPgf.PgfAlignment; // Shows an integer, not a constant
EndIf
```



4

IMPORTANT: When you use a constant in your code, you *do not* enclose it in single quotes.

3 Control Structures

By default, FrameScript commands execute from top to bottom in a script. You can control the flow of your scripts by using *control structures*. Control structures include **If**/**ElseIf**/**Else**/**EndIf** statements that can determine if and when certain commands are executed. **Loop**/**EndLoop** statements can be used to perform the same command on a list of objects. Blocks of related commands can be put in *subroutines* by using **Sub**/**EndSub** statements. Subroutines can be called from other parts of your script, and can be reused in other scripts.

IMPORTANT: Unlike FrameScript commands, FrameScript control statements do not end with a semicolon.

If/ElseIf/Else/EndIf

The simplest control structure is **If**/**EndIf**. We can verbalize it by saying, "If this condition occurs, then do this." The optional **Else** statement expands this by saying, "If this condition occurs, then do this; or else, do that." FrameScript 3 adds the optional **ElseIf** statement for testing multiple conditions. The **EndIf** statement is always required.

A common example is a test that occurs at the beginning of many scripts—the test for an active document. If a script requires an active document (an open document that has input focus), we can use an **If**/**EndIf** structure to see if an active document exists.

Code Listing 3-1

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf
```

In reality, we are testing for a *negative* condition—if an active document *doesn't* exist, we display a message, and exit the script. If an active document does exist, the commands inside the **If**/**EndIf** structure never get executed, and the script continues past it.

We can use the optional **Else** statement to do something if the condition is opposite to that which is tested for in the **If** statement. Here, the previous example is expanded to treat the opposite condition.

Code Listing 3-2

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the active document.
  Set oDoc = ActiveDoc;
EndIf
```

You can also reverse the order of the test. The following code is equivalent to the previous.

Code Listing 3-3

```
// Test for an active document.
If ActiveDoc = 1
  // Set a variable for the active document.
  Set oDoc = ActiveDoc;
Else
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf
```

What exactly are you testing for with the **If** statement? Many times, you will be testing to see if a condition is True or False. You can use the keywords **True** and **False**, or you can use **1** and **0** like we did in the examples. To be more precise, our example code is testing to see if an object (the **ActiveDoc**) *exists*. If it exists, it is **True** (**1**); if it doesn't exist, it is **False** (**0**). In FrameScript, anything that isn't **False** is considered to be **True**, so you can leave the **= True** or **= 1** condition off of your test. In the following example, we are testing to see if the Heading 1 paragraph format exists in the document.

Code Listing 3-4

```
// Set a variable for the active document.
Set oDoc = ActiveDoc;
// Get the paragraph format object.
Get Object Type(PgfFmt) Name('Heading 1') DocObject(oDoc)
  NewVar(oPgfFmt);
If oPgfFmt
  MsgBox 'The format exists.    ';
EndIf
```

Notice that we simply used **If oPgf**, which is equivalent to **If oPgf = True** and **If oPgf = 1**. When testing for existence, the scripts in this publication use the short form of the test, as in the previous example.

It is important to distinguish a test for existence from the test for a particular value. If you are testing a value, the **= 1** condition may be necessary on the **If** statement. For example, the following script reports the alignment of the paragraph containing the insertion point.

Code Listing 3-5

```
// Set a variable for the paragraph at the insertion point.
Set oPgf = TextSelection.Begin.Object;
// Test the paragraph's alignment.
If oPgf.PgfAlignment = 1
  MsgBox 'The paragraph is left-aligned.    ';
Else
  If oPgf.PgfAlignment = 2
    MsgBox 'The paragraph is right-aligned.    ';
  Else
    If oPgf.PgfAlignment = 3
      MsgBox 'The paragraph is center-aligned.    ';
    Else
      MsgBox 'The paragraph is justified.    ';
    EndIf
  EndIf
EndIf
```

We are not testing for **True** or **False**, or for the existence of an object. We are testing for specific values. To illustrate the difference, remove **= 1** from the first **If** statement and try the code on a paragraph that is not left aligned. Regardless of the paragraph's alignment, it will be reported as left-aligned. In effect, the test now says, "If **oPgf.PgfAlignment** is any non-zero value, it is **True**." Any of the possible **oPgf.PgfAlignment** values will fulfill this because they are all non-zero values; as a result, the first **If** test will always be **True**.

As you can see from the above script, **If**/**Else**/**EndIf** statements can be nested to handle more than two conditions; in this case, four conditions. However, nested statements can become unwieldy and hard to follow. FrameScript 3 adds the optional **ElseIf** statement that makes it easier to work with multiple conditions. Here is the previous code using the **ElseIf** statement.

Code Listing 3-6

```
// Set a variable for the paragraph at the insertion point.
Set oPgf = TextSelection.Begin.Object;
// Test the paragraph's alignment.
If oPgf.PgfAlignment = 1
  MsgBox 'The paragraph is left-aligned.    ';
ElseIf oPgf.PgfAlignment = 2
  MsgBox 'The paragraph is right-aligned.    ';
ElseIf oPgf.PgfAlignment = 3
  MsgBox 'The paragraph is center-aligned.    ';
Else
  MsgBox 'The paragraph is justified.    ';
EndIf
```

For another method of testing multiple conditions, See "Using a Loop to Handle Multiple Conditions" on page 3-13.

Loop/EndLoop

Loops are useful for processing a list of items, and for executing commands a certain number of times. Because many FrameMaker objects are contained in lists, you will use loops quite often in your scripts. FrameScript has several loop structures to choose from.

Loop ForEach

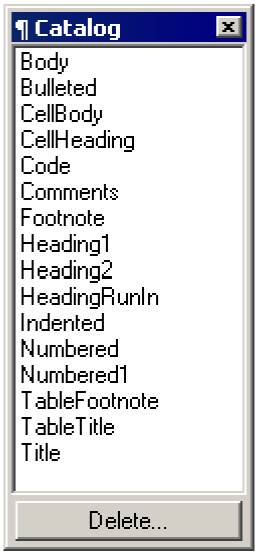
This is the simplest of the loop structures and is designed to move through a predefined list of objects. FrameMaker documents contain internal lists of many different kinds of objects, such as paragraphs, markers, and paragraph formats. For example, the following code loops through each of the paragraph formats in the active document.

Code Listing 3-7

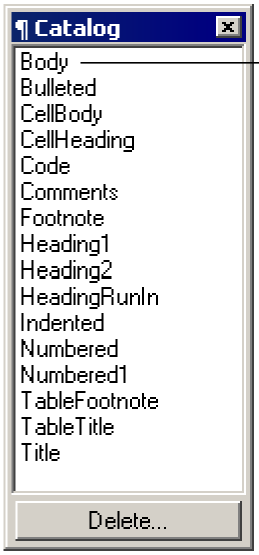
```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the active document.
Set oDoc = ActiveDoc;

// Loop through the paragraph formats in the document.
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt)
  // Write the paragraph format name to the Console.
  Write Console oPgfFmt.Name;
EndLoop
```

All three parameters on the loop are required. **ForEach** is the type of object that you want to loop through; **In** is the object that contains the list of **ForEach** objects; and **LoopVar** is a variable that the current member of the loop is assigned to.

To understand this code, let’s try to visualize it by using the Paragraph Catalog. The **oDoc** object contains a *list* of paragraph formats, and the **ForEach** loop is designed to process a list of objects.

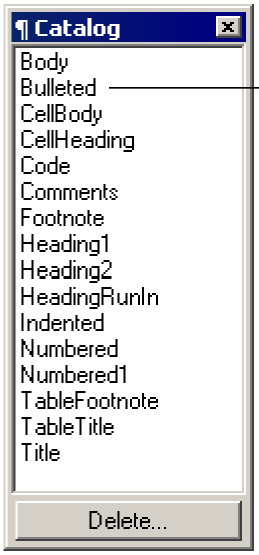


When the loop is first entered, the first member of the list is assigned to the **LoopVar** variable.



oPgfFmt = Body paragraph format object

Each member of the list is assigned to the **LoopVar** variable as the loop processes.



oPgfFmt = Bulleted paragraph format object

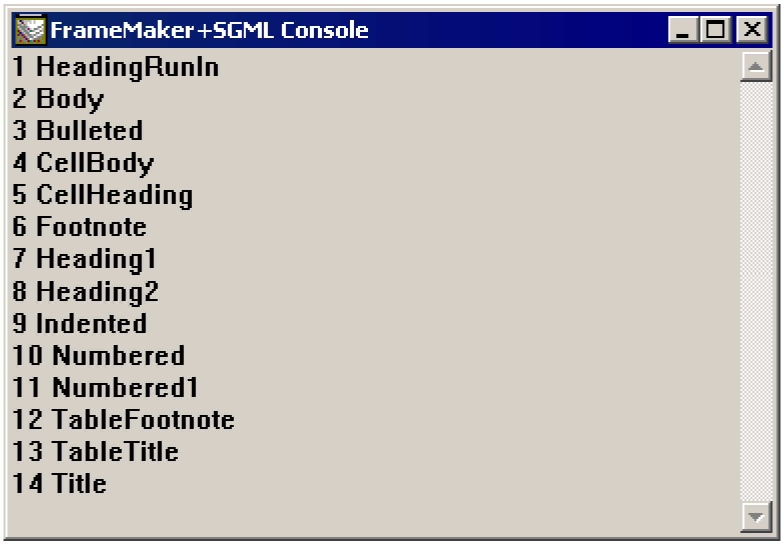
The loop continues moving down the list of objects until it reaches the end of the list. The loop then exits and the script continues with the first command (if any) after the **EndLoop** statement.

Another way to visualize this is to add a counter to the loop. Make a new FrameMaker document and run the following code.

Code Listing 3-8

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the active document.
Set oDoc = ActiveDoc;
```

```
// Initialize a counter.
Set i = 0;
// Loop through the paragraph formats in the document.
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt)
  // Increment the counter.
  Set i = i + 1;
  // Write the counter and the paragraph format name to the Console.
  Write Console i+' '+oPgfFmt.Name;
EndLoop
```



```
1 HeadingRunIn
2 Body
3 Bulleted
4 CellBody
5 CellHeading
6 Footnote
7 Heading1
8 Heading2
9 Indented
10 Numbered
11 Numbered1
12 TableFootnote
13 TableTitle
14 Title
```

Look at the FrameMaker Console window and you will see a number in front of paragraph format name. The last number tells you how many paragraph formats are in the document's list. It also tells you how many times the code inside the loop was executed—once for each member in the list.

Note that the list of paragraph formats in the document is not necessarily in the same order as you see them displayed in the Paragraph Catalog. Most object lists are in not in any particular order. For example, the **Pgf** (paragraph) objects in a document's list are not necessarily in the same order that they appear in the document. We will discuss how to process paragraphs in document order in a later section.

There are situations when when this kind of loop will not work. Not all FrameMaker objects can be used in the **ForEach** loop. For instance, you cannot use **Loop ForEach** to loop through all of the **Graphic** objects in a document. In addition, the **Delete** command will not work correctly inside the loop. Here is an example: make a new FrameMaker document, display the Paragraph Catalog, and run the following script.

Code Listing 3-9

```
Set oDoc = ActiveDoc;

Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt)
  // Delete the current paragraph format.
  Delete Object(oPgfFmt);
EndLoop
```

We would expect the script to delete all of the paragraph formats in the document, but it doesn't. Each time you run the script, it deletes one format and stops. To understand why this happens, let's introduce a different kind of loop and examine how it works.

Loop While

The commands inside a **Loop While** loop will be executed as long as the **While** condition is true. The basic syntax is

```
Loop While(Condition = True)
  // Commands inside loop.
  // ...
EndLoop
```

The **Loop ForEach** loop is simply a shorthand version of the **Loop While** loop. We can see this by comparing the two. Here is a **Loop ForEach** loop, followed by the equivalent **Loop While** loop.

Code Listing 3-10

```
// Loop ForEach loop.
Set oDoc = ActiveDoc;
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt)
  Write Console oPgfFmt.Name;
EndLoop

// Loop While loop.
Set oDoc = ActiveDoc;
Set oPgfFmt = oDoc.FirstPgfFmtInDoc;
Loop While(oPgfFmt)
  Write Console oPgfFmt.Name;
  Set oPgfFmt = oPgfFmt.NextPgfFmtInDoc;
EndLoop
```

The scripts perform the same tasks, but if you look closely, you will see the subtle differences in the code. Here is an outline of what the second loop does.

```
Set oPgfFmt = oDoc.FirstPgfFmtInDoc;
```

1. This line sets the **oPgfFmt** variable equal to the first paragraph format in the document's list of **PgfFmt** objects. This is represented by the document's **FirstPgfFmtInDoc** property.

```
Loop While(oPgfFmt)
```

2. This line begins a loop that will continue as long as **oPgfFmt** is **True**. In other words, the commands inside the loop will execute as long as **oPgfFmt** *exists*.

```
Write Console oPgfFmt.Name;
```

3. Inside the loop, this command will write the current paragraph format's name to the Console window.

```
Set oPgfFmt = oPgfFmt.NextPgfFmtInDoc;
```

4. *This line is the key.* The value of **oPgfFmt** is changed to the next paragraph format in the document. If there are no more formats in the document, **oPgfFmt** will become **0** (zero) and the loop will exit.

While the syntax of the loops is different, it is important to understand that the two loops operate the same. Steps 1 and 4 are simply hidden in the **ForEach** loop.

The basic purpose of these loops is to perform a task for each member of a list of objects (paragraphs, markers, paragraph formats, etc.). You go from one member of the list to the next, performing the task on each member, until you reach the bottom of the list. The "task" is performed inside the loop. To understand how the loop moves down the list, you need to understand the FrameMaker concept of *linked lists*.

Linked Lists

FrameMaker maintains separate *linked lists* of many different kinds of objects. For example there is a list of open documents when FrameMaker is running. Each document has linked lists of other objects, such as character formats, graphics, and tables. Here is the key to understanding linked lists: *only the first member is exposed to objects outside the list*. To see this, look back at the previous code. We see that there is a **FirstPgfFmtInDoc** property of a document, which gives us the first member of the linked list of paragraph formats. But what if we want the second paragraph format in the document? Can we use this code?

```
Set oPgfFmt = oDoc.SecondPgfFmtInDoc;
```

This won't work because the document object only gives access to the *first* member of the list. It says, in effect, "I'll get you to the first member of the list, but after that, you are on your own."

The responsibility of moving from member-to-member in a list falls on each of the list's members. Each **PgfFmt** object has a **NextPgfInFmt** property that points to the next member in the list. Some properties have **Prev-** properties that point to the previous member of the list; for instance, each **Pgf** object has a **PrevPgfInDoc**, as well as a **NextPgfInDoc** property. The document object "passes the baton" to the first member of the list, and the first member passes it to the next, etc. This passing from member-to-member is what the following line does inside the loop.

```
Set oPgfFmt = oPgfFmt.NextPgfFmtInDoc;
```

When the script gets to the last member in the list, its **NextPgfFmtInDoc** property returns **0** (zero) because the **NextPgfFmtInDoc** object does not exist. The causes the **oPgfFmt** variable to be **False** and the loop exits.

The **ForEach** script uses the same method, but the "baton passing" from member to member is hidden.

Let's reproduce the script to delete paragraphs, this time using the **Loop While** form.

Code Listing 3-11

```
Set oDoc = ActiveDoc;

Set oPgfFmt = oDoc.FirstPgfFmtInDoc;
Loop While(oPgfFmt)
  Delete Object(oPgfFmt);
  Set oPgfFmt = oPgfFmt.NextPgfFmtInDoc;
EndLoop
```

Run the script and see what happens. Like the **ForEach** loop, it only deletes one format, and this time, it gives an error. Can you figure out why? If we use the runner with the baton metaphor, the answer is simple. This line

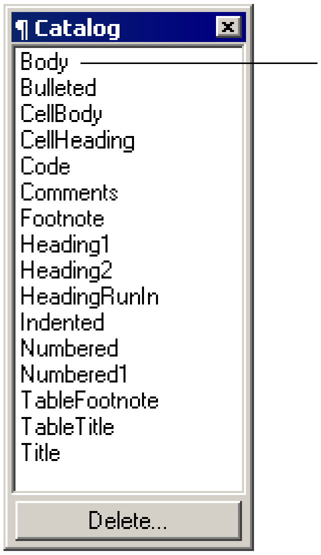
```
Delete Object(oPgfFmt)
```

deletes the **oPgfFmt** object before it has a chance to "pass the baton" to the next paragraph format in the document. To put it bluntly, we killed the runner before he could pass the baton! The next line in the script gives an error because the **oPgfFmt** object no longer exist.

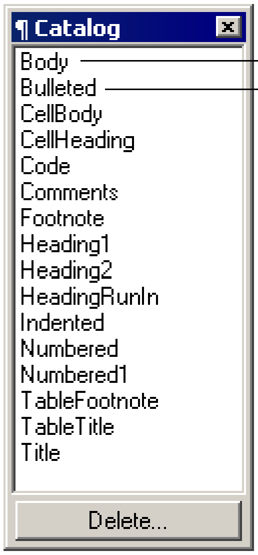
Here is the solution followed by a visual explanation.

Code Listing 3-12

```
Set oDoc = ActiveDoc;

Set oPgfFmt = oDoc.FirstPgfFmtInDoc;
Loop While(oPgfFmt)
  // "Pass the baton" to the next format using a temporary variable.
  Set oNextPgfFmt = oPgfFmt.NextPgfFmtInDoc;
  // Delete the paragraph format.
  Delete Object(oPgfFmt);
  // Set the variable to the temporary variable.
  Set oPgfFmt = oNextPgfFmt;
EndLoop
```

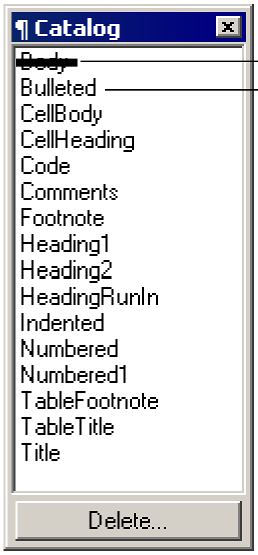


```
Set oPgfFmt = oDoc.FirstPgfFmtInDoc;
```



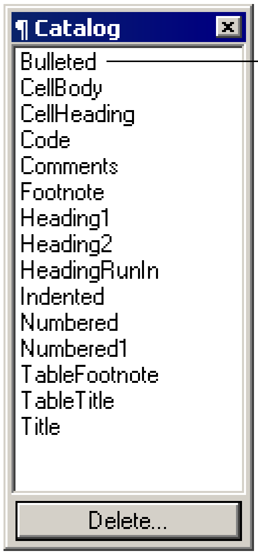
oPgfFmt

```
Set oNextPgfFmt = oPgfFmt.NextPgfInDoc;
```



```
Delete Object(oPgfFmt);
```

oNextPgfFmt



```
Set oPgfFmt = oNextPgfFmt;
```

Give the script a try on your temporary document and you will see that all of the paragraph formats are deleted.

LoopVar Parameter

You can use the **LoopVar** parameter with **Loop While** loops to run the commands inside a loop a particular number of times. When you use the **LoopVar** parameter, you must also use the **Init** and **Incr** paremeters. The **LoopVar** parameter designates a counter variable that is incremented every time the loop processes. **Init** is the value that you want the loop to start with and **Incr** is how many units the counter is incremented by for each pass through the loop. Here is a simple example.

Code Listing 3-13

```
// Set a variable to stop the loop.
Set iStop = 10;
// Loop while the counter is less than or equal to the stop value.
Loop While(i <= vStop) LoopVar(i) Init(1) Incr(2)
  // Write the counter to the Console window.
  Write Console i;
EndLoop
```

Run the script and take a look at the Console window. Change the **Init** value to **3** and run the script again. This time the loop will begin with **i = 3**. Change **Init** back to **1** and **Incr** to **2**, and rerun the script. This time, the script counts by twos.

You can also run the loop backwards from the maximum value to the minimum.

Code Listing 3-14

```
Set iStop = 10;
// Loop while the counter is greater than zero.
Loop While(i > 0) LoopVar(i) Init(vStop) Incr(-1)
  // Write the counter to the Console window.
  Write Console i;
EndLoop
```

The **-1** on the **Incr** parameter causes the counter to increment backwards. It starts at **10** (the value of **vStop**), and runs while the **LoopVar** is greater than zero.

In many scripts you will use list variables, such as string lists (**StringList**), integer lists (**IntList**), or text lists (**TextList**). List variables have a **Count** property that tells how many members are in the list. You can use this with a **Loop While** structure to loop through all of the members in the list. For example, you can use the following script to get a text list of markers in the paragraph containing the insertion point. Then the loop writes the text of each marker to the Console window. Add a few markers to a paragraph, click in the paragraph, and try the script.

Code Listing 3-15

```
// Set a variable for the paragraph containing the insertion point.
Set oPgf = TextSelection.Begin.Object;
// Get a list of markers in the paragraph.
Get TextList InObject(oPgf) MarkerAnchor NewVar(tTextList);
// Loop through the list of markers.
Loop While(i <= tTextList.Count) LoopVar(i) Init(1) Incr(1)
  // Get each member of the list.
  Get Member Number(i) From(tTextList) NewVar(tMarkerAnchor);
  // Write the marker text to the Console window.
  Write Console tMarkerAnchor.TextData.MarkerText;
EndLoop
```

The beauty of this code is that it will work regardless of the number of markers in the paragraph. If there are no markers, the loop will not begin because the condition **i <= tTextList.Count** (1 <= 0) will not be True.

Loop Until

This version of the loop is essentially a mirror image of the **Loop While** loop; it loops *until* the condition is fulfilled. For example, you can use a **Loop Until** loop to repeatedly display a dialog box until the user chooses or enters an item. This prevents them from dismissing an empty dialog box. Here is a script that displays a scrollbox dialog of color choices. The OK button will not dismiss the dialog box unless a color is chosen.

Code Listing 3-16

```
// Make a string list of colors.
New StringList NewVar(sColors);
Add Member('Red') To(sColors);
Add Member('Blue') To(sColors);
Add Member('Yellow') To(sColors);
Add Member('Green') To(sColors);

// Sort the stringlist.
Sort List(sColors);

// Initialize a variable for the position in the list of the choice.
Set iIndex = 0;
// Loop until an item is chosen in the scrollbox.
Loop Until(iIndex > 0)
  // Display a scrollbox dialog box with the colors.
  DialogBox Type(ScrollBox) List(sColors) Caption('Choose a Color')
    Title('Please select a color from the list:')
    Button(bButton) NewVar(sColor) ReturnIndex(iIndex);
  If bButton = CancelButton
    LeaveLoop;
  EndIf
EndLoop
```

The **ReturnIndex** parameter on the **ScrollBox DialogBox** returns the position in the list of the user's choice. If the user clicks OK without selecting a color, **iIndex** remains zero and the loop continues. The loop will continue until **iIndex > 0**, that is, when the user makes a choice. The only other way out of the loop is by the user cancelling the dialog box. This invokes the **LeaveLoop** command, which causes the code to exit the loop.

In practice, there is little difference between the **Loop While** and **Loop Until** structures. They can usually be interchanged by simply changing the condition. Here is the previous script modified to use the **Loop While** statement. Notice that the condition has been changed.

Code Listing 3-17

```
// Make a string list of colors.
New StringList NewVar(sColors);
Add Member('Red') To(sColors);
Add Member('Blue') To(sColors);
Add Member('Yellow') To(sColors);
Add Member('Green') To(sColors);

// Sort the stringlist.
Sort List(sColors);

// Initialize a variable for the position in the list of the choice.
Set iIndex = 0;
// Loop while there is no selection in the scrollbox.
Loop While(iIndex = 0)
  // Display a scrollbox dialog box with the colors.
  DialogBox Type(ScrollBox) List(sColors) Caption('Choose a Color')
    Title('Please select a color from the list:')
    Button(bButton) NewVar(sColor) ReturnIndex(iIndex);
  If vButton = CancelButton
    LeaveLoop;
  EndIf
EndLoop
```

LeaveLoop

The **LeaveLoop** command simply exits the loop, causing the script to continue with any commands after the **EndLoop** statement. It is typically used to exit a loop before the loop condition is fulfilled. In the previous script, **LeaveLoop** was used to exit the loop if the user cancelled the dialog box. The next section shows another example of the **LeaveLoop** command.

Using a Loop to Handle Multiple Conditions

In an earlier section, you saw how to test for multiple conditions by using the **If/ElseIf/Else/EndIf** structure. There is another way that uses a loop to avoid nested conditions. It is similar to the Case statement that some other programs employ. We will use the same example that we used before—this script tests the alignment of the paragraph containing the insertion point.

Code Listing 3-18

```
// Set a variable for the paragraph at the insertion point.
Set oPgf = TextSelection.Begin.Object;
```

```
// Make a loop that is always true.
Loop While(1)
  // Test the paragraph’s alignment.
  If oPgf.PgfAlignment = 1
    MsgBox 'The paragraph is left-aligned.    ';
    LeaveLoop; // Exit the loop.
  EndIf
  If oPgf.PgfAlignment = 2
    MsgBox 'The paragraph is right-aligned.    ';
    LeaveLoop; // Exit the loop.
  EndIf
  If oPgf.PgfAlignment = 3
    MsgBox 'The paragraph is center-aligned.    ';
    LeaveLoop; // Exit the loop.
  EndIf
  MsgBox 'The paragraph is justified.    ';
  LeaveLoop; // Exit the loop.
EndLoop
```

The script will always enter the loop, because of the **1** on the **While** condition. Each condition is tested one-by-one by **If** statements, and as soon as the correct value is encountered, the code inside the condition is executed. The first line displays the message box, and the **LeaveLoop** command on the second line causes the script to exit the loop. If none of the three conditions are true, the last message box will be displayed, and the last **LeaveLoop** command will cause the script to exit the loop.

It is critical that you provide a **LeaveLoop** statement outside of any of the conditions, in case none of them are True. Otherwise, the script will get stuck in an endless loop. If this should occur, you can try hitting the Escape key on the PC or Command+. on the Mac to abort the script.

Because of the **ElseIf** option that was added to the FrameScript 3 **If**/**Else**/**EndIf** structure, this way of handling multiple conditions is largely unnecessary.

Sub/EndSub

Subroutines are blocks of code bordered by the **Sub**/**EndSub** statements. The purpose of a subroutine is to isolate code that performs a specific task. The subroutine can then be called from anywhere else in the script where the task needs to be performed. Subroutines can also be called from other scripts.

A script can contain as many subroutine as you want, but they must always go at the end of a script, after the *body* of the script. The body of a script consists of any command that are *not* inside a subroutine.

Here is a simple script that turns off hyphenation for all paragraph formats and paragraphs in the active document.

Code Listing 3-19

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set oDoc = ActiveDoc;
EndIf
```

```
// Turn off the document display to prevent flicker.
Set Displaying = 0;

// Turn off hyphenation for each paragraph format.
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt);
  Set oPgfFmt.Hyphenate = 0;
EndLoop

// Turn off hyphenation for each paragraph in the document.
Loop ForEach(Pgf) In(oDoc) LoopVar(oPgf)
  Set oPgf.Hyphenate = 0;
EndLoop

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(oDoc) Redisplay;
```

We can make a subroutine from the code to turn off hyphenation and then call it from the body of the script.

Code Listing 3-20

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set oDoc = ActiveDoc;
EndIf

// Turn off the document display to prevent flicker.
Set Displaying = 0;

// Call the subroutine to turn off hyphenation.
Run SetHyphenation;

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(oDoc) Redisplay;

Sub SetHyphenation
//
// Turn off hyphenation for each paragraph format.
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt);
  Set oPgfFmt.Hyphenate = 0;
EndLoop

// Turn off hyphenation for each paragraph in the document.
Loop ForEach(Pgf) In(oDoc) LoopVar(oPgf)
  Set oPgf.Hyphenate = 0;
EndLoop
//
EndSub
```

As you can see, we isolate the hyphenation code and put it in a subroutine called **SetHyphenation**. Then we use the **Run** command to call the subroutine, which is placed after the body of the script. This example also shows how subroutines alter the top-to-bottom flow of a script. The script runs from the top until it hits the subroutine call at the line

```
Run SetHyphenation;
```

The script *branches* to the subroutine and runs all of its commands, before returning to the commands after the subroutine call.

In a simple script like this, it may not seem practical to use a subroutine. However, by doing so, you have a block of useful code that can easily be copied and used in another script. The subroutine performs a useful task that is identified by the name of the subroutine.

Using subroutines can also make the body of your scripts easier to follow. Here is the body of a script that shows a series of subroutine calls for an active document. (The subroutines are left off for brevity).

Code Listing 3-21

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set oDoc = ActiveDoc;
EndIf

// Turn off the document display to prevent flicker.
Set Displaying = 0;

// Run a subroutine to turn off hyphenation.
Run SetHyphenation;

// Run a subroutine to change view settings.
Run ChangeViewSettings;

// Run a subroutine to change save the document as text only.
Run SaveDocAsText;

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(oDoc) Redisplay;
```

You can see what the script does to the document, simply by looking at the subroutine calls. The script's purpose is evident, while the implementation details are "hidden" in the individual subroutines.

Using Parameters With Subroutines

Parameters are variables that you can "pass" to a subroutine when you call it with the **Run** command. Parameters allow you to make your subroutines general enough to work with any values. For example, here is a subroutine that calculates the square root of a number. Don't worry about the mechanics of the subroutine; we have purposely picked one that is a little complex. A subroutine is sometimes like a "black box"; you don't need to know how it works, you just need to know what values to pass to it.

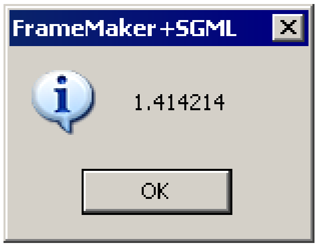
Code Listing 3-22

```
Sub CalculateSquareRoot
//
New Real NewVar(vValue) Value(vValue);
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

Since we want to be able to use this subroutine to find the square root of *any* number, it has to be able to accept any number that we pass to it. The variable **vValue** is used to accept the number we pass to it, and the variable **vSquareRoot** is the answer that the subroutine generates. Here is a sample call to the subroutine.

Code Listing 3-23

```
// Find the square root of a number.
Run CalculateSquareRoot vValue(2);

// Display the answer.
Display vSquareRoot;

Sub CalculateSquareRoot
//
New Real NewVar(vValue) Value(vValue);
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```



1.414214

In the **Run** command, we pass the value **2** to the subroutine by assigning it to **vValue**. You can also pass property values or the results of calculations to a subroutine. For example, suppose you want to calculate the length of a selected line. Here is a way to do a calculation with the line's height and width and pass the result to the subroutine.

Code Listing 3-24

```
// Set a variable for the selected line.
Set vLine = FirstSelectedGraphicInDoc;

// Run the subroutine with the appropriate values.
Run CalculateSquareRoot
  vValue((vLine.Height * vLine.Height) +
  (vLine.Width * vLine.Width));

// Display the answer.
Display vSquareRoot;

Sub CalculateSquareRoot
//
New Real NewVar(vValue) Value(vValue);
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

You can also return values from a subroutine by using the **Returns** keyword on the **Run** command. Here is the previous code returning the square root in a variable called **vLength**. We also simplified the parameter passing by doing the **vLine.Height** and **vLine.Width** calculations before the **Run** command.

Code Listing 3-25

```
// Set a variable for the selected line.
Set vLine = FirstSelectedGraphicInDoc;
```

```
// Square the line height and width and add them together.
Set vSquared = (vLine.Height * vLine.Height) +
  (vLine.Width * vLine.Width);

// Run the subroutine with the appropriate values.
Run CalculateSquareRoot vValue(vSquared)
  Returns vSquareRoot(vLength);

// Display the answer.
Display vLength;

Sub CalculateSquareRoot
//
New Real NewVar(vValue) Value(vValue);
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

Notice now that we display the **vLength** return value instead of the **vSquareRoot** value from the subroutine.

Parameters can also be used to make a subroutine do something different, based on the value of the parameter. This is valuable because it makes the subroutine more general, and thus useful in more situtations. Here is the **SetHyphenation** subroutine modifed to turn hyphenation on or off, depending on the value that is passed to **vSetting**.

Code Listing 3-26

```
Set oDoc = ActiveDoc;

// Set the hyphenation.
Run SetHyphenation vSetting(1); // Use 0 to turn it off.

Sub SetHyphenation
//
// Turn off hyphenation for each paragraph format.
Loop ForEach(PgfFmt) In(oDoc) LoopVar(oPgfFmt);
  Set oPgfFmt.Hyphenate = vSetting;
EndLoop
```

```
// Turn off hyphenation for each paragraph.
Loop ForEach(Pgf) In(oDoc) LoopVar(oPgf)
  Set oPgf.Hyphenate = vSetting;
EndLoop
//
EndSub
```

Making Subroutines Independent

Earlier, we mentioned the *black box* approach to subroutines. This means that once a subroutine is developed, you should be able to plug it in to a script, and not have to remember how it works. You simply need to know what values to pass to it, and what values it returns. To facilitate this, you should document the subroutine, so you know how to use it. Here is our **CalculateSquareRoot** subroutine with some comments telling how to use it.

Code Listing 3-27

```
Sub CalculateSquareRoot
//
// vValue is the number that you want to find the square root of.
// vSquareRoot returns the square root of vValue.
//
New Real NewVar(vValue) Value(vValue);
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

Error checking in your subroutines will also help make them independent. Error checking handles situations where bad values might be passed to the subroutine. The **CalculateSquareRoot** subroutine should only accept values greater than zero. Here is the code with a test to make sure that vValue is greater than zero.

Code Listing 3-28

```
Sub CalculateSquareRoot
//
// vValue is the number that you want to find the square root of.
// vValue must be a number greater than zero.
// vSquareRoot returns the square root of vValue.
//
New Real NewVar(vValue) Value(vValue);
// Test for correct value range.
If vValue <= 0
  LeaveSub; // Exit the subroutine.
EndIf
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

If **vValue** is not greater than zero, **LeaveSub** will cause the subroutine to be exited without performing the calculation. This is good, because it will prevent an error in the subroutine. However, we should provide a way to report the error to the user. We could put an error message right in the subroutine like this.

```
If vValue <= 0
  MsgBox 'The value must be greater than zero.    ';
  LeaveSub; // Exit the subroutine.
EndIf
```

The problem with this is that the script will stop and wait for the user to dismiss the dialog box. Furthermore, the script will then keep running after the **Run** command that called the subroutine; this may cause problems, because the subroutine did not return an expected value. Neither of these occurences may be appropriate in the context of the entire script. As a result, it is best to report the error by setting an error code inside the subroutine, and leave error handling outside of the subroutine.

FrameScript has a variable called **ErrorCode** that is used to indicate error conditions. There is a list of possible error codes and their meanings in the *FrameScript Scriptwriter's Reference*. You can use **ErrorCode** to indicate an error in your subroutine by setting it to a non-zero value. Here are the general steps to use.

1. Before the **Run** command to call the subroutine, use **Set ErrorCode = 0;** to initialize it.
2. If there is an error in the subroutine, set **ErrorCode** to a non-zero value, for example **-1**.
3. After the Run command, test the value of **ErrorCode**; if it is not equal to zero, handle the error outside of the subroutine as appropriate.

Here is an example script, using the **CalculateSquareRoot** subroutine.

Code Listing 3-29

```
// Initialize ErrorCode to zero.
Set ErrorCode = 0;
// Call the subroutine.
Run CalculateSquareRoot vValue(-25) Returns vSquareRoot(vLength);
// Test the value of ErrorCode.
If ErrorCode not= 0
  // Display a message to the user.
  MsgBox 'The square root could not be calculated.    ' Mode(Warn);
  LeaveSub; // Exit the script.
EndIf

Sub CalculateSquareRoot
//
// vValue is the number that you want to find the square root of.
// vValue must be a number greater than zero; if not ErrorCode
//   will be set to -1.
// vSquareRoot returns the square root of vValue.
//
New Real NewVar(vValue) Value(vValue);
// Test for correct value range.
If vValue <= 0
  // Set ErrorCode to -1.
  Set ErrorCode = -1;
  LeaveSub; // Exit the subroutine.
EndIf
Set vTolerance = .000005;
Set vOldApp = vValue;
Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
Set vTest = vValue;
Loop While(vTest > vTolerance)
  Set vOldApp = vSquareRoot;
  Set vSquareRoot = (vOldApp + vValue/vOldApp)/2;
  Set vTest = (vSquareRoot - vOldApp)/vSquareRoot;
  If vTest < 0
    Set vTest = (vTest * (-1));
  EndIf
EndLoop
//
EndSub
```

In this case, we are assuming that an error condition is critical, so we exit the script when there is an error. In other cases, an error may not be critical enough to stop the rest of the script. Here is an example where we simply warn the user with a message to the Console and continue the script with a default value. The subroutine is left out for brevity.

Code Listing 3-30

```
// Initialize ErrorCode to zero.
Set ErrorCode = 0;
// Call the subroutine.
Run CalculateSquareRoot vValue(-25) Returns vSquareRoot(vLength);
// Test the value of ErrorCode.
If ErrorCode not= 0
  // Write a message to the Console windo.
  Write Console 'The square root could not be calculated.';
  Write Console 'Default value of 5 was used.';
  // Set the default value.
  Set vLength = 5;
EndIf
```

To summarize, you should set the **ErrorCode** variable to a non-zero condition (inside the subroutine) if an error occurs, but you should leave the specific error handling outside the subroutine.

LeaveSub

You have already seen the **LeaveSub** command in action; it exits a subroutine when it is encountered inside the subroutine. If **LeaveSub** occurs in the body of the script outside of any subroutines, it exits the entire script.

Be sure that you "clean up" before using the **LeaveSub** command, especially in the body of the script. For example, in the following incomplete code, an error forces the script to exit before it finishes. We turned off the document display at the beginning of the script to prevent screen flicker and to make the script run faster. If the script exits before the end, the **Displaying** property will never be turned back on, the the screen will be left blank.

```
Set Displaying = 0;

...

Set ErrorCode = 0;
Run MySubroutine;
If ErrorCode not= 0
  LeaveSub;
EndIf

...

Set Displaying = 1;
Update DocObject(oDoc) Redisplay;
```

The solution is to restore the appropriate properties before using the **LeaveSub** command.

```
Set Displaying = 0;

...
```

```
Set ErrorCode = 0;
Run MySubroutine;
If ErrorCode not= 0
  // Restore the display before leaving the script.
  Set Displaying = 1;
  LeaveSub;
EndIf

...

Set Displaying = 1;
Update DocObject(oDoc) Redisplay
```

If you are left with a blank screen after a script error, open the Script Window, and run the following code.

```
Set Displaying = 1;
```

4 Documents and Books

FrameScript has a multitude of commands and options for working with documents and books. Many of them duplicate functions available in the FrameMaker interface, but some of the options can only be accessed through FrameScript commands.

Opening Documents and Books

Documents

You use the **Open Document** command to open existing FrameMaker documents. The **File** parameter specifies the absolute path to the file that you want to open. The **NewVar** parameter returns the document object variable of the opened document. Here is a simple example.

Code Listing 4-1

```
// Set a variable for the full path name of the file.
Set vDocName = 'G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Training'+
  DIRSEP+'FrameScriptCrashCourse'+DIRSEP+'WorkingWithText.fm';

// Open the document.
Open Document File(vDocName) NewVar(vCurrentDoc);
```

The **DIRSEP** variable gives the correct folder delimiter, based on what platform the script is running on. On Windows it is the backslash (\) character; on the Mac it is a colon (:). If you do not need your scripts to work across platforms, you can hard-code the delimiter into the paths.

```
// Windows path name.
Set vDocName = 'G:\FrameScript2_1\Training\FrameScriptCrashCourse'+
  '\WorkingWithText.fm';

// Macintosh path name.
Set vDocName = 'My Drive:FrameScript2_1:Training:'+
  'FrameScriptCrashCourse:WorkingWithText.fm';
```

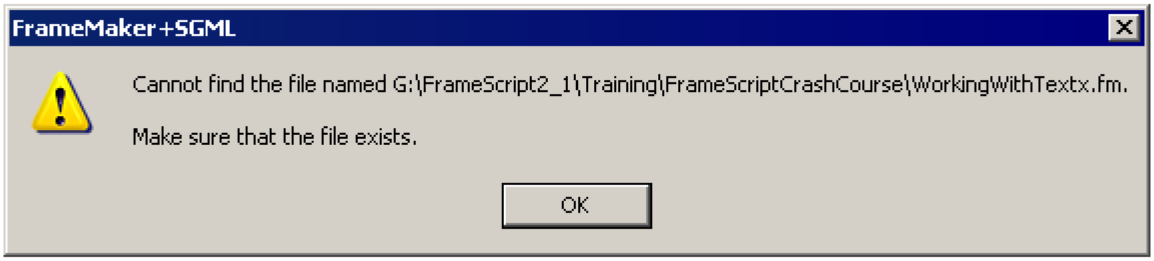
While you can use platform-specific delimiters, your code will be more portable if you use the cross-platform **DIRSEP** delimiter.

You should make sure that the document was actually opened before attempting to work with it. You can do this by initializing the **ErrorCode** variable to **0** (zero) or **Success** before opening the document, and then check its value after the **Open Document** command.

Code Listing 4-2

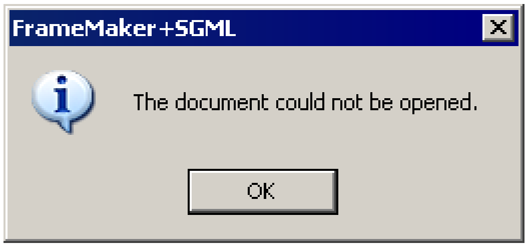
```
// Set a variable for the full path name of the file.
Set vDocName = 'G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Training'+
  DIRSEP+'FrameScriptCrashCourse'+DIRSEP+'WorkingWithTextx.fm';
```

```
// Initialize the ErrorCode variable.
Set ErrorCode = Success;

// Attempt to open the document.
Open Document File(vDocName) NewVar(vCurrentDoc);

// Test the ErrorCode value.
If ErrorCode not= Success
  MsgBox 'The document could not be opened.    ';
  LeaveSub;  // Exit the script.
EndIf
```



Cannot find the file named G:\FrameScript2_1\Training\FrameScriptCrashCourse\WorkingWithTextx.fm.

Make sure that the file exists.



The document could not be opened.

Suppressing FrameMaker Open Errors

You can see that FrameMaker displayed its own message because it could not find the specified file. Then the **MsgBox** message from the script was displayed. You can suppress FrameMaker's built-in messages by using the **AlertUserAboutFailure(False)** option on the **Open Document** command.

Code Listing 4-3

```
// Set a variable for the full path name of the file.
Set vDocName = 'G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Training'+
  DIRSEP+'FrameScriptCrashCourse'+DIRSEP+'WorkingWithTextx.fm';

// Initialize the ErrorCode variable.
Set ErrorCode = Success;

// Attempt to open the document.
Open Document File(vDocName) AlertUserAboutFailure(False)
  NewVar(vCurrentDoc);

// Test the ErrorCode value.
If ErrorCode not= Success
  MsgBox 'The document could not be opened.    ';
  LeaveSub;  // Exit the script.
EndIf
```

When you run this code, you will only get the FrameScript message box. The **AlertUserAboutFailure(False)** option will also suppress other messages, such as missing fonts, missing graphics, unresolved cross-references, etc. If you want your document to open with these conditions, you must include other parameters on the **Open Document** command. Here is a list and description of the some of the parameters.

Open Document Option	Description
`FileIsInUse(ResetLockAndContinue)`	Opens a network locked document. Be careful with this option because someone else may be using the file.
`FileIsOldVersion(OK)`	Opens an older version FrameMaker file with a newer version of the program.
`FontNotFoundInCatalog(OK)`	Ignores missing fonts in the document's catalogs.
`FontNotFoundInDocument(OK)`	Ignores missing fonts in the document.
`LanguageNotAvailable(OK)`	Opens a file with missing dictionaries. Be careful with this option because text in the file may reflow.
`RefFileNotFound(AllowAllRefFilesUnFindable)`	Ignores missing graphics, cross-references, and text insets.

Here is an example that opens all of the FrameMaker documents in a folder, ignores the "File is old version message," and saves them. This is script is useful for updating a series of documents to a new version of FrameMaker; for example, FrameMaker 6 to FrameMaker 7.

Code Listing 4-4

```
// Make sure the script is running with FrameMaker 7.
If VersionMajor < 7
  MsgBox 'This script requires FrameMaker 7 or higher.   ';
  LeaveSub; // Exit the script.
EndIf

// Prompt the user for a folder.
DialogBox Type(ChooseFile) Mode(OpenDirectory) Button(vButton)
  Title('Please choose a folder of files to update:')
  NewVar(vFolder);

// Loop through the documents in the folder.
Loop ForEach(File) In(vFolder) LoopVar(vFile)
  // See if it is a FrameMaker file.
  Find String('.fm') InString(vFile) NoCase Suffix
    ReturnStatus(vFound);
  If vFound
    // Open the document.
    Open Document File(vFile) FileIsOldVersion(OK)
      AlertUserAboutFailure(False) NewVar(vCurrentDoc);
    // If the document is opened, save it and close it.
    If vCurrentDoc
      Save Document DocObject(vCurrentDoc);
      Close Document DocObject(vCurrentDoc) IgnoreMods;
    EndIf
  EndIf
EndLoop
```

Opening Documents Invisibly

The **Open Document** command can open documents but keep them hidden from view when you use the **MakeVisible(False)** option. Your scripts will run much faster on invisible documents, and they will look “cleaner” as they run. This is what actually happens when you update a book with FrameMaker; it opens each document in the book, but they are opened invisibly. You will usually do this with FrameScript when you are processing all of the documents in a book or folder. The previous script would be better with the **MakeVisible(False)** option, because there is no reason that you need to see the documents as they open and close.

Here are some important points about working with invisible documents.

- There are some things that you cannot do with an invisible document, such as using F-codes.
- You should always specify the **DocObject** variable when working with invisible documents.
- If a script fails when working on an invisible document, try running the script again with the document visible. Some commands may not work correctly on an invisible document.
- Make sure you close or display an invisible document before your script ends. If an invisible document is left opened, you may not know it.

To display an invisible document, set its **IsOnScreen** property to **1** (or **True**). You can do this in scripts that create and generate a report or log. You leave the report document invisible as you write to it, and then display it for the user at the end of the script. Here is an incomplete code snippet that illustrates this idea.

```
// Run a subroutine to make a report.
// The report is opened invisibly.
Run MakeReport;

// Commands that process documents or books and write to the report.
...

// At the end of the script, make the report visible.
Set vReportDoc.IsOnScreen = 1;
```

Books

To open an existing book, use the **Open Book** command and supply the complete path to the book in the **File** parameter. Use the **NewVar** parameter to store the book object in a variable.

Code Listing 4-5

```
// Set a path to the book.
Set vBookPath = 'G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'art8sfs.book';
// Open the book.
Open Book File(vBookPath) NewVar(vBook);
```

Creating New Documents and Books

Documents

The **New Document** command is used to create a new FrameMaker document. You can use the **Portrait** or **Landscape** option to make a document based on FrameMaker's built-in templates. This script opens a portrait document; it is equivalent to choosing **File** > **New** > **Document** and clicking the Portrait button.

Code Listing 4-6

```
// Make a new, portrait FrameMaker document.
New Document NewVar(vCurrentDoc) Portrait;
```

You can add also add parameters to change properties as the document is created. This code turns off borders, rulers, and text symbols in the newly created document.

Code Listing 4-7

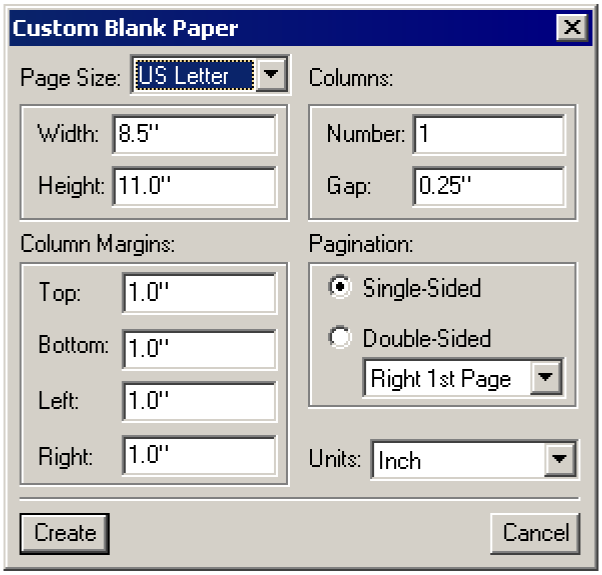
```
// Make a new, portrait FrameMaker document.
New Document NewVar(vCurrentDoc) Portrait ViewRulers(0)
  ViewTextSymbols(0) ViewBorders(0);
```

To use an existing file as a template, specify a path to the file in the **Template** parameter. A copy of the file will open as an "Untitled" document.

Code Listing 4-8

```
// Make an untitled document based on a template.
New Document NewVar(vCurrentDoc)
  Template('N:'+DIRSEP+'Templates'+DIRSEP+'Online.fm');
```

There are a series of parameters that you can use to create custom documents. These are similar to the options in the Custom Blank Paper dialog box in the interface. You can use as many of the parameters as you need to; defaults will be provided for parameters that you leave off.

Custom Blank Paper Dialog Box	Parameter	Description
 Custom Blank Paper Page Size: US Letter Width: 8.5" Height: 11.0" Columns: Number: 1 Gap: 0.25" Column Margins: Top: 1.0" Bottom: 1.0" Left: 1.0" Right: 1.0" Pagination: ● Single-Sided ○ Double-Sided Right 1st Page Units: Inch Create Cancel	`Width`	The width of the document.
	`Height`	The height of the document.
	`NumCols`	The number of columns.
	`ColumnGap`	The gap between the columns.
	`TopMargin`	The top margin
	`BottomMargin`	The bottom margin
	`LeftInsideMargin`	The left margin on a single-sided document or the inside margin on a double-sided document.
	`RightOutsideMargin`	The right margin on a single-sided document or the outside margin on a double-sided document.
	`Sidedness`	Use **SingleSided** for a single-sided document; **FirstPageRight** for a double-sided document starting on a right page; and **FirstPageLeft** for a double-sided document starting on a left page.
	`MakeVisible`	**True** (default) or **False**.

Books

The **New Book** command makes a new book; you specify the full path to the book in the **Name** parameter. The **NewVar** parameter sets an object variable to the new book.

Code Listing 4-9

```
// Make a new book.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);
```

A new book appears on the screen; it must be saved using the **Save Book** command before it is written to the disk (see the next section). A new book will be empty when you first create it. See “Adding Book Components to a Book” on page 4-8 to learn how to add components to books.

Saving Documents and Books

Documents

To save a document, you use the **Save Document** command and specify the document variable on the **DocObject** parameter. If the document is “untitled,” you must specify a complete path name on the **File** parameter. To see if a document is untitled, you can test its **Name** property; if it is an empty string, the file hasn’t yet been saved.

```
// Save a "named" document.
Save Document DocObject(vCurrentDoc);
```

```
// Save an "untitled" document.
Save Document DocObject(vCurrentDoc)
  File('C:'+DIRSEP+'TestFolder'+DIRSEP+'TestDoc.fm');
```

If a document is already named, you can use the **File** parameter to save the document with a new name. This is the same as using the “Save As” command in the FrameMaker interface.

You can save a FrameMaker document in different formats by using the **FileType** parameter. For instance, to save a document as MIF (Maker Interchange Format), use the **SaveFmtInterchange** value. Here is a script that saves the active document as MIF. It drops the .fm extension from the current name and adds the .mif extension on the saved document.

Code Listing 4-10

```
// Save a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make sure the document is not Untitled.
If vCurrentDoc.Name = ''
  MsgBox 'Save the document and rerun the script.    ';
  LeaveSub; // Exit the script.
EndIf

// Drop the .fm extension from the document's name.
Set vName = vCurrentDoc.Name;
Find String('.fm') InString(vName) NoCase Suffix
  ReturnStatus(vFound) ReturnPos(vPos);
If vFound
  // Drop the .fm extension.
  Get String FromString(vName) EndPos(vPos-1) NewVar(vName);
EndIf

// Add the .mif extension.
Set vName = vName + '.mif';

// Save the document as MIF.
Save Document DocObject(vCurrentDoc) File(vName)
  FileType(SaveFmtInterchange);
```

Books

To save a book, use the **Save Book** command with the **BookObject** parameter. If the book is untitled, you must supply a path and filename using the **File** parameter. If the book already has a name, you can omit the **File** parameter, unless you want to “Save As” the book to a new name.

Closing Documents and Books

Documents

Use the **Close Document** command to close an open document. If the document has unsaved changes, the command will fail and the document will not be closed. To close a document without saving changes, use the **IgnoreMods** option.

Code Listing 4-11

```
// Close the document. Changes will NOT be saved!
Close Document DocObject(vCurrentDoc) IgnoreMods;
```

Normally, you will use the **Save Document** and **Close Document** commands in sequence in your scripts.

Code Listing 4-12

```
// Save and close the document.
Save Document DocObject(vCurrentDoc);
Close Document DocObject(vCurrentDoc) IgnoreMods;
```

Books

The **Close Book** command works the same way as the **Close Document** command, except that you specify the book to close with the **BookObject** variable. To close an unsaved book without saving it, use the **IgnoreMods** option.

```
// Close the book. Changes will NOT be saved!
Close Book BookObject(ActiveBook) IgnoreMods;
```

Code Listing 4-13

```
// Close the book after saving the changes.
Save Book BookObject(ActiveBook);
Close Book BookObject(ActiveBook) IgnoreMods;
```

Working With Book Components

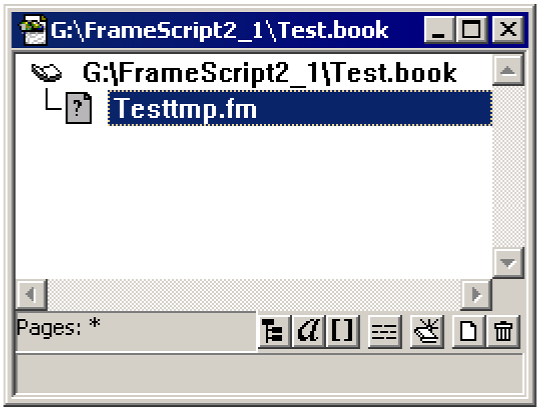
Each file is a book is called a *book component*. Each component is represented by a **BookComponent** object.

Adding Book Components to a Book

You add a component to a book by using the **New BookComponent** command. The book has to be open in order to add components to it. You specify the book object on the **BookObject** parameter of the **New BookComponent** command. The following script creates a new book and adds a component to it.

Code Listing 4-14

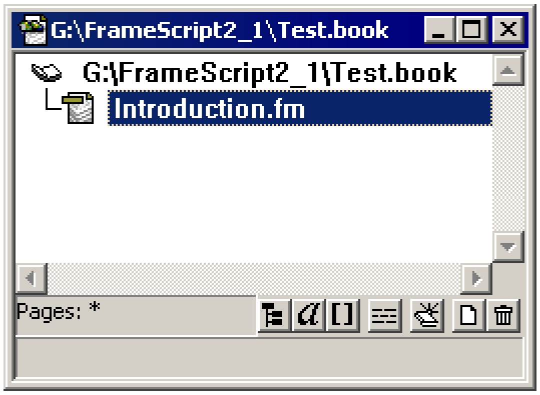
```
// Make a new book and add a component to it.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);
New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
```



You can see that the new component has a generic name of a document that does not (necessarily) exist on disk. If the component name does not match the name of a file on disk, you will see a question mark in the icon next to the name. You can use the new book component's **Name** property to name the component.

Code Listing 4-15

```
// Make a new book and add a component to it.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);
New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);

// Assign a name to the book component.
Set vBookComp.Name = 'Introduction.fm';
```



The screenshot shows a FrameMaker document icon next to the component name. This means that the file “Introduction.fm” exists in the same folder as the book.

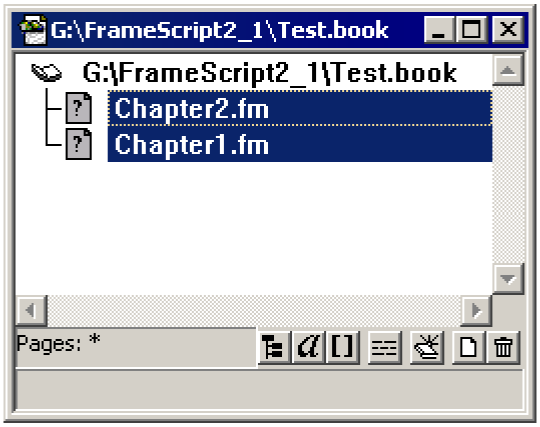
Rearranging Book Components

Each new component added to a book is added above any existing book components. Here is an example that makes a book and adds two components to it.

Code Listing 4-16

```
// Make a new book and add two components to it.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);
```

```
New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
// Assign a name to the book component.
Set vBookComp.Name = 'Chapter1.fm';

New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
// Assign a name to the book component.
Set vBookComp.Name = 'Chapter2.fm';
```



G:\FrameScript2_1\Test.book

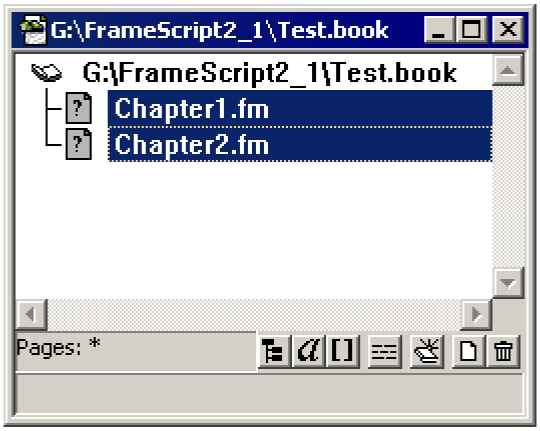
G:\FrameScript2_1\Test.book
- Chapter2.fm
- Chapter1.fm

Pages: *

The components appear in the book in reverse order. You can rearrange components by using their **NextComponentInBook** and **PrevComponentInBook** properties.

Code Listing 4-17

```
// Make a new book and add two components to it.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);

New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
// Assign a name to the book component.
Set vBookComp.Name = 'Chapter1.fm';

New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
// Assign a name to the book component.
Set vBookComp.Name = 'Chapter2.fm';
// Move the component to the end of the book.
Set vBookComp.NextComponentInBook = 0;
```



G:\FrameScript2_1\Test.book

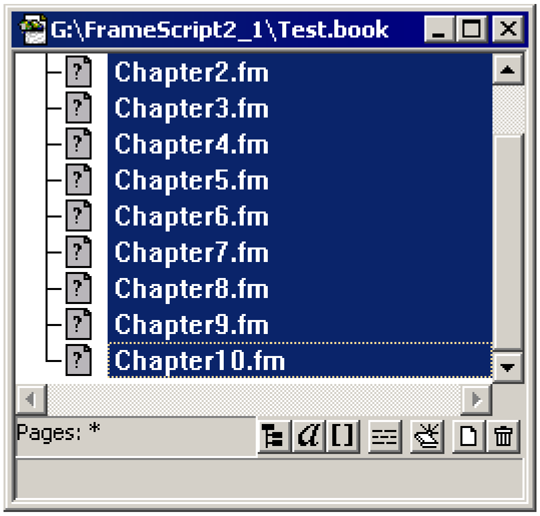
G:\FrameScript2_1\Test.book
- Chapter1.fm
- Chapter2.fm

Pages: *

By setting the last component's **NextComponentInBook** property to **0**, the component is moved to the bottom of the book. This technique will work regardless of how many components you are adding. The only restriction is that you can't use it when adding the *first* component to the book. Here is a script that uses a loop to add a series of numbered chapters to a book.

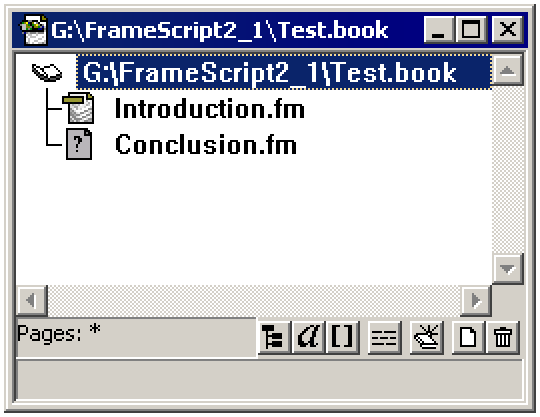
Code Listing 4-18

```
// Make a new book.
New Book Name('G:'+DIRSEP+'FrameScript2_1'+DIRSEP+'Test.book')
  NewVar(vCurrentBook);

// Use a loop to add a 10 components to the book.
Loop While(n <= 10) LoopVar(n) Init(1) Incr(1)
  // Add a component to the book.
  New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
  // Assign a name to the book component using the counter.
  Set vName = 'Chapter'+n+'.fm';
  Set vBookComp.Name = vName;
  If n > 1 // Skip the first component.
    // Move the component to the end of the book.
    Set vBookComp.NextComponentInBook = 0;
  EndIf
EndLoop
```


G:\FrameScript2_1\Test.book

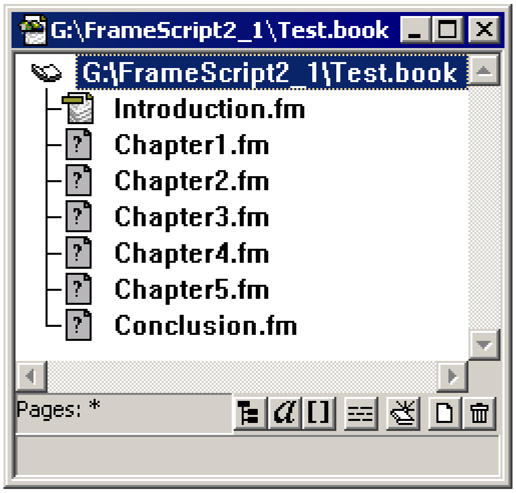
Chapter2.fm
Chapter3.fm
Chapter4.fm
Chapter5.fm
Chapter6.fm
Chapter7.fm
Chapter8.fm
Chapter9.fm
Chapter10.fm

The script below adds new components to an existing book that already contains components. They will be added between the original first and second components. Here is a screenshot of the book before the components are added.



Code Listing 4-19

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Find the last component in the book.
Set vLastComp = vCurrentBook.FirstComponentInBook;
Loop Until(vLastComp.NextComponentInBook = 0)
  Set vLastComp = vLastComp.NextComponentInBook;
EndLoop

// Use a loop to add a 5 components to the book.
Loop While(n <= 5) LoopVar(n) Init(1) Incr(1)
  // Add a component to the book.
  New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
  // Assign a name to the book component using the counter.
  Set vName = 'Chapter'+n+'.fm';
  Set vBookComp.Name = vName;
  // Move the component so it's before the last one.
  Set vBookComp.NextComponentInBook = vLastComp;
EndLoop
```



Renaming Book Components

When you rename a book component with the FrameMaker interface (version 6 and above), three things happen.

1. The component name is changed in the book window.
2. The component file is renamed on the disk (if possible).
3. Cross-references (and other references) from the other book components to the renamed one are updated.

With FrameScript, you must perform each step separately in your code. Each of the steps is outlined in this section.

To rename the component in the book window (step 1), you simply change its **Name** property. This script renames the selected book component in the active book.

Code Listing 4-20

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Make sure a component is selected.
If vCurrentBook.FirstSelectedComponentInBook = 0
  MsgBox 'Please select a component to rename.    ';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the selected component.
Set vBookComp = vCurrentBook.FirstSelectedComponentInBook;

// Rename the component.
Set vBookComp.Name = 'Chapter01.fm';
```

If you are using Windows FrameMaker, you can use the System Object commands to rename the file on disk (step 2). You will need the complete path for both the old and new component names. Here is a scripts that prompts you for a new name and derives the new path from the old component name. It then renames the component in the book window and on disk.

Code Listing 4-21

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Make sure a component is selected.
If vCurrentBook.FirstSelectedComponentInBook = 0
  MsgBox 'Please select a component to rename.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the selected component.
  Set vBookComp = vCurrentBook.FirstSelectedComponentInBook;
EndIf

// Set a variable for the old file name.
Set vOldName = vBookComp.Name;
```

```
// Prompt the user for a new name.
Set vNewName = '';
Loop While(vNewName = '')
  DialogBox Type(String) NewVar(vNewName) Button(vButton)
    Title('Enter a name for the component:');
  If vButton = CancelButton
    LeaveSub; // Cancelling the dialog exits the script.
  EndIf
EndLoop

// Find the path name of the selected component.
Find String(DIRSEP) InString(vOldName) Backward ReturnPos(vPos);
Get String FromString(vOldName) EndPos(vPos) NewVar(vNewPath);
// Get the base file name of the old file for later use.
Get String FromString(vOldName) StartPos(vPos+1) NewVar(vOldFile);

// Rename the component.
Set vBookComp.Name = vNewName;

// Check the platform before attempting to rename the file on disk.
If Platform not= 'Intel'
  MsgBox 'Use the Finder to rename the file on the disk.    ';
  LeaveSub; // Exit the script.
EndIf

// Create a system object (Windows only) and rename the file.
New ESystem NewVar(vESystem);
Run vESystem.RenameFile From(vOldName) To(vNewPath+vNewName);
```

For completeness and to prevent errors, you should check for the existence of the old file on disk before you attempt to rename it. Also, if the new file exists, you may not want to overwrite it. For more information, see the *FrameScript System Object Reference*.

For step 3, you must loop through all of the book components and update any references to the old component name with the new name. You need to check the following references:

- External cross-references that reference the old file name.
- Text insets that may have originated from the renamed file.
- Hypertext marker text that reference the old file name.

The following script has three loops that will update these references. For clarity, the loops are contained in a subroutine called **UpdateReferences**. The call to this subroutine will go inside of a loop that will open each book component in the book, except the one being renamed. This code should go at the end of the previous script.

Code Listing 4-22

```
// Update the other book components.
// Loop through the book components.
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  // Make sure the component is not the renamed component.
  If vBookComp.Name not= vNewPath+vNewName
    // Attempt to open the component.
    Set ErrorCode = Success;
    Open Document File(vBookComp.Name) AlertUserAboutFailure(False)
      RefFileNotFound(AllowAllRefFilesUnFindable)
      UpdateTextReferences(No) UpdateXRefs(No)
      MakeVisible(False) NewVar(vCurrentDoc);
    If ErrorCode not= Success
      // If the document can't be opened, write a message to
      // the Console.
      Write Console vBookComp.Name+' could not be opened.';
    EndIf
    If vCurrentDoc
      // Call the subroutine to update the references.
      Run UpdateReferences;
      // Save and close the document.
      Save Document DocObject(vCurrentDoc);
      Close Document DocObject(vCurrentDoc) IgnoreMods;
    EndIf
  EndIf
EndLoop

Sub UpdateReferences
//
// Loop through the cross-references.
Loop ForEach(XRef) In(vCurrentDoc) LoopVar(vXRef)
  // See if the cross-reference is to the old file name.
  If vXRef.XRefFile = vOldName
    // Change the reference to the new name.
    Set vXRef.XRefFile = vNewPath+vNewName;
  EndIf
EndLoop

// Loop through the text insets.
Set vTi = vCurrentDoc.FirstTiInDoc;
Loop While(vTi)
  // See if the text inset is from the old file.
  If vTi.TiFile = vOldName
    // Change the reference to the new name.
    Set vTi.TiFile = vNewPath+vNewName;
  EndIf
  Set vTi = vTi.NextTiInDoc;
EndLoop
```

```
// Loop through the Hypertext markers.
// Since Hypertext markers usually use relative paths, use the
// base file name of the old file.
Loop ForEach(Marker) In(vCurrentDoc) LoopVar(vMarker)
  // See if it is a Hypertext marker.
  If vMarker.MarkerTypeId.Name = 'Hypertext'
    // Change the text.
    Set vMarkerText = vMarker.MarkerText;
    Get String FromString(vMarkerText) ReplaceAll(vOldFile)
      With(vNewName) NewVar(vMarkerText);
    Set vMarker.MarkerText = vMarkerText;
  EndIf
EndLoop
//
EndSub
```

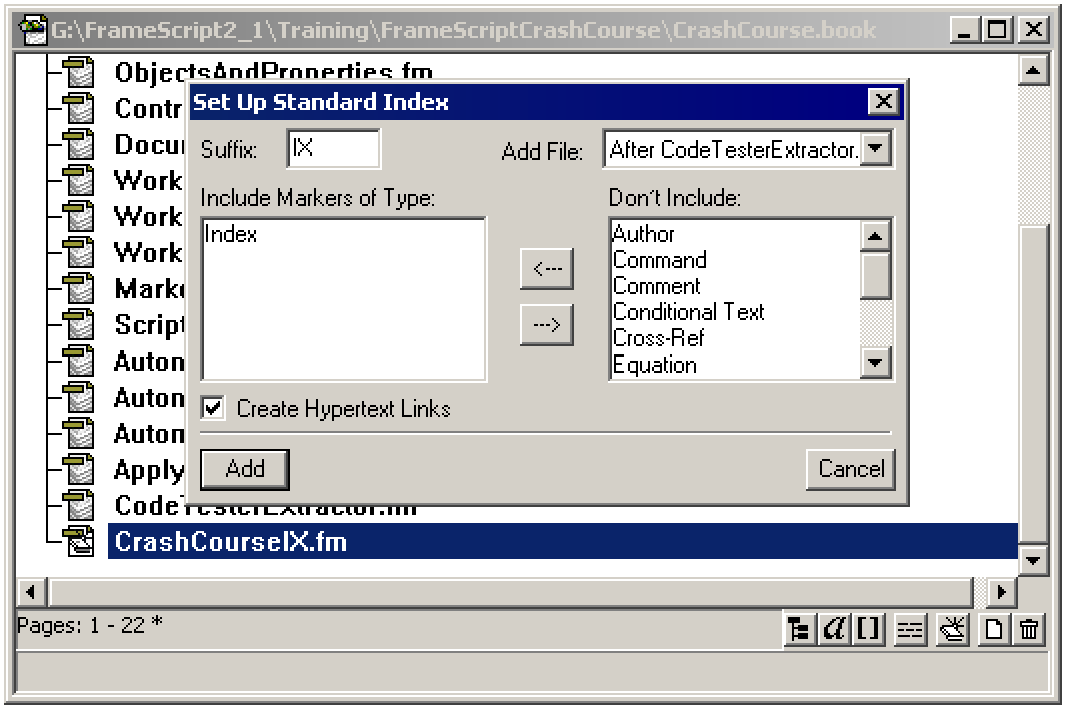
Working With Generated Files

Generated files, such as tables of contents and indexes, are created by FrameMaker by extracting paragraphs or marker text from documents in a book. You use FrameScript to create and work with generated files by setting special book component properties.

A book component's type is identified by its **BookComponentType** property. Here is a list and description of the most common values.

BookComponentType Value	Description
`BkNotGeneratable`	The component is not a generated file.
`BkIndexAuthor`	Index of Author markers.
`BkIndexMarker`	Index of custom markers.
`BkIndexStan`	Index of standard Index markers.
`BkListMarker`	List of markers.
`BkListMarkerAlpha`	Alphabetical list of markers.
`BkListPgf`	List of paragraphs.
`BkListPgfAlpha`	Alphabetical list of paragraphs.
`BkToc`	Standard table of contents.

When you use the FrameMaker interface to add a generated file to a book, it is given a name based on the book's name and the type of generated file that it is. For example, if you add a table of contents to a book called "MyBook.book," it will be named "MyBookTOC.fm." You should follow this convention when naming generated book components with FrameScript. To find out the correct suffix for a book component type, add a temporary one to a book and see what the default suffix is in the **Set Up** dialog box. Here is a screenshot of the **Set Up Standard Index** dialog box. You can see that the Suffix is "IX" in both the dialog box and the book window. The front of the name is "CrashCourse" because the book is called "CrashCourse.book."



Suffix: IX

Add File: After CodeTesterExtractor.

Include Markers of Type: Index

Don't Include: Author, Command, Comment, Conditional Text, Cross-Ref, Equation

☑ Create Hypertext Links

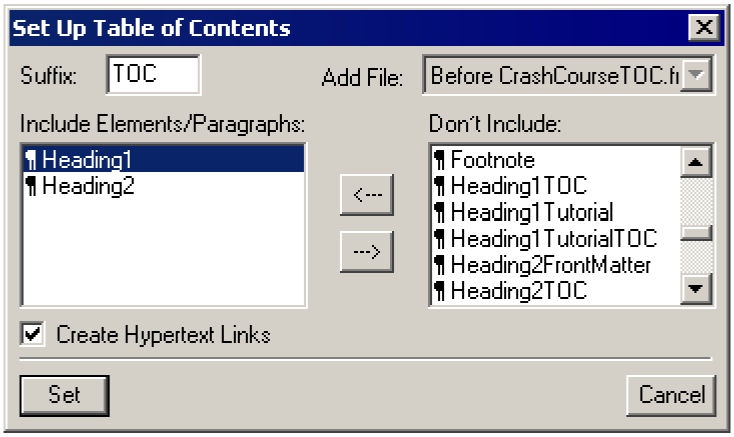
Pages: 1 - 22 *

The following script adds a table of contents **BookComponent** to the active book. It sets the appropriate name by deriving it from the book's name, and sets the **BookComponentType** value to **BkToc**.

Code Listing 4-23

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Get the base file name of the book.
// Find the last folder separator in the path.
Find String(DIRSEP) InString(vCurrentBook.Name) Backward
  ReturnPos(vPos);
// Drop the path.
Get String FromString(vCurrentBook.Name) StartPos(vPos+1)
  NewVar(vBaseName);
// Find the book extension.
Find String('.book') InString(vBaseName) Suffix NoCase
  ReturnPos(vPos);
// Drop the extension.
If vPos > 0
  Get String FromString(vBaseName) EndPos(vPos-1) NewVar(vBaseName);
EndIf

// Add the TOC component to the book and set its name and type.
New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
Set vBookComp.Name = vBaseName+'TOC.fm';
Set vBookComp.BookComponentType = BkToc;
```

You still need to set the items that will be included in the generated file. For a TOC or list of paragraphs, you need to supply a list of paragraphs that will be included; for a list or index of markers, you need to supply the marker types. You do this by setting the **ExtractTags** property of the **BookComponent**. The **ExtractTags** property is a string list containing the appropriate data. The string list will include the same names that are in the **Include** window of the **Set Up** dialog box in the FrameMaker interface. For example, the following screenshot shows a TOC that will including “Heading1” and “Heading2” paragraphs.



Set Up Table of Contents

Suffix: TOC

Add File: Before CrashCourseTOC.fı

Include Elements/Paragraphs:
¶ Heading1
¶ Heading2

Don't Include:
¶ Footnote
¶ Heading1TOC
¶ Heading1Tutorial
¶ Heading1TutorialTOC
¶ Heading2FrontMatter
¶ Heading2TOC

☑ Create Hypertext Links

Here is the code to set up the corresponding string list.

Code Listing 4-24

```
// Set up a string list for the include paragraphs.
New StringList NewVar(vExtractTags);
Add Member('Heading1') To(vExtractTags);
Add Member('Heading2') To(vExtractTags);
```

Below is this code added to the script in Code 4-23. A line is also included that sets the book component's **ExtractTags** property to the string list.

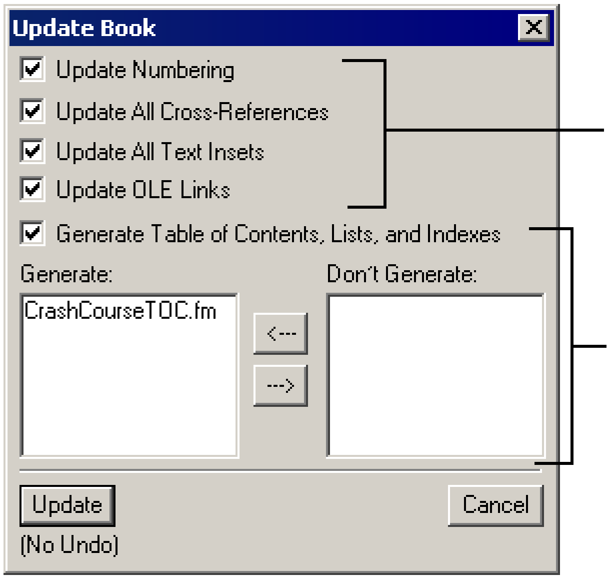
Code Listing 4-25

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Get the base file name of the book.
// Find the last folder separator in the path.
Find String(DIRSEP) InString(vCurrentBook.Name) Backward
  ReturnPos(vPos);
// Drop the path.
Get String FromString(vCurrentBook.Name) StartPos(vPos+1)
  NewVar(vBaseName);
// Find the book extension.
Find String('.book') InString(vBaseName) Suffix NoCase
  ReturnPos(vPos);
// Drop the extension.
If vPos > 0
  Get String FromString(vBaseName) EndPos(vPos-1) NewVar(vBaseName);
EndIf

// Add the TOC component to the book and set its name and type.
New BookComponent BookObject(vCurrentBook) NewVar(vBookComp);
Set vBookComp.Name = vBaseName+'TOC.fm';
Set vBookComp.BookComponentType = BkToc;
```

```
// Set up a string list for the include paragraphs.
New StringList NewVar(vExtractTags);
Add Member('Heading1') To(vExtractTags);
Add Member('Heading2') To(vExtractTags);
// Set the extract tags property to the string list.
Set vBookComp.ExtractTags = vExtractTags;
```

Updating and Generating Books

The Update Book command in the FrameMaker interface actually performs two major operations. It updates properties in the book components, and generates content for any generated book components in the book. Here is screenshot of the Update Book dialog box.



Update Book

☑ Update Numbering

☑ Update All Cross-References

☑ Update All Text Insets

☑ Update OLE Links

☑ Generate Table of Contents, Lists, and Indexes

Generate: CrashCourseTOC.fm

Don't Generate:

Updates book components

Generates contents for generated files

FrameScript's **Generate BookFiles** command performs both of the operations; it updates book components and generates contents for generated files. Use the **BookObject** parameter to specify the book object. The **Interactive(True)** option allows warnings and error messages to be displayed to the user; use **False** to suppress the messages. The **Visible** option specifies whether to display generated files after they are generate. Use **True** to show them or **False** to hide them.

The **Update Book** command gives you more control over which items are updated or generated. It also allows you to selectively ignore warning messages. The following table shows the parameters that correspond to the Update Book dialog box settings from top to bottom (see the screenshot above). For other parameters, see the *FrameScript Scriptwriter's Reference.*

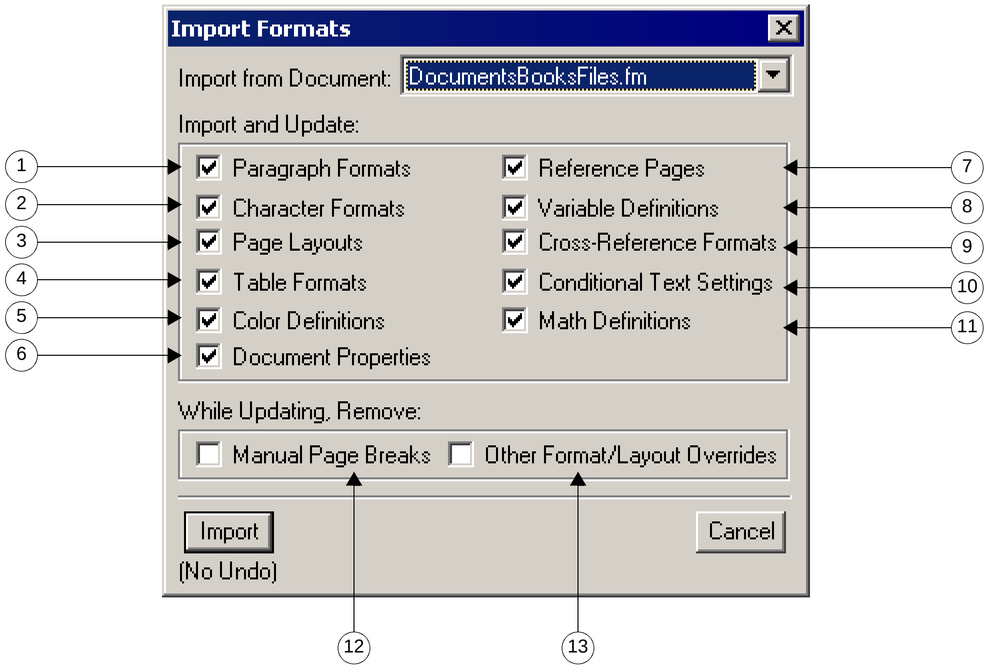
Parameter	Description	Possible Values
`UpdateBookNumbering`	Update numbering in all documents.	True or False
`UpdateBookXRefs`	Update all cross-references.	True or False
`UpdateTextReferences`	Update all text insets.	Yes, No, or DoUserPreference

Parameter	Description	Possible Values
`UpdateBookOLELinks`	Update all OLE links.	True or False
`UpdateBookGeneratedFiles`	Updates generated files.	True or False

When you choose the **UpdateBookGeneratedFiles(True)** option, you must make sure that each **BookComponent** that you want generated has its **GenerateInclude** property set to **True**. This is like moving the file to the Generate window on the left side of the Update Book dialog in the interface.

Importing Formats into Documents and Books

The **Import Formats** command works with both documents and books, depending on whether you specify a **DocObject** or **BookObject** object. The **FromDocObject** parameter is the document object of the template that you are importing formats from. You must specify which formats you want to import by including them as options. Each option corresponds to a checkbox in the Import Formats dialog box, as shown below.



Import Formats

Import from Document: DocumentsBooksFiles.fm

Import and Update:

- ☑ Paragraph Formats
- ☑ Character Formats
- ☑ Page Layouts
- ☑ Table Formats
- ☑ Color Definitions
- ☑ Document Properties
- ☑ Reference Pages
- ☑ Variable Definitions
- ☑ Cross-Reference Formats
- ☑ Conditional Text Settings
- ☑ Math Definitions

While Updating, Remove:

- □ Manual Page Breaks
- □ Other Format/Layout Overrides

1	`Pgf`	6	`DocumentProps`	11	`Math`
2	`Font`	7	`RefPage`	12	`RemovePageBreaks`
3	`Page`	8	`Var`	13	`RemoveExceptions`
4	`Table`	9	`XRef`	NA	`CombinedFonts`
5	`Color`	10	`Cond`		

Here is a sample script that prompts the user for a template and then imports the paragraph, character, and table formats into the active document.

Code Listing 4-26

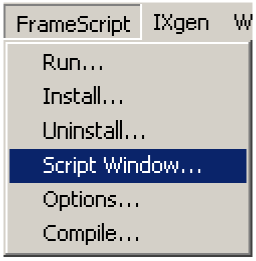
```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

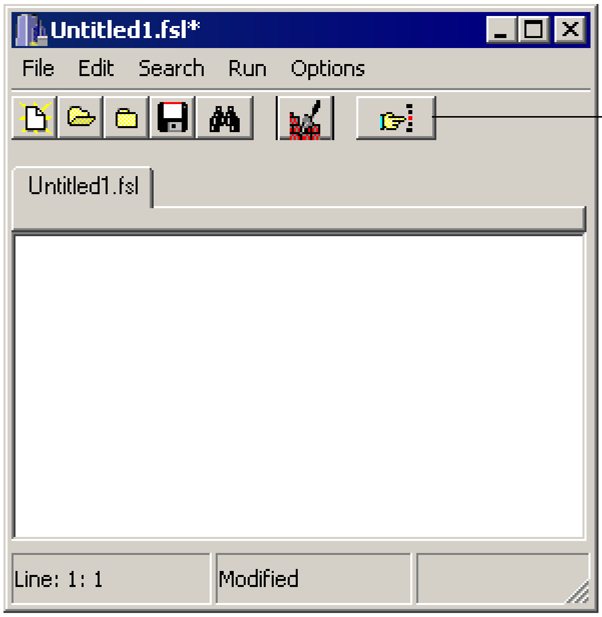
```
// Prompt the user for a template.
DialogBox Type(ChooseFile) Mode(SelectFile) Button(vButton)
  Title('Please select a template:') Init('*.fm')
  NewVar(vTemplatePath);
// If the user cancels the dialog box, exit the script.
If vButton = CancelButton
  LeaveSub; // Exit the script.
EndIf

// Attempt to open the template.
Set ErrorCode = Success;
Open Document File(vTemplatePath) NewVar(vTemplateDoc)
  AlertUserAboutFailure(False) MakeVisible(False);
// If the document wasn't opened, warn the user and exit.
If ErrorCode not= Success
  MsgBox 'The template could not be opened.    ';
  LeaveSub; // Exit the script.
EndIf

// Import the paragraph, character, and table formats.
Import Formats DocObject(vCurrentDoc) FromDocObject(vTemplateDoc)
  Pgf Font Table RemoveExceptions;

// Close the template.
Close DocObject(vTemplateDoc) IgnoreMods;
```

5 *Working With Text*

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **SubCol** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.

To work with the examples in this chapter, open a new FrameMaker document and choose **FrameScript** > **Script Window**. To run code in the Script Window, choose **Run** > **Script**, press Ctrl+R, or click the Run button (the last button on the right).





Run button

Understanding Text Structures

FrameMaker has two text structures, **TextRange** (text range) and **TextLoc** (text location), that are foundational to working with text. They may seem a little obscure at first, but your understanding will become clear as you work with them in scripts.

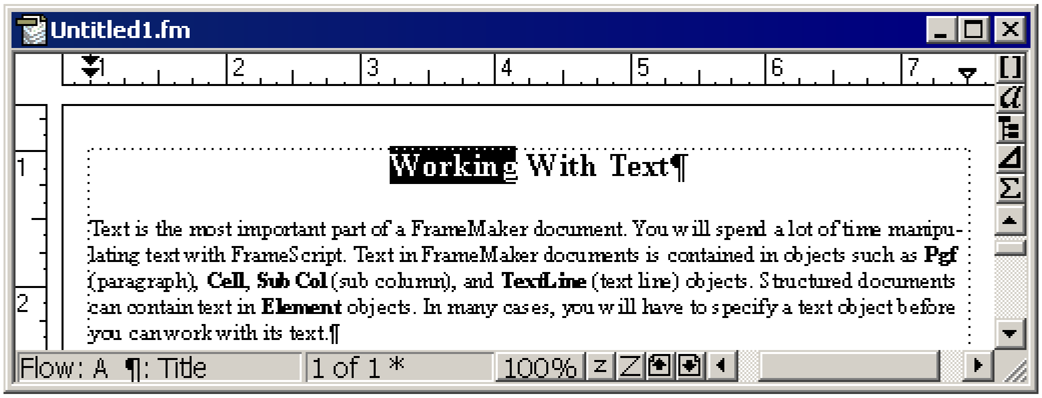
FrameMaker documents have a **TextSelection** property that specifies the text insertion point or text selection in the document. A **TextSelection** is a **TextRange** structure. A

TextRange is bounded by two **TextLoc** structures, one at the beginning of the text range and the other at the end of the text range.

Open a new FrameMaker document and type a small paragraph of text. Select the first word and run the following code.

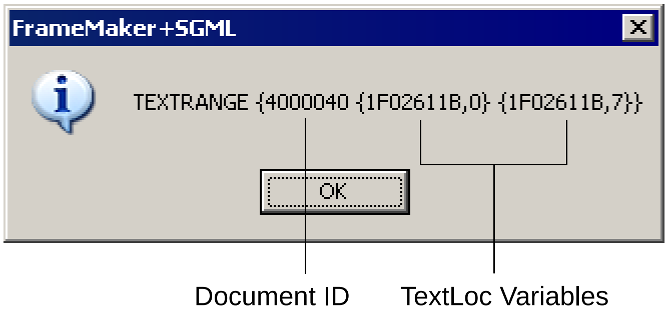
Code Listing 5-1

```
Display TextSelection;
```



Working With Text¶

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **Sub Col** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.¶

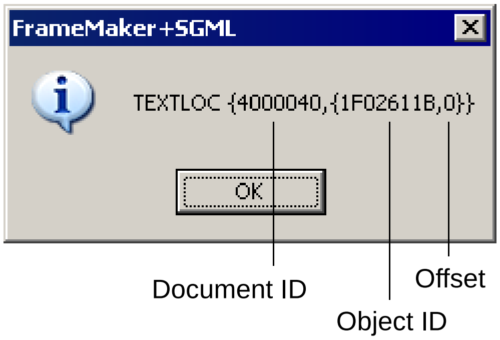


TEXTRANGE {4000040 {1F02611B,0} {1F02611B,7}}

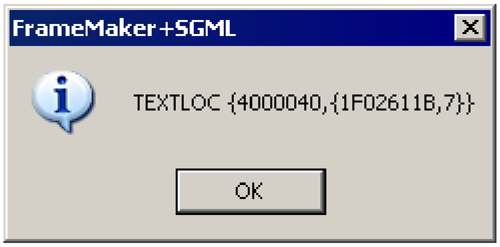
You will see the **TextRange** variable displayed. The first number is the document ID; the second two sets of numbers inside the curly braces represent the **TextLoc** variables. Run the following to see the **TextLoc** variables.

Code Listing 5-2

```
Display TextSelection.Begin;
Display TextSelection.End;
```



TEXTLOC {4000040,{1F02611B,0}}



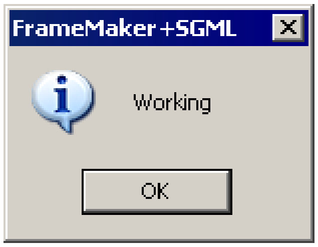
TEXTLOC {4000040,{1F02611B,7}}

Again, the first number is the document ID. The second number is the ID of the object containing the **TextLoc**; in this case a **Pgf** (paragraph) object. The third number is the **Offset** from the beginning of the object. Since the first word in the paragraph is selected, the offset at the beginning of the selection is 0 (zero). In the example above, the end of the selection is offset 7 characters from the beginning of the paragraph.

You can use the **Text** object to display the text that is part of the **TextSelection**.

Code Listing 5-3

```
Display TextSelection.Text;
```

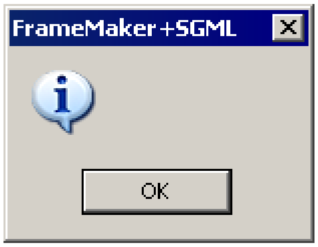


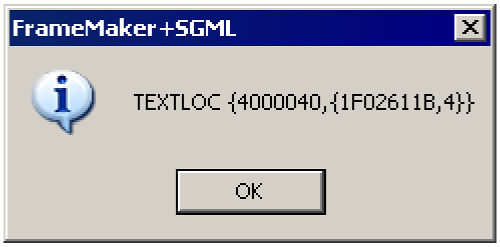
Working

If no text is selected but you have an insertion point in the document, **TextSelection.Text** will return an empty string. In addition, **TextSelection.Begin** and **TextSelection.End** will return the same text location. Click an insertion point in the document and run the following code.

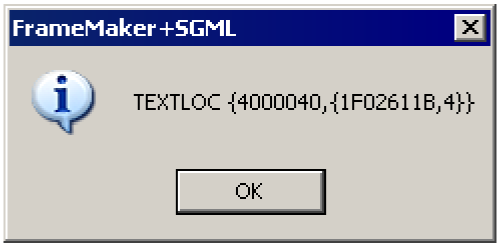
Code Listing 5-4

```
Display TextSelection.Text;
Display TextSelection.Begin;
Display TextSelection.End;
```





TEXTLOC {4000040,{1F02611B,4}}



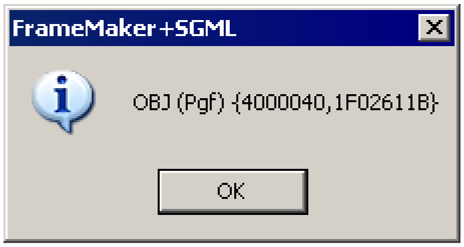
TEXTLOC {4000040,{1F02611B,4}}

As you can see, the beginning and ending points of an insertion point have the same **TextLoc**.

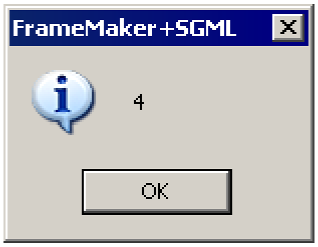
A **TextLoc** structure has two parts; one is the **Pgf** or **TextLine** object, the other is the **Offset** from the beginning of the text object. Click in the document text and run the following code.

Code Listing 5-5

```
Display TextSelection.Begin.Object;
Display TextSelection.Begin.Offset;
```



OBJ (Pgf) {4000040,1F02611B}



4

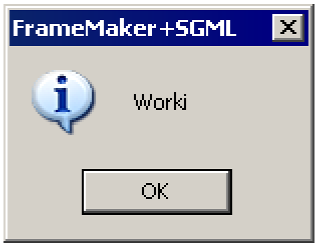
The first line will display the **Pgf** object of the **TextLoc**, and the second line will display the **Offset** of the **TextLoc** from the beginning of the **Pgf**.

New TextRange

Up to now, we have discussed a specific **TextRange**—the document **TextSelection** property. Often, you will make your own text ranges, using the **New TextRange** command. Run the following code on your sample document.

Code Listing 5-6

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Make a new text range.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0) Offset(5);
Display vTextRange.Text;
```

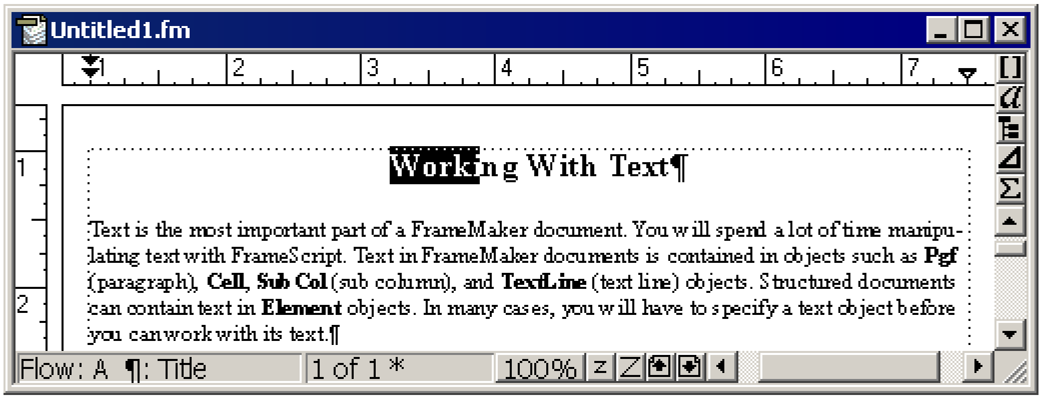


Worki

This will display the first five characters of the first paragraph in your document. You can use set the **TextSelection** property of the document to the **TextRange** in order to select it.

Code Listing 5-7

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Make a new text range.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0) Offset(5);
Set vCurrentDoc.TextSelection = vTextRange;
```



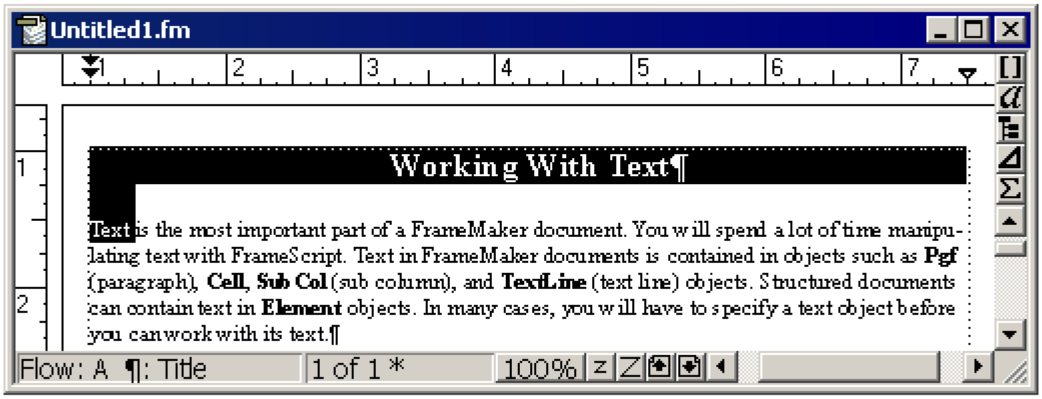
Working With Text¶

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **Sub Col** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.¶

You can make a text range that spans more than one paragraph by adding another **Object** parameter to the **New TextRange** command. Make sure you have at least two paragraphs in your document and try this code.

Code Listing 5-8

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Make a new text range.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0)
  Object(vPgf.NextPgfInFlow) Offset(5);
Set vCurrentDoc.TextSelection = vTextRange;
```



Working With Text¶

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **Sub Col** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.¶

This will select the entire first paragraph and the first five characters of the second paragraph.

New TextLoc

You use the **New TextLoc** command to make a text location in a document. This is useful when you want to add text at a particular location in a text object, such as a **Pgf** (paragraph) or **TextLine** (text line). The following script makes a **TextLoc** after the fifth character in the first paragraph.

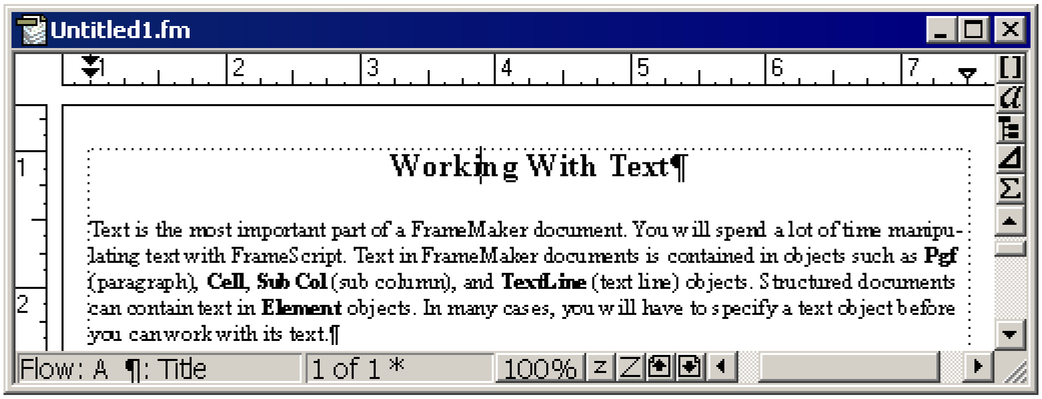
Code Listing 5-9

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Make a text location in the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(5);
```

You can use the following code to place the insertion point at the **TextLoc**.

Code Listing 5-10

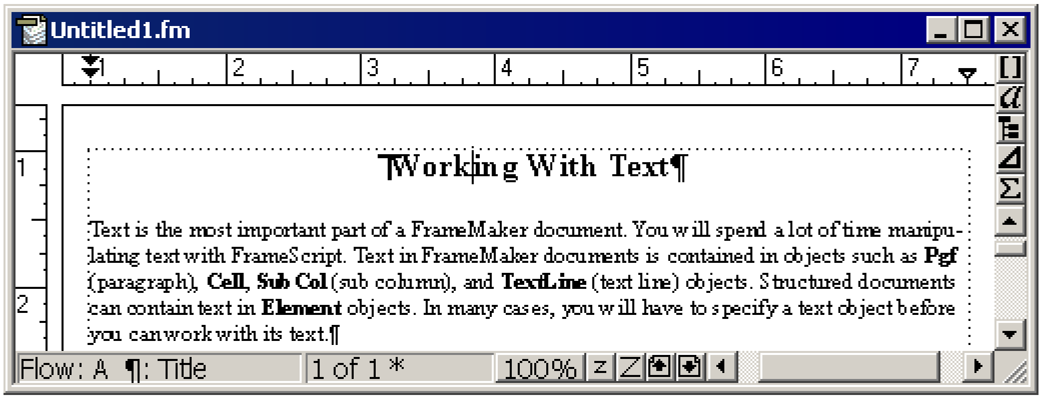
```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Make a text location in the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(5);
Set vCurrentDoc.TextSelection = vTextLoc;
```



Working With Text¶

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **Sub Col** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.¶

It is important to note that the **Offset** parameter of both **New TextRange** and **New TextLoc** counts any anchored objects that are in the text object. To see this, add a marker at the beginning of the first paragraph and rerun the previous script. The insertion point will be placed at the end of the fourth character, because the marker is counted in the offset from the beginning of the paragraph.



TWorking With Text¶

Text is the most important part of a FrameMaker document. You will spend a lot of time manipulating text with FrameScript. Text in FrameMaker documents is contained in objects such as **Pgf** (paragraph), **Cell**, **Sub Col** (sub column), and **TextLine** (text line) objects. Structured documents can contain text in **Element** objects. In many cases, you will have to specify a text object before you can work with its text.¶

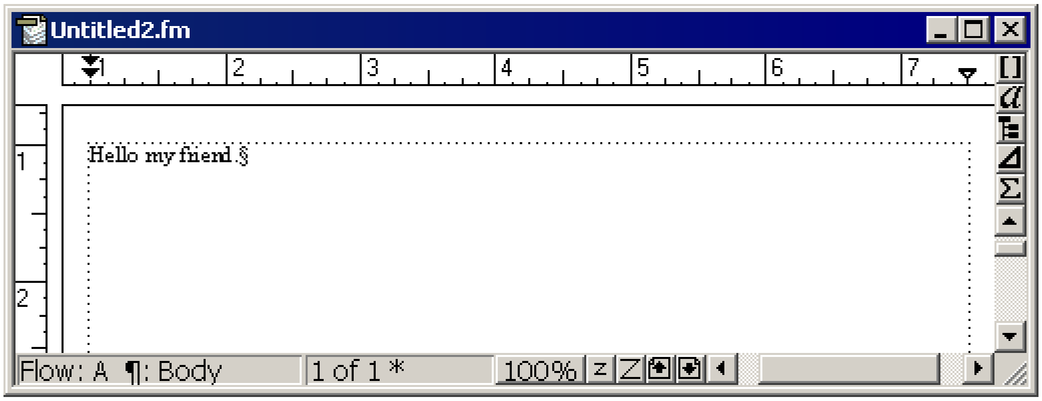
Adding Text

New Text

To add text to a document, you use the **New Text** command. Make a new FrameMaker document and run the following code.

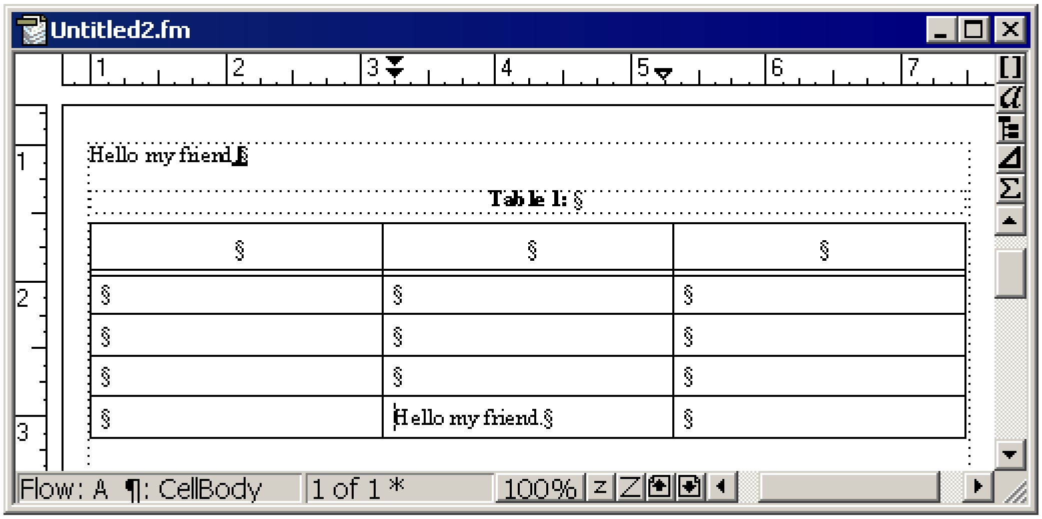
Code Listing 5-11

```
New Text 'Hello my friend.';
```



Hello my friend.§

The text appears at the insertion point of the document. We did not have to specify a location for the text, because the insertion point was in the first paragraph when the new document was created. Add a table to the document, put the insertion point anywhere in the table, and run the script. The text will appear at the insertion point in the table.



Hello my friend.§

Table 1: §

§	§	§
§	§	§
§	§	§
§	§	§
§	Hello my friend.§	§

Most of the time, you will specify a location when using the **New Text** command. To specify a location, you use the **Object** or **TextLoc** parameter on the **New Text** command.

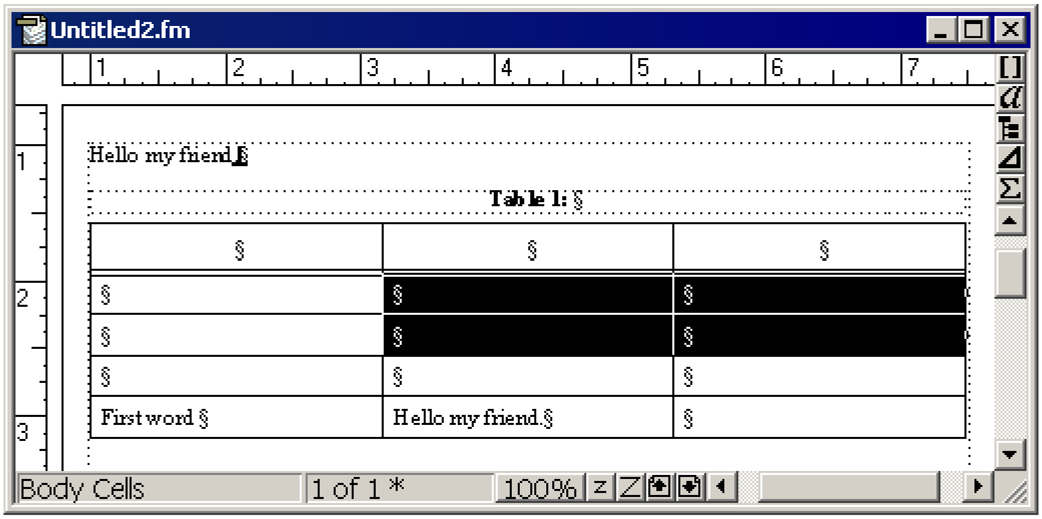
Object

The **Object** parameter will be a text object, such as a **Pgf** (paragraph). When you use the **Object** parameter, the text will be inserted at the *beginning* of the text object.

Select a few of the cells in the table and run the following script to see what happens.

Code Listing 5-12

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for a paragraph in the table.
Set vPgf = vTbl.LastRowInTbl.FirstCellInRow.FirstPgf;
// Add text to the table cell paragraph.
New Text Object(vPgf) 'First word ';
```



Hello my friend.§

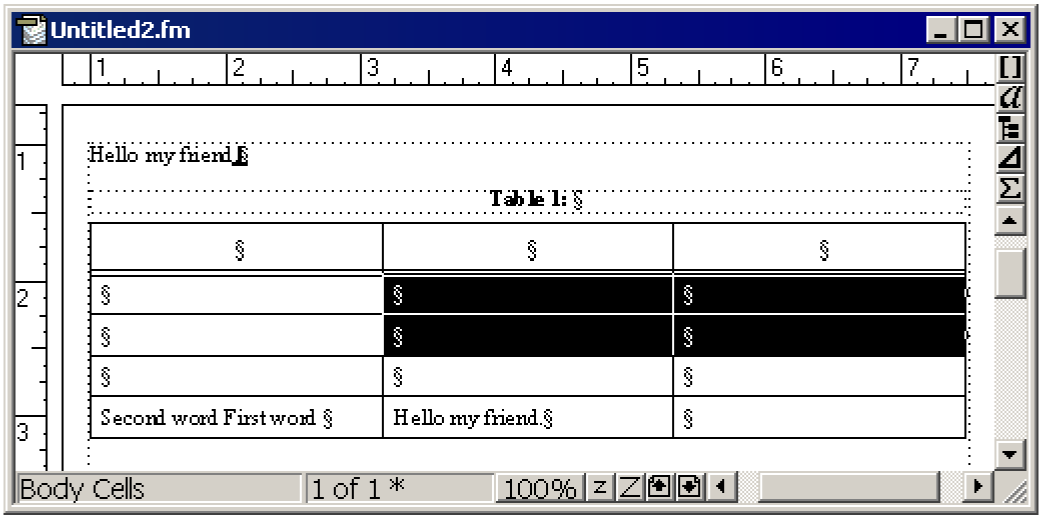
Table 1: §

§	§	§
§	§	§
§	§	§
§	§	§
First word §	Hello my friend.§	§

In the second line, we set a **vTbl** variable for the selected table (**SelectedTbl**). **SelectedTbl** is a document object representing the currently selected table in the document. A table is selected if one or more of its cells are selected or if the insertion point is anywhere in the table. If no table is selected, **SelectedTbl** returns 0 (zero), and our script would fail.

The third line sets a variable (**vPgf**) for the paragraph object of the first cell of the last row of the table. We use this object on the fourth line as the **Object** parameter for the **New Text** command.

Modify the fourth line of the script by changing '**First word** ' to '**Second word** ' and run the script again. The new text appears in *front* of the text we previously added. This illustrates that the **Object** parameter always adds text at the beginning of the text object.



Hello my friend.§

Table 1: §

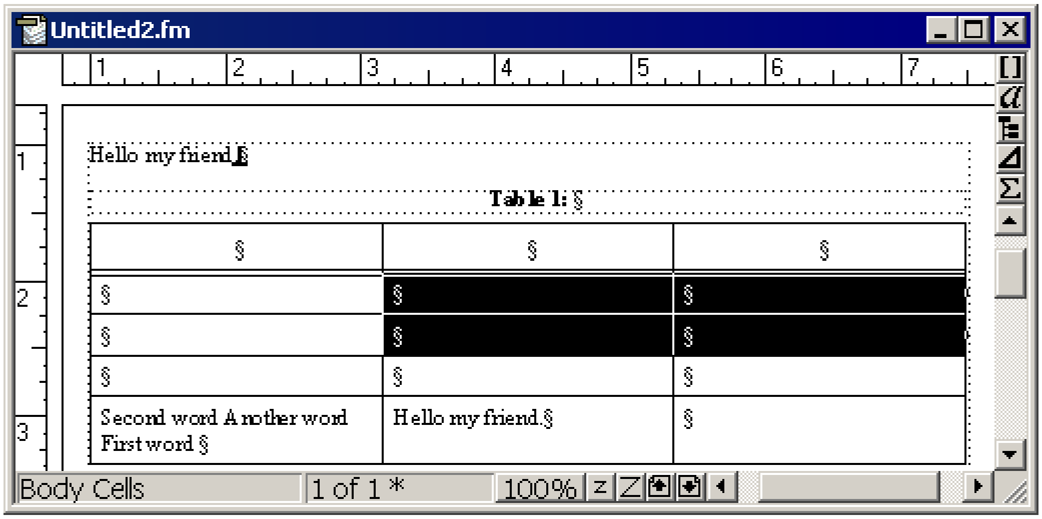
§	§	§
§	§	§
§	§	§
§	§	§
Second word First word §	Hello my friend.§	§

TextLoc

To add text at a particular location within a text object, you use the **TextLoc** parameter. A text location (**TextLoc**) is a specific location within a **Pgf** (paragraph) or **TextLine** (text line). A **TextLoc** consists of an **Object** and an **Offset** from the beginning of the text object. Click anywhere in the table and run the following code.

Code Listing 5-13

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for a paragraph in the table.
Set vPgf = vTbl.LastRowInTbl.FirstCellInRow.FirstPgf;
// Make a text location in the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(12);
// Add text at the text location.
New Text TextLoc(vTextLoc) 'Another word ';
```



Hello my friend.§

Table 1: §

§	§	§
§	§	§
§	§	§
§	§	§
Second word Another word First word §	Hello my friend.§	§

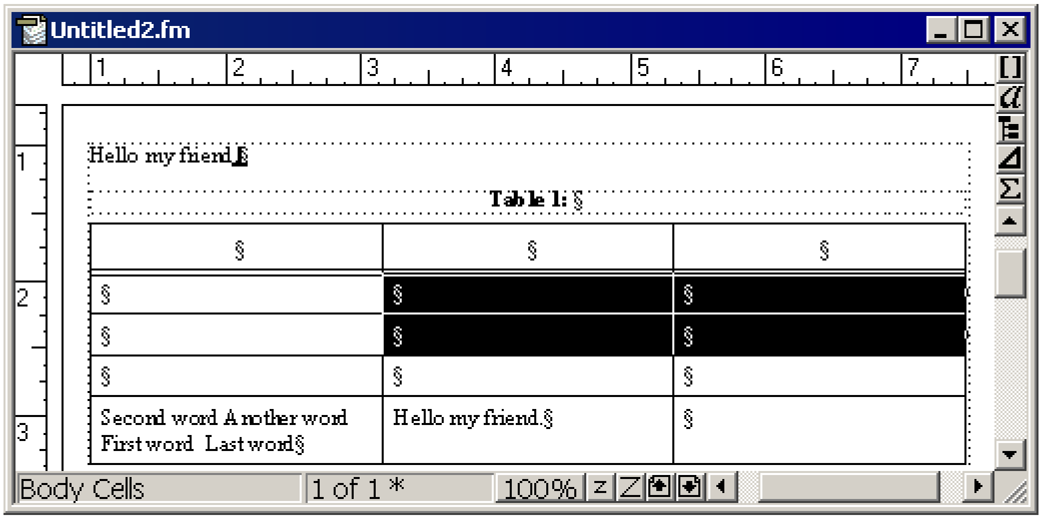
The phrase “Another word ” is inserted after the 12th character in the paragraph.

Many times, you will want to insert text at the end of a text object. FrameScript has a special variable—**ObjEndOffset**—for the end of a **Pgf** or **TextLine**. Use this for the **Offset** parameter in the **New TextLoc** command. Try the following code.

Code Listing 5-14

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for a paragraph in the table.
Set vPgf = vTbl.LastRowInTbl.FirstCellInRow.FirstPgf;
// Make a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);
// Add text at the text location.
New Text TextLoc(vTextLoc) 'Last word ';
```

Actually, **ObjEndOffset** falls to the right of the paragraph mark in a paragraph, so you need to use **ObjEndOffset-1** to get the correct location. If you are adding text to a TextLine, you use **ObjEndOffset**.



Hello my friend.§

Table 1: §

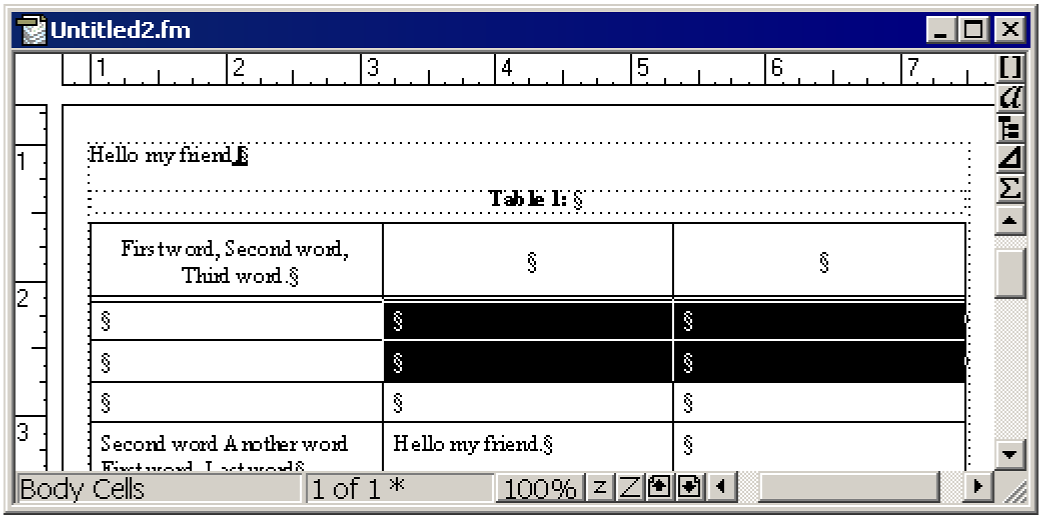
§	§	§
§	§	§
§	§	§
§	§	§
Second word Another word First word Last word§	Hello my friend.§	§

NewVar

The **New Text** command can return a text location variable if you use the **NewVar** parameter. This is the text location at the end of the text added by the **New Text** command. Here is an example.

Code Listing 5-15

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;
// Set a variable for a paragraph in the table.
Set vPgf = vTbl.FirstRowInTbl.FirstCellInRow.FirstPgf;
// Add text to the paragraph.
New Text Object(vPgf) NewVar(vTextLoc) 'First word, ';
// Add text at the end of the previous text.
New Text TextLoc(vTextLoc) NewVar(vTextLoc) 'Second word, ';
// Add text at the end of the previous text.
New Text TextLoc(vTextLoc) NewVar(vTextLoc) 'Third word.';
```



Hello my friend.§

Table 1: §

First word, Second word, Third word.§	§	§
§	§	§
§	§	§
§	§	§
Second word Another word First word Last word§	Hello my friend.§	§

Deleting Text

You use the **Delete Text** command to delete text from a text object. The **Delete Text** command uses the required **TextRange** parameter to specify which text to delete. Click in a paragraph in your document and run the following code.

Code Listing 5-16

```
// Get the paragraph object at the insertion point.
Set vPgf = TextSelection.Begin.Object;
// Make a text range of the last 4 characters in the paragraph.
New TextRange NewVar(vTextRange) Object(vPgf)
  Offset(ObjEndOffset-5) Offset(ObjEndOffset-1);
// Delete the text.
Delete Text TextRange(vTextRange);
```

The last four characters of the paragraph are deleted.

The first line of the script illustrates a way to get the text object that contains the insertion point; in this case, the **Pgf** object. This is a good technique for testing scripts that work on paragraphs. The second line shows how to make a **TextRange** that is relative to the end of the paragraph.

You can also use the Delete Text command on the currently selected text in the document.

```
Delete Text TextRange(TextSelection);
```

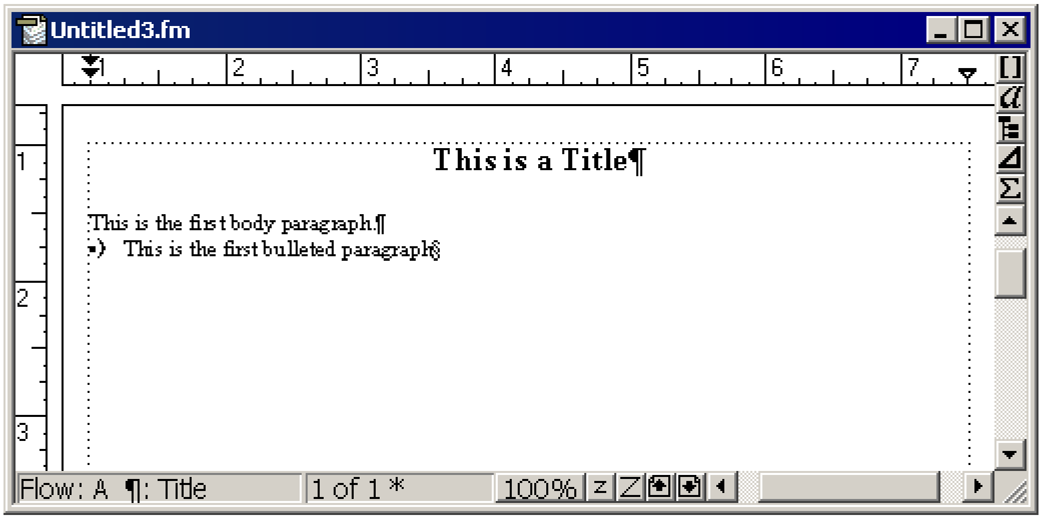
You will rarely use this, but it demonstrates that the **TextSelection** in a document is simply a **TextRange**.

Adding Text Objects

You can add **Pgf** (paragraph) and **TextLine** (TextLine) objects to a document.

New Pgf

The basic syntax for adding a paragraph is **New Pgf**. If you do not supply a location, the new paragraph is added to the beginning of the main text flow. Make a new FrameMaker document, and add three short paragraphs of text.



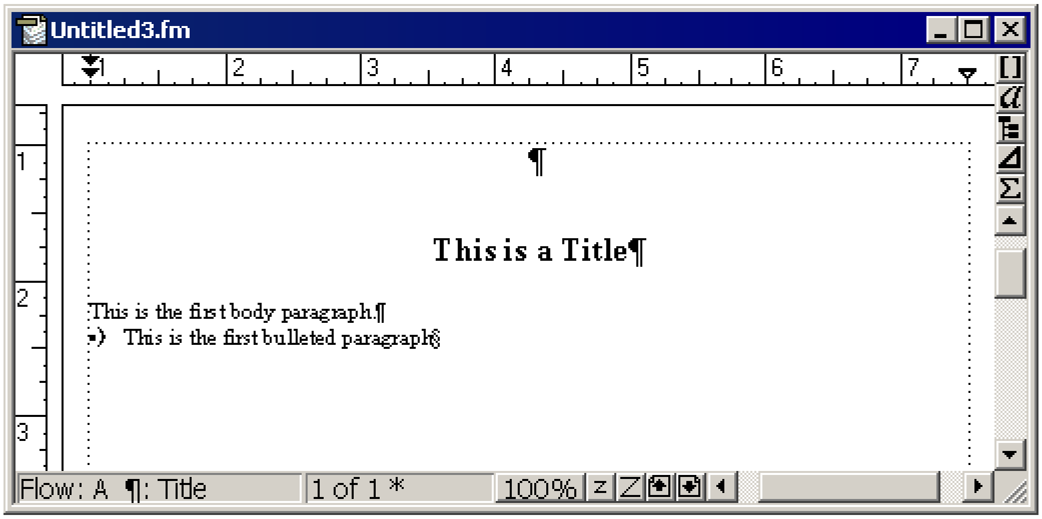
This is a Title¶

This is the first body paragraph.¶

This is the first bulleted paragraph§

Click in the second paragraph, and run the following code.

```
New Pgf;
```



¶

This is a Title¶

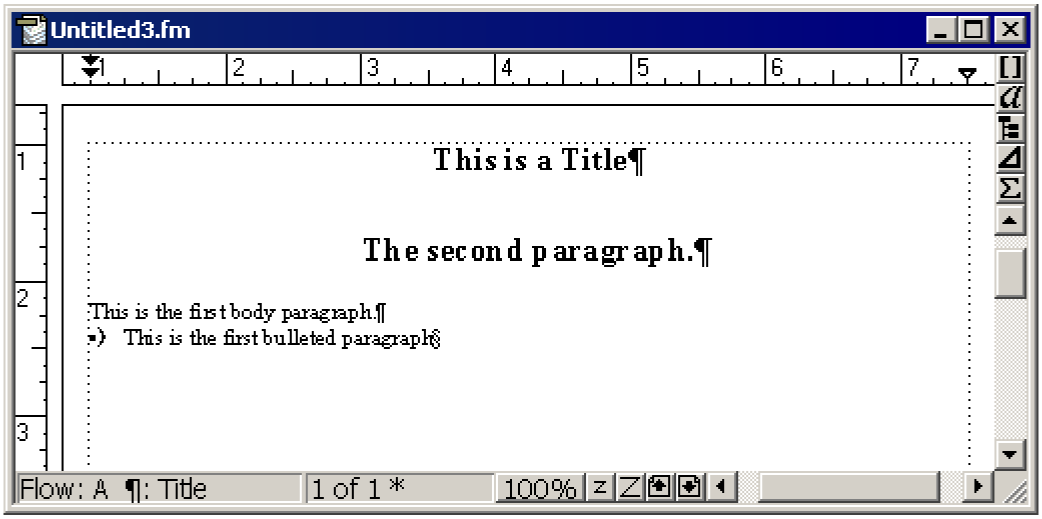
This is the first body paragraph.¶

This is the first bulleted paragraph§

A new paragraph is added at the top of the document. To specify a location for a new paragraph, use the **PrevObject** parameter of the **New Pgf** command. **PrevObject** represents the text object that comes before the new paragraph. For example, this code adds a paragraph after the first one in the document.

Code Listing 5-17

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Add a new paragraph below the first one.
New Pgf PrevObject(vPgf) NewVar(vPgf) Text('The second paragraph.');
```



This is a Title¶

The second paragraph.¶

This is the first body paragraph.¶

•) This is the first bulleted paragraph§

The **New Pgf** command has a **Text** parameter that allows you to add text to the new paragraph (FrameScript 2 and above). You can also specify the paragraph format of the new paragraph by using the **PgfFmtName** parameter.

To add a paragraph to the beginning of the document, use a **Flow** object as the **PrevObject**. The **MainFlowInDoc** is a document object representing the main flow in the document, usually flow “A.”

Code Listing 5-18

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Add a paragraph at the beginning of the main text flow.
New Pgf PrevObject(vCurrentDoc.MainFlowInDoc) NewVar(vPgf)
  Text('The first paragraph.') PgfFmtName('Heading1');
```

To add text to the paragraph after it is created, you must use the **NewVar** parameter on the **New Pgf** command and then use the **New Text** command.

Code Listing 5-19

```
// Add a paragraph.
New Pgf NewVar(vPgf);
// Add text to the paragraph.
New Text Object(vPgf) 'Text added to the new paragraph.';
```

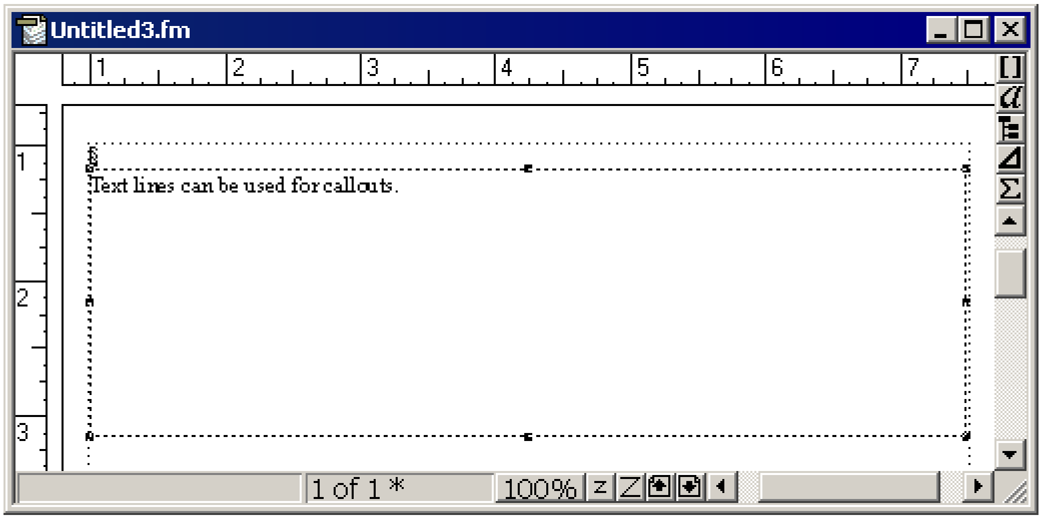
Adding Text Lines

New TextLine

A **TextLine** object is like a hybrid object—it is a graphic object that contains text. Text lines are often used for callouts in anchored frames. Make a new FrameMaker document, add an anchored frame to the document, and leave the anchored frame selected. Run the following code.

Code Listing 5-20

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first selected graphic in the document.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
// Add a text line to the selected anchored frame.
New TextLine NewVar(vTextLine) ParentObject(vAFrame) LocY(12pt);
// Add text to the text line.
New Text Object(vTextLine) 'Text lines can be used for callouts.';
```



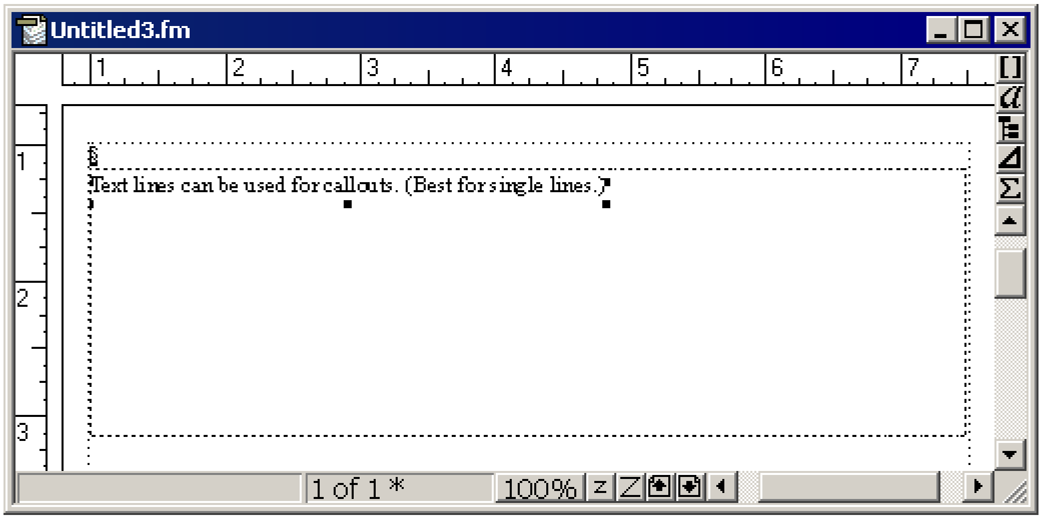
Text lines can be used for callouts.

In the second line, we set a **vAFrame** variable for the selected anchored frame. **FirstSelectedGraphicInDoc** is a document property that returns the first selected graphic in the document. If no objects are selected in the document, **FirstSelectedGraphicInDoc** returns 0 (zero) and the script will fail. **vAFrame** is used in the required **ParentObject** parameter to specify where the **TextLine** will go. We also used the optional **LocY** parameter to position the **TextLine** 12 points from the top of the anchored frame. A text line's Y location is relative to its baseline, so if **LocY** is 0 (zero), the text will be obscured by the top of its parent object.

You can add text to an existing **TextLine**, by setting a **TextLoc** variable in the **TextLine** where you want to add the text. Select the text line as an object (Control+Click) and run the following code.

Code Listing 5-21

```
// Set a variable for the selected text line.
Set vTextLine = FirstSelectedGraphicInDoc;
// Make a text location at the end of the text line.
New TextLoc NewVar(vTextLoc) Object(vTextLine)
  Offset(ObjEndOffset);
// Add text at the text location.
New Text TextLoc(vTextLoc) ' (Best for single lines.)';
```



Text lines can be used for callouts. (Best for single lines.)

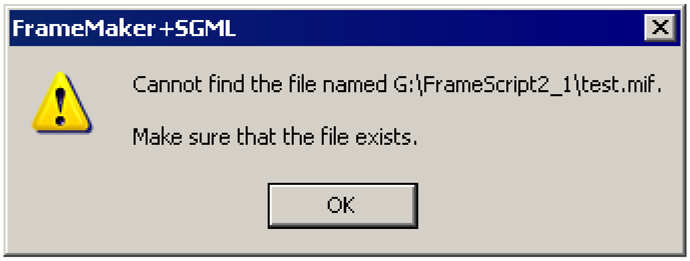
Importing Text from External Files

You can import text from external files by using the **Import File** command. The **Import File** command is also used to import graphics. (See "Imported Graphics" on page 6-9.) If you use the **Import File** command without any parameters, it will prompt you for a file as if you chose **File** > **Import** > **File** in the FrameMaker interface. You will usually use the **File** parameter to specify the file that you want to import. This script imports the FrameScript .INI file at the insertion point of the active document.

Code Listing 5-22

```
Import File File(ClientDir+DIRSEP+'fscript.ini');
```

If the file you specify in the **File** parameter does not exist, you will see a standard FrameMaker warning message. You can suppress this and other error messages by using the **AlertUserAboutFailure(False)** option.



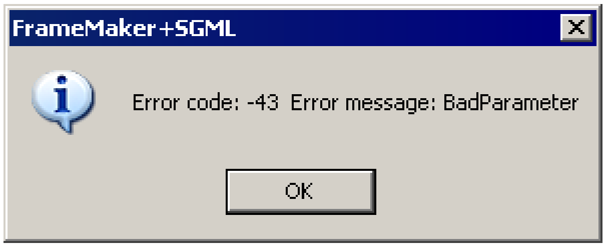
Cannot find the file named G:\FrameScript2_1\test.mif.

Make sure that the file exists.

If you use **AlertUserAboutFailure(False)**, you should trap for any errors that might occur by using the **ErrorCode** and **ErrorMsg** variables.

Code Listing 5-23

```
// Initialize ErrorCode to Success.
Set ErrorCode = 0;

// Attempt to import the file.
Import File File(ClientDir+DIRSEP+'test.mif')
  AlertUserAboutFailure(False)
  HowToImport(DoByCopy) FilterFormatId('MIF ');

// Display the error code and message.
If ErrorCode = 0
  MsgBox 'The file was successfully imported.    ';
Else
  MsgBox 'Error code: '+ErrorCode+'  Error message: '+ErrorMsg;
EndIf
```

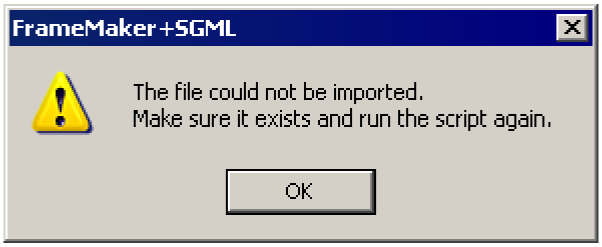


Error code: -43 Error message: BadParameter

The error messages are not always very descriptive (or helpful), so you should usually display your own message, as in the following script.

Code Listing 5-24

```
// Initialize ErrorCode to Success.
Set ErrorCode = 0;

// Attempt to import the file.
Import File File(ClientDir+DIRSEP+'test.mif')
  AlertUserAboutFailure(False)
  HowToImport(DoByCopy) FilterFormatId('MIF ');

// Display the error code and message.
If ErrorCode not= 0
  MsgBox Mode(Warn) 'The file could not be imported.    '+CHARLF+
    'Make sure it exists and run the script again.    ';
EndIf
```

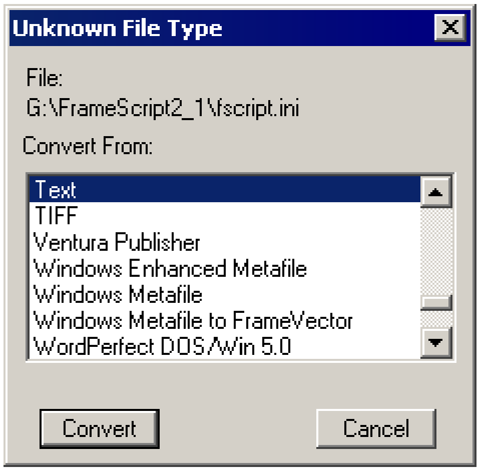


The file could not be imported.
Make sure it exists and run the script again.

The **HowToImport** parameter determines if the file will be imported by reference (the default) or imported by copy. Use **DoByRef** to import by reference and **DoByCopy** to import by copy. When you import text by reference, you are creating a text inset containing the imported text. The **NewVar** parameter will give you ID of the newly created text inset. Note that the **NewVar** parameter is only used on the **Import File** command when you import a file by reference.

You can specify a text location for the file by using the **TextLoc** parameter. See “New TextLoc” on page 5-5 of this chapter.

FrameMaker will prompt you with the Unknown File Type dialog box unless you use the **FilterFormatId** parameter that indicate what type of file you are importing.



Unknown File Type

File:
G:\FrameScript2_1\fscript.ini

Convert From:

- Text
- TIFF
- Ventura Publisher
- Windows Enhanced Metafile
- Windows Metafile
- Windows Metafile to FrameVector
- WordPerfect DOS/Win 5.0

The **FilterFormatId** parameter takes a string value. To determine the correct value, look in the [Filters] section of the maker.ini (or fmsgml.ini) file. The vendor ID is the four-character string that you need; IDs with three characters are padded at the end with a single space. Notice the padding in the example below.

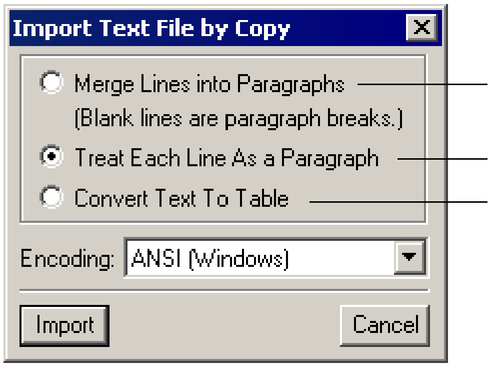
Code Listing 5-25

```
Import File File(ClientDir+DIRSEP+'test.mif')
  HowToImport(DoByCopy) FilterFormatId('MIF ');
```

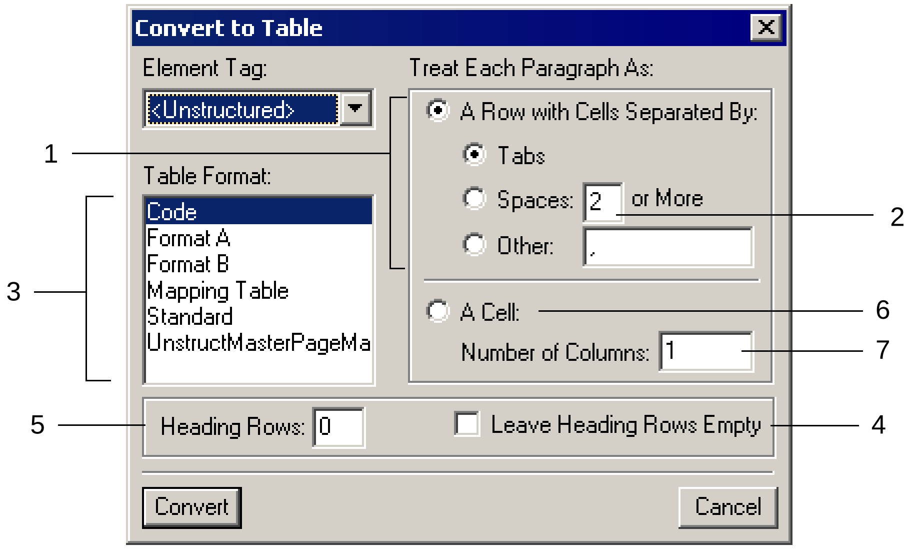
Importing Text Options

There are several important options for importing plain text files. They correspond to settings in the Import Text File dialog box in the FrameMaker interface.

FrameMaker Interface	Import File Command Option


Import Text File by Copy
○ Merge Lines into Paragraphs (Blank lines are paragraph breaks.)
● Treat Each Line As a Paragraph
○ Convert Text To Table
Encoding: ANSI (Windows)
Import   Cancel
	`FileIsText(TextFileEolIsNotEop)`
	`FileIsText(TextFileEolIsEop)`
	`FileIsText(DoImportAsTable)`
	(`TextFileEolIsEop` is the default)

If you use the **FileIsText(DoImportAsTable)** option, you can set options that correspond to the Convert To Table dialog box in the interface.



Convert to Table

Element Tag: <Unstructured>

Table Format: Code
Format A
Format B
Mapping Table
Standard
UnstructMasterPageMa

Treat Each Paragraph As:

● A Row with Cells Separated By:

● Tabs

○ Spaces: 2 or More

○ Other: ,

○ A Cell:

Number of Columns: 1

Heading Rows: 0

□ Leave Heading Rows Empty

Callout Number	Option used with FileIsText(DoImportAsTable)	Description
1	`CellSeparator`	Specifies the character used to parse the text into table cells. The default is **CHARTAB** (tab character).
2	`NumCellSeparators`	Specifies the number of spaces to require when the CellSeparator option is set to spaces (' ').
3	`ImportTblTag`	Specifies the table format for the new table. The format must exist in the document or the **Import File** command will fail.
4	`LeaveHeadingRowsEmpty`	Same as the “Leave Heading Row Empty” option in the dialog box.
5	`TblNumHeadingRows`	The number of heading rows in the new table. The default is 1.
6	`TreatParaAsRow`	Set this to **False** to convert each line in the text file into a separate cell. This is equivalent to clicking the “A Cell” radio button in the dialog box.
7	`NumColumns`	Use this with the **TreatParaAsRow(False)** option to specify the number of columns. For example, if you use **NumColumns(3)**, every third line in the text file will start a new table row.

Here is an example that imports a text file as a table at the insertion point.

Code Listing 5-26

```
Import File File('G:\FrameScript2_1\Test.txt')
  HowToImport(DoByCopy) FileIsText(DoImportAsTable)
  AlertUserAboutFailure(False) CellSeparator(CHARTAB)
  ImportTblTag('Format A') LeaveHeadingRowsEmpty(True)
  TblNumHeadingRows(1);
```

Working with Text Frames and Flows

Text frames are graphic objects that contain text. Most text in a FrameMaker document will be contained in one or more text frames. FrameMaker uses Flows to connect text frames in a document. Every document has a main flow, usually called flow A. The main flow in a document is represented by the **MainFlowInDoc** property.

Creating Text Frames

To create a text frame, use the **New TextFrame** command. You need to supply the **ParentObject** parameter, which determines the location of the new text frame. The parent object will be the frame that contains the new text frame. It can be an anchored frame or an unanchored frame, such as a **PageFrame**. A **PageFrame** is the invisible unanchored frame that surrounds every FrameMaker page. To put a text frame directly on a page, use the page’s **PageFrame** object as the **ParentObject** parameter.

For example, the following script creates a new reference page and then adds a text frame on the reference page.

Code Listing 5-27

```
// Set a variable for the active document.
Set vDoc = ActiveDoc;

// Create a new reference page.
New ReferencePage NewVar(vRefPage) Name('MyRefPage')
  DocObject(vDoc);

// Create a text frame on the reference page.
New TextFrame ParentObject(vRefPage.PageFrame)
  NewVar(vTextFrame);

// Set the location and size of the text frame.
Set vTextFrame.LocX = 1in;
Set vTextFrame.LocY = 1in;
Set vTextFrame.Width = 6.5in;
Set vTextFrame.Height = 9in;
```

Creating Flows

You do not create text flows directly; a **Flow** object is created automatically when you create a text frame. The following script can be added to the previous listing to name the flow in the text frame. The autoconnect property is turned on so that new reference pages will be created automatically as the text frames are filled.

Code Listing 5-28

```
// Give a name to the text frame's flow.
Set vTextFrame.Flow.Name = 'MyFlow';
```

```
// Set the autoconnect property to on, so that new pages
// are created automatically.
Set vTextFrame.Flow.FlowIsAutoconnect = 1;
```

Formatting Text

Text can be formatted at the paragraph level by applying **PgfFmt** (paragraph format) properties. Individual characters and words can be formatted with **CharFmt** (character format) properties, or by applying individual **TextProperties** (text properties).

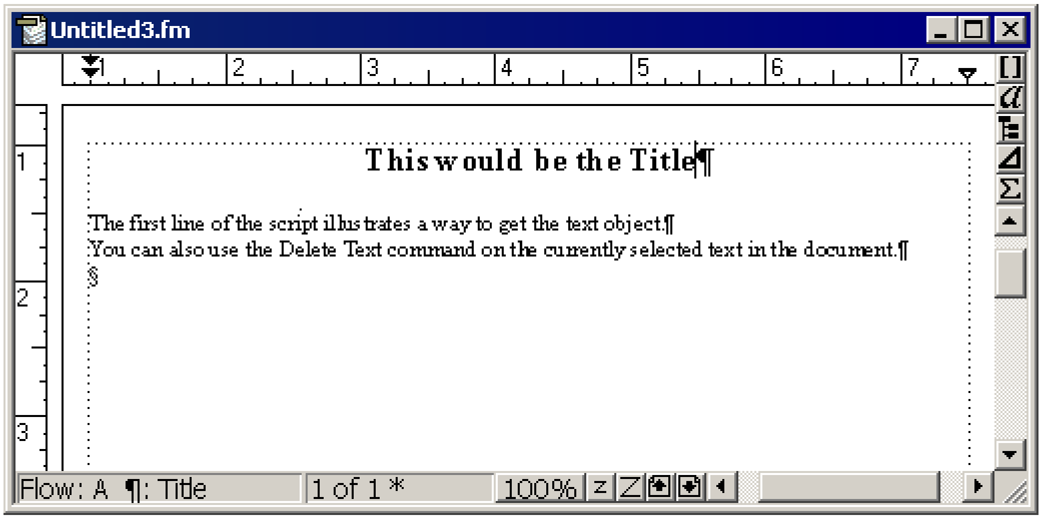
Applying Paragraph Formats

To apply a **PgfFmt** to a paragraph, you must first get the **PgfFmt** object for the document containing the paragraph. You use the **Get Object** command with the **Name** parameter. Paragraph formats are *Named Objects* so you use the **PgfFmt**'s name with the **Name** parameter. Remember that format names are case-sensitive. You must also specify the **Type** of object you are trying to get; in this case it is a **PgfFmt** object. Other possible types include **TblFmt** (table format), **RefPage** (reference page), and **XRefFmt** (cross-reference format).

Make a new document and add three short paragraphs of text using the Body paragraph format. Run the following code.

Code Listing 5-29

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Get the Title paragraph format object.
Get Object Type(PgfFmt) Name('Title') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
// Apply the paragraph format properties to the paragraph.
Set vPgf.Properties = vPgfFmt.Properties;
```



This would be the Title¶

The first line of the script illustrates a way to get the text object.¶
You can also use the Delete Text command on the currently selected text in the document.¶
§

Flow: A ¶: Title
1 of 1 *
100%

When you use the **Get Object** command, you need to test to see if the object exists. In our example, if the Title paragraph format doesn't exist in the document, the **vPgfFmt** variable will return 0 (zero) and the format will not be applied. For some objects, your script may give an error, if the object doesn't exist.

Here is a way to test to see if the object exists. To try this code, delete the Title paragraph format from the document before running it.

Code Listing 5-30

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Get the paragraph format object for the Title format.
Get Object Type(PgfFmt) Name('Title') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
If vPgfFmt
  // If the format exists, apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
Else
  // If the format doesn't exist, warn the user.
  MsgBox 'The Title paragraph format does not exist.    ';
EndIf
```

Modifying Paragraph Properties

Some properties apply to an entire paragraph, such as Alignment and Hyphenation, while others can apply to particular characters in a paragraph, such as font properties. You can change individual paragraph properties by applying them directly to the paragraph. For example, the following code changes the alignment of the paragraph containing the insertion point.

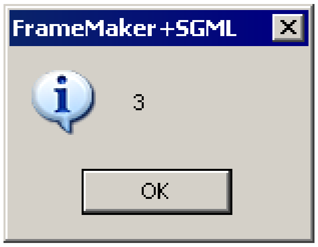
Code Listing 5-31

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
// Center-align the paragraph.
Set vPgf.PgfAlignment = PgfCenter;
```

To figure out which property to change, look up "Paragraphs" in the Object Reference section of the *FrameScript Scriptwriter's Reference*. For each property, you will find a list of possible values, like **PgfCenter** in our example. These values are usually listed as *constants* that actually represent integers. Constants are used because they are easier to remember than numbers. However, when you query a paragraph's property, FrameScript will display the integer instead of the constant. Click in a center-aligned paragraph and run the following code.

Code Listing 5-32

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
Display vPgf.PgfAlignment;
```

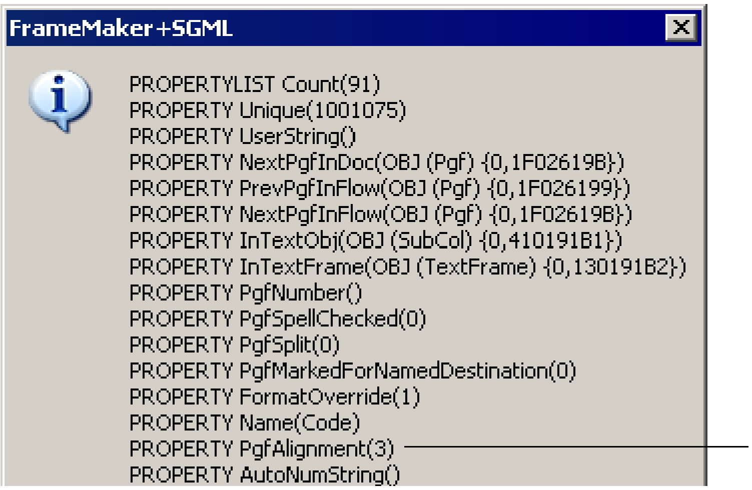


3

You will also see integers if you display a list of all the paragraph properties. Run the following code and look for the **PgfAlignment** property in the list.

Code Listing 5-33

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
Display vPgf.Properties;
```



```
PROPERTYLIST Count(91)
PROPERTY Unique(1001075)
PROPERTY UserString()
PROPERTY NextPgfInDoc(OBJ (Pgf) {0,1F02619B})
PROPERTY PrevPgfInFlow(OBJ (Pgf) {0,1F026199})
PROPERTY NextPgfInFlow(OBJ (Pgf) {0,1F02619B})
PROPERTY InTextObj(OBJ (SubCol) {0,410191B1})
PROPERTY InTextFrame(OBJ (TextFrame) {0,130191B2})
PROPERTY PgfNumber()
PROPERTY PgfSpellChecked(0)
PROPERTY PgfSplit(0)
PROPERTY PgfMarkedForNamedDestination(0)
PROPERTY FormatOverride(1)
PROPERTY Name(Code)
PROPERTY PgfAlignment(3)
PROPERTY AutoNumString()
```

Paragraph Alignment

When you set properties in your scripts, you can either use the constants or the integers that they represent.

Modifying Paragraph Formats

Note that when you set individual properties on a paragraph with a script, this can result in paragraph overrides, just like it does with the FrameMaker interface. You will usually want to modify properties of a paragraph format (**PgfFmt**), and then apply these changes to the paragraphs in the document that use that format.

To modify a paragraph format, you must first get the **PgfFmt** object and then modify the properties that you want to change. For example, suppose we want to turn off hyphenation for the Body paragraph format.

Code Listing 5-34

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Get the object for the Body paragraph format.
Get Object Type(PgfFmt) Name('Body') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
If vPgfFmt = 0
  // If the format does not exist, warn the user.
  MsgBox 'Body paragraph format does not exist in this document.';
Else
  // Turn off hyphenation on the paragraph format.
  Set vPgfFmt.Hyphenate = 0;
EndIf
```

This code only updates the properties of the **PgfFmt** object; individual Body paragraphs will still have the old hyphenation settings. We now have to loop through the Body paragraphs in the document and change the hyphenation property on each one.

Code Listing 5-35

```
// Loop through all of the paragraphs in the document.
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  // Test for a Body paragraph.
  If vPgf.Name = 'Body'
    // Turn off hyphenation.
    Set vPgf.Hyphenate = 0;
  EndIf
EndLoop
```

Using the last two blocks of code is the same as making changes in the Paragraph Designer and clicking the Update All button.

If you are making many changes to a **PgfFmt**, you may want to use the following code to apply all of the changes to the paragraphs in the document.

Code Listing 5-36

```
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  If vPgf.Name = 'Body'
    Set vPgf.Properties = vPgfFmt.Properties;
  EndIf
EndLoop
```

Instead of applying one property at a time to the Body paragraph, we simply apply all of the properties from the Body paragraph format. Note that this will remove any format overrides that are on the paragraphs. Here is the complete script with some slight modifications.

Code Listing 5-37

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Get the Body paragraph format object.
Get Object Type(PgfFmt) Name('Body') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
If vPgfFmt = 0
  // If the format doesn't exist, warn the user and exit the script.
  MsgBox 'Body paragraph format does not exist in this document.';
  LeaveSub;
EndIf
// Turn off hyphenation on the paragraph format.
Set vPgfFmt.Hyphenate = 0;

// Loop through the paragraphs in the document.
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  If vPgf.Name = 'Body'
    // If the paragraph is tagged with Body, apply the Body formats.
    Set vPgf.Properties = vPgfFmt.Properties;
  EndIf
EndLoop
```

You can eliminate the last part of the script by using the **Import Formats** command to remove paragraph format overrides. This is especially useful if you make changes to more than one paragraph format.

Code Listing 5-38

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Get the Body paragraph format object.
Get Object Type(PgfFmt) Name('Body') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
If vPgfFmt
  // Turn off hyphenation.
  Set vPgfFmt.Hyphenate = 0;
EndIf
// Get the Bulleted paragraph format object.
Get Object Type(PgfFmt) Name('Bulleted') DocObject(vCurrentDoc)
  NewVar(vPgfFmt);
If vPgfFmt
  // Turn off hyphenation.
  Set vPgfFmt.Hyphenate = 0;
EndIf

// Import formats from the current document, removing overrides.
Import Formats DocObject(vCurrentDoc) FromDocObject(vCurrentDoc)
  Pgf RemoveExceptions;
```

The script turns off hyphenation for the Body and Bulleted paragraph formats and then uses **Import Formats** to apply the changes to the paragraphs in the document. You must use the **RemoveExceptions** option, which is the same as the Remove Other Format/Layout Overrides checkbox in the Import Formats dialog box. Be aware that using this command will remove overrides from ALL of the paragraphs in your document, not just the ones you changed.

Applying Character Formatting

There are two ways to apply character-level formatting to text: by applying a character format or by applying individual text properties. Both methods require a **TextRange** object to which you want to apply the formatting. The following code applies the Emphasis character format to the selected text. Remember that the **TextSelection** property of a document is a **TextRange** object.

Code Listing 5-39

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Get the Emphasis character format object.
Get Object Type(CharFmt) Name('Emphasis') DocObject(vCurrentDoc)
  NewVar(vCharFmt);
If vCharFmt
  // Apply the character format properties to the text selection.
  Apply TextProperties TextRange(TextSelection)
    Properties(vCharFmt.Properties);
EndIf
```

To apply individual properties to a **TextRange**, you first create a **PropertyList** and then apply the **PropertyList** to the **TextRange**. You can look up the syntax under “Text properties” in the Object Reference section of the *FrameScript Scriptwriter's Reference.* Keep in mind that you will be making a local override to the text with this method; it is usually better to apply formatting with a character format.

The following example applies the Uppercase property to the selected text. Notice that you don't use the **.Properties** syntax when applying a **PropertyList**.

Code Listing 5-40

```
// Make an empty property list.
New PropertyList NewVar(vProps);
// Add the uppercase property to the list.
Add Property To(vProps) Capitalization(CapitalCaseUpper);

// Apply the property list to the selected text.
Apply TextProperties TextRange(TextSelection)
  Properties(vProps);
```

As mentioned earlier, you can use the property constants, such as **CapitalCaseUpper**, for the properties, or you can use the integers that they represent. However, when using FrameScript to view the properties, you will see the integers instead of the constants.

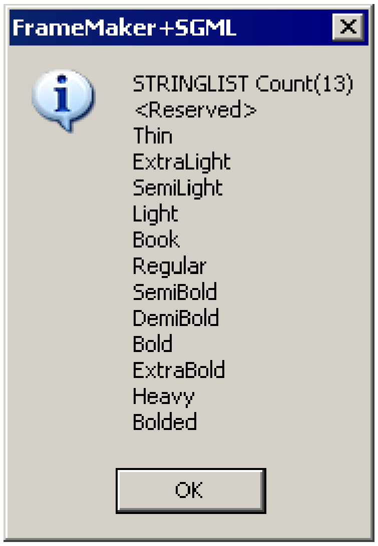
There are several text properties that do not have constants to represent them. They use an integer that indicates the position of the property value in a list of property values. For example, suppose you want to use the Bold font property. You cannot use the following code because Bold is not a valid constant for the **FontWeight** property.

```
New PropertyList NewVar(vProps);
Add Property To(vProps) FontWeight(Bold);
```

Instead, you have to find “Bold” in the list of **FontWeightNames**. This is a **Session** property that is always available when FrameMaker is running. To see the **FontWeightNames** list, run the following code.

Code Listing 5-41

```
Display FontWeightNames;
```

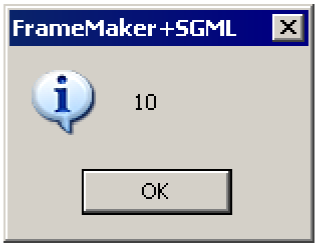


```
STRINGLIST Count(13)
<Reserved>
Thin
ExtraLight
SemiLight
Light
Book
Regular
SemiBold
DemiBold
Bold
ExtraBold
Heavy
Bolded
```

Here is the code you can use to get the correct number from the list.

Code Listing 5-42

```
Find Member('Bold') InList(FontWeightNames) ReturnPos(vPos);
Display vPos;
```

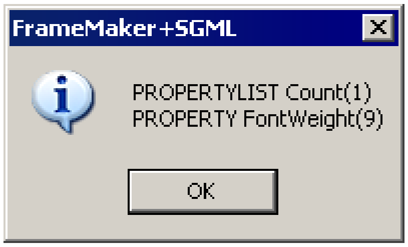


```
10
```

Actually, you need to use **vPos-1** to get the correct number because the first member (<Reserved>) does not count. If **vPos** is equal to **0** (zero), then the font weight you are testing for does not exist. Here is the completed code.

Code Listing 5-43

```
// Find Bold in the FontWeightNames list.
Find Member('Bold') InList(FontWeightNames) ReturnPos(vPos);
If vPos = 0
  // If Bold is not found, warn the user.
  MsgBox 'Bold does not exist.    ';
Else
  // Make a new property list.
  New PropertyList NewVar(vProps);
  // Add Bold to the property list.
  Add Property To(vProps) FontWeight(vPos-1);
EndIf
Display vProps;
```



PROPERTYLIST Count(1)
PROPERTY FontWeight(9)

In our example, Bold is represented by the integer 9. However, that does not mean that 9 will always represent Bold for all FrameMaker installations. You should always check the **FontWeightNames** property for the correct integer value.

Technically, someone could test for the <Reserved> member in the list. This would result in **vPos** being greater than zero, but would add an invalid property to **vProps**. This is not likely to happen, but here is a safer version of the script that accounts for this.

Code Listing 5-44

```
Find Member('<Reserved>') InList(FontWeightNames) ReturnPos(vPos);
If vPos <= 1
  MsgBox 'Invalid font weight name.    ';
Else
  New PropertyList NewVar(vProps);
  Add Property To(vProps) FontWeight(vPos-1);
EndIf
```

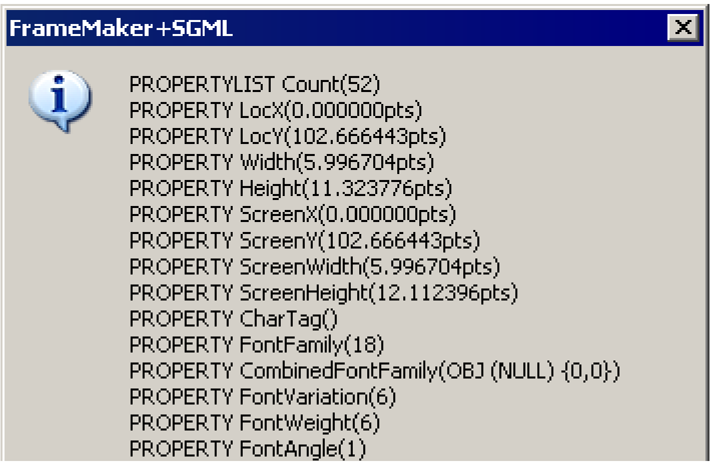
Font names are stored in the **Session**'s **FontFamilyNames** property, and font angles (such as Italic) are stored in the **Session**'s **FontAngleNames** property.

Getting Text Properties

You can get a list of text properties at any valid text location in your FrameMaker document. Since the text in a **TextRange** (including the **TextSelection**) can have different properties applied to each of its characters, you test for text properties at a **TextLoc** object. Properties are always tested at the character to the right of the **TextLoc**. If you have an insertion point in a document, the character to the right of the insertion point is tested. This code will display all of the properties at the beginning of the selected text or insertion point.

Code Listing 5-45

```
Display TextSelection.Begin.Properties;
```

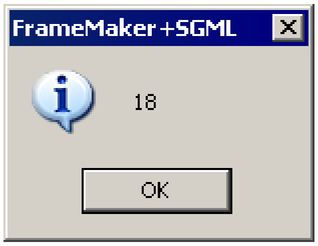


```
PROPERTYLIST Count(52)
PROPERTY LocX(0.000000pts)
PROPERTY LocY(102.666443pts)
PROPERTY Width(5.996704pts)
PROPERTY Height(11.323776pts)
PROPERTY ScreenX(0.000000pts)
PROPERTY ScreenY(102.666443pts)
PROPERTY ScreenWidth(5.996704pts)
PROPERTY ScreenHeight(12.112396pts)
PROPERTY CharTag()
PROPERTY FontFamily(18)
PROPERTY CombinedFontFamily(OBJ (NULL) {0,0})
PROPERTY FontVariation(6)
PROPERTY FontWeight(6)
PROPERTY FontAngle(1)
```

You can also display individual properties.

Code Listing 5-46

```
Display TextSelection.Begin.FontFamily;
```

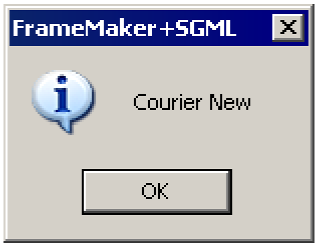


18

For many properties, FrameScript reports property integers instead of constants. For properties that have values stored in a Session property list, you can determine the property name. To see which properties have values stored in a Session property list, see “Session” in the Object Reference section of the *FrameScript Scriptwriter’s Reference*. Here is a way to see what Font name corresponds with a FontFamily integer.

Code Listing 5-47

```
Set vPos = TextSelection.Begin.FontFamily;
Get Member Number(vPos+1) From(FontFamilyNames)
  NewVar(vFontFamilyName);
Display vFontFamilyName;
```



Courier New

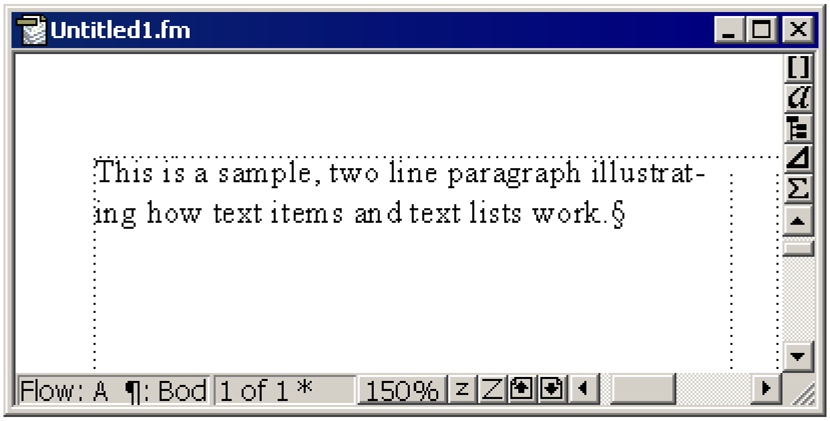
Working With Text Lists and Text Items

Text items (**TextItem**) are objects that can be contained in FrameMaker text, including text strings, markers, frame and table anchors, variables, and cross-references. There are also “invisible” text items indicating the beginning and end of paragraphs, lines, subcolumns, and text frames. There is another kind of text item that marks where character property changes occur in a text object.

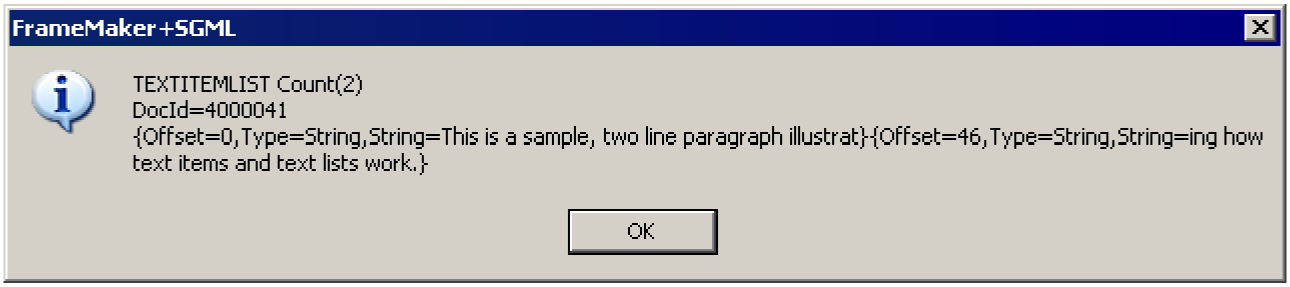
You can get a list of text items in a text object or text range by using the **Get TextList** command. A **TextList** is a list of text items. To see this, type a two line paragraph in a FrameMaker document, and put the cursor in the paragraph. Run the following code.

Code Listing 5-48

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of strings in the paragraph.
Get TextList InObject(vPgf) NewVar(vTextList) String;

// Display the list.
Display vTextList;
```

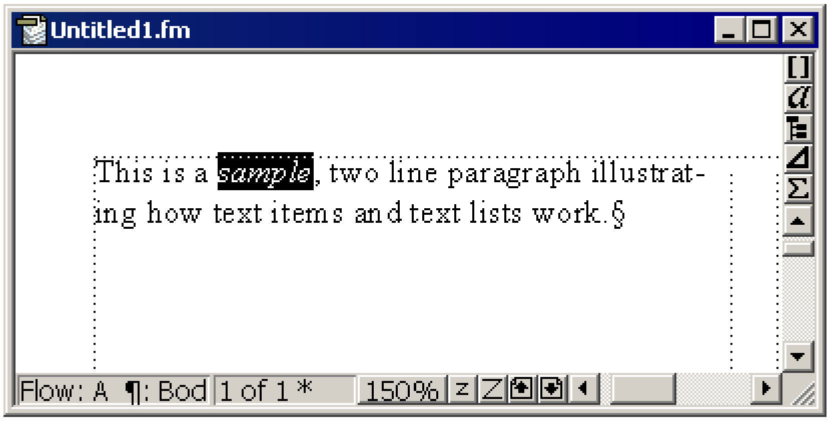


This is a sample, two line paragraph illustrat-
ing how text items and text lists work.§

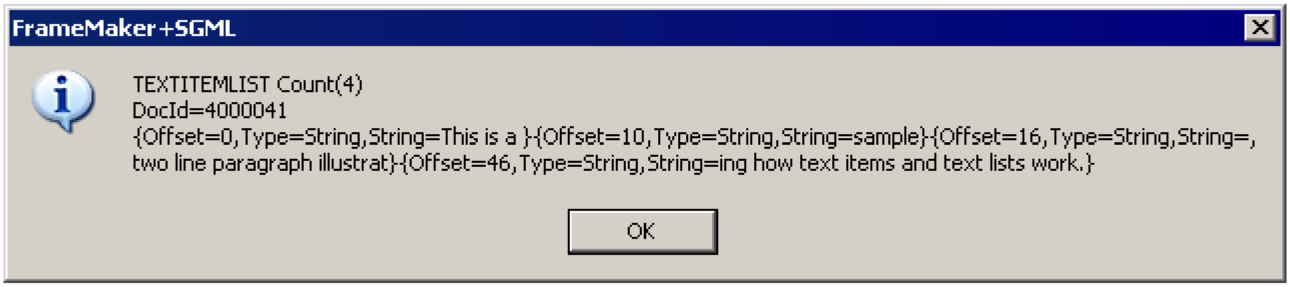


```
TEXTITEMLIST Count(2)
DocId=4000041
{Offset=0,Type=String,String=This is a sample, two line paragraph illustrat}{Offset=46,Type=String,String=ing how
text items and text lists work.}
```

When you use the **Get TextList** command, you add the text items that you want to return in the list; in the above example, we used **String** to return strings. Make one of the words in the paragraph italic, and rerun the script.



This is a *sample*, two line paragraph illustrat-
ing how text items and text lists work.§



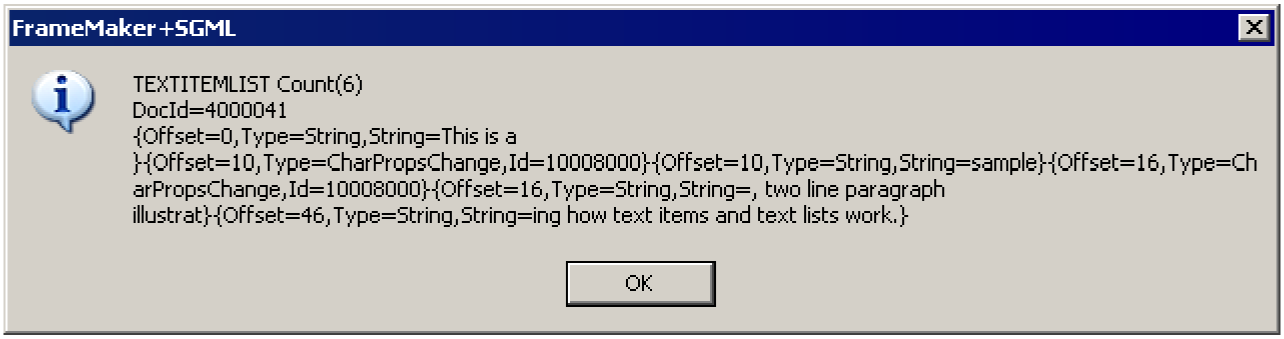
```
TEXTITEMLIST Count(4)
DocId=4000041
{Offset=0,Type=String,String=This is a }{Offset=10,Type=String,String=sample}{Offset=16,Type=String,String=,
two line paragraph illustrat}{Offset=46,Type=String,String=ing how text items and text lists work.}
```

If you look carefully, you can see that the first line now contains three strings, with the italicized word being a separate string. Any new character property change or new line will make a separate text item. It is important to see that text items in a text list are always in the order that they appear in the text object, in this case, the paragraph.

Add the **CharPropsChange** (character property change) option to the **Get TextList** command and run it on the sample paragraph. A **CharPropsChange** text item occurs everytime a text property changes in the text object.

Code Listing 5-49

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of strings and character property changes.
Get TextList InObject(vPgf) NewVar(vTextList) String
  CharPropsChange;

// Display the list.
Display vTextList;
```



```
TEXTITEMLIST Count(6)
DocId=4000041
{Offset=0,Type=String,String=This is a
}{Offset=10,Type=CharPropsChange,Id=10008000}{Offset=10,Type=String,String=sample}{Offset=16,Type=Ch
arPropsChange,Id=10008000}{Offset=16,Type=String,String=, two line paragraph
illustrat}{Offset=46,Type=String,String=ing how text items and text lists work.}
```

You should see that the **CharPropsChange** text item has an offset of 10, which is the same offset as the second **String** text item. The second **String** item is the italicized word.

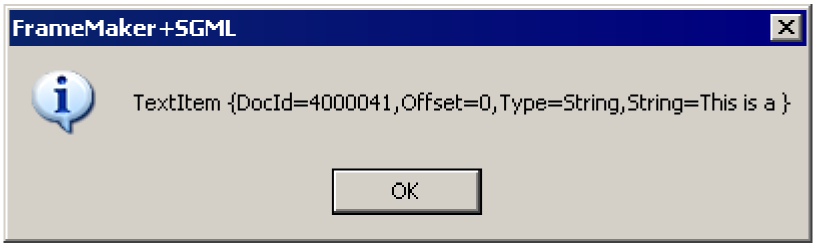
NOTE: Be careful when typing text item names. If you misspell a name, you may not get an error, but the text items you are expecting will not be returned in the text list.

Text lists displayed this way can be hard to read. You will normally work with each text item one at a time so you will have to get them out of the **TextList**. You do this by using the **Get Member** command.

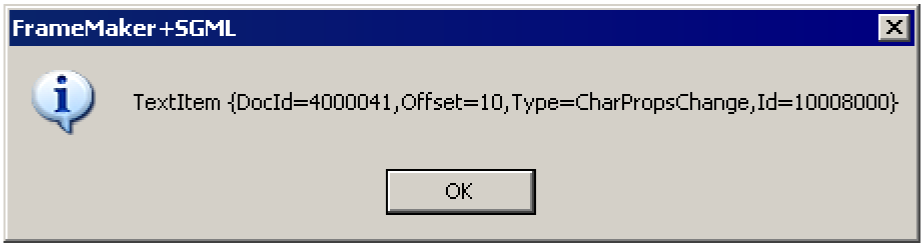
Code Listing 5-50

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of strings and character property changes.
Get TextList InObject(vPgf) NewVar(vTextList) String
  CharPropsChange;

// Get each member of the list using a loop.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  // Display the text item.
  Display vTextItem;
EndLoop
```

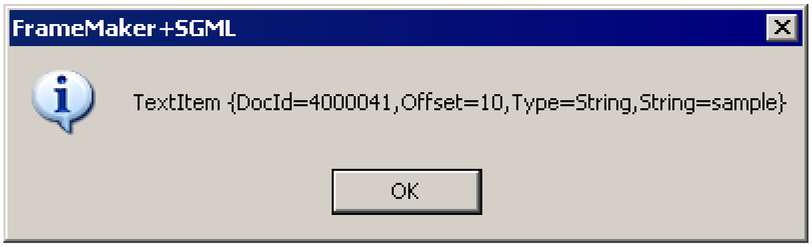
Here are screenshots of the first three text items.



TextItem {DocId=4000041,Offset=0,Type=String,String=This is a }



TextItem {DocId=4000041,Offset=10,Type=CharPropsChange,Id=10008000}



TextItem {DocId=4000041,Offset=10,Type=String,String=sample}

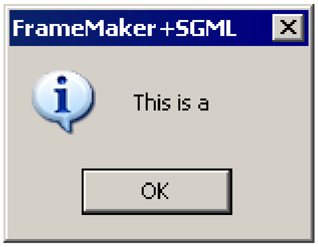
TextItem objects have three properties: **TextData**, **TextOffset**, and **TextType**. The **TextOffset** property gives the offset of the text item from the beginning of the text object. The **TextType** property simply returns the **TextItem** name that you specified in the **Get TextList** command; for example **String** or **CharPropsChange**.

The **TextData** property varies depending on what kind of **TextItem** it is used with. For a **String** text item, **TextData** returns the string. For a **CharPropsChange** text item, **TextData** returns a string list of the text properties that have changed. Try the script below on the sample paragraph.

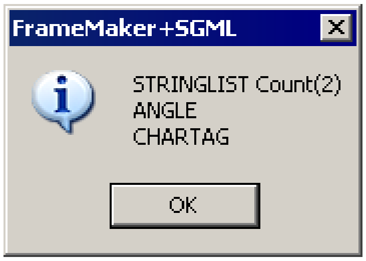
Code Listing 5-51

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of strings and character property changes.
Get TextList InObject(vPgf) NewVar(vTextList) String
  CharPropsChange;

// Get each member of the list using a loop.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  // Display the text data property.
  Display vTextItem.TextData;
EndLoop
```

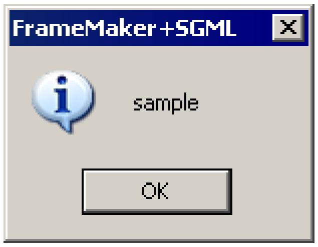
Here are the first three text items showing the **TextData** property.



This is a



STRINGLIST Count(2)
ANGLE
CHARTAG



sample

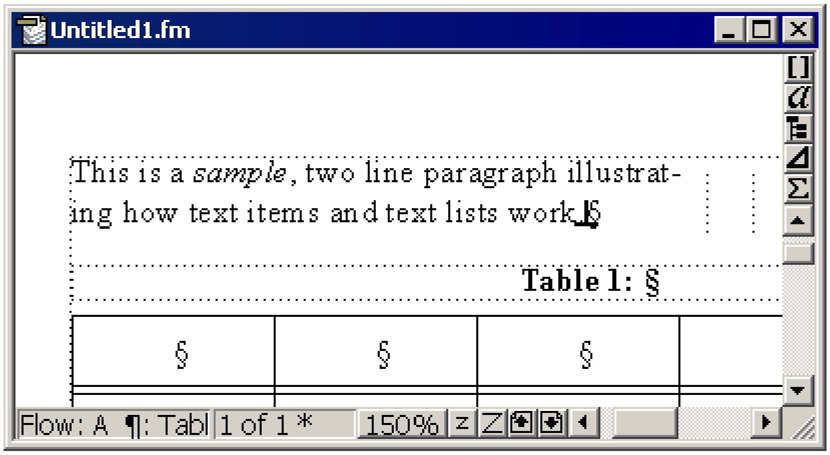
Getting the Object of a Text Item

The **TextData** property returns the object of certain text item types. For example, if you get a list of **TblAnchor** text items, each **TblAnchor**'s **TextData** property will return the **Tbl** object associated with the table. To see this, insert a table in the sample paragraph, and run the following script.

Code Listing 5-52

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of table anchors in the paragraph.
Get TextList InObject(vPgf) NewVar(vTextList) TblAnchor;
```

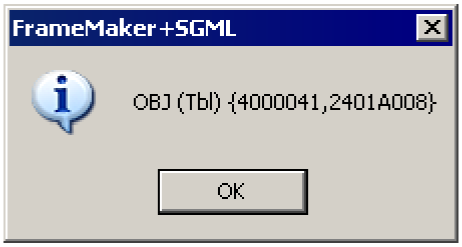
```
// Get each member of the list using a loop.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  // Display the text data property, which is the table object.
  Display vTextItem.TextData;
EndLoop
```



This is a *sample*, two line paragraph illustrating how text items and text lists work.§

Table 1: §

§ § §



OBJ (Tbl) {4000041,2401A008}

You can see that this is useful way of getting objects in document order. The following script gets a list of all of the markers in the main text flow. Notice that the **InObject** parameter contains the **MainFlowInDoc** object instead of a **Pgf** object.

Code Listing 5-53

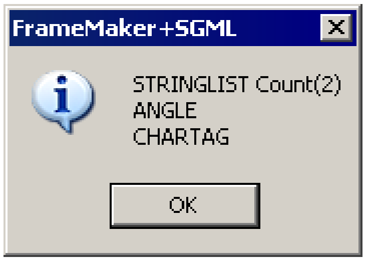
```
// Set a variable for the current document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the main document flow.
Set vMainFlow = vCurrentDoc.MainFlowInDoc;

// Get a list of marker anchors in the main document flow.
Get TextList InObject(vMainFlow) NewVar(vTextList) MarkerAnchor;

// Process the list of markers.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  // Get the marker anchor text item.
  Get Member Number(n) From(vTextList) NewVar(vMarkerAnchor);
  // Set a variable for the marker object.
  Set vMarker = vMarkerAnchor.TextData;
  // Do something with the marker.
  // ...
EndLoop
```

Working With Character Property Changes

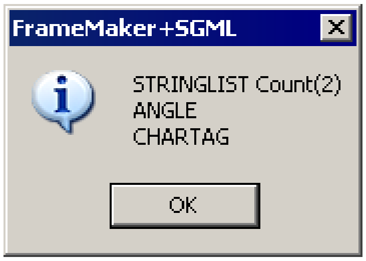
As you saw earlier, you can get a list of character property changes in a text object. The **TextData** property of a **CharPropsChange** is a string list containing the properties that have changed at that point in the text object. It does not tell you *how* they have changed. To see this, format a single word in the middle of a paragraph with the Emphasis character format. Click in the paragraph and run the following code.

Code Listing 5-54

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of character property changes in the paragraph.
Get TextList InObject(vPgf) NewVar(vTextList) CharPropsChange;

// Get each member of the list using a loop.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  // Display the text data property.
  Display vTextItem.TextData;
EndLoop
```



STRINGLIST Count(2)
ANGLE
CHARTAG

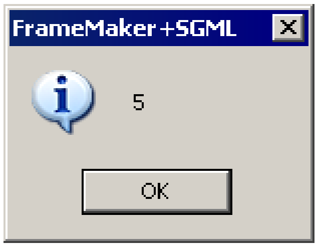


STRINGLIST Count(2)
ANGLE
CHARTAG

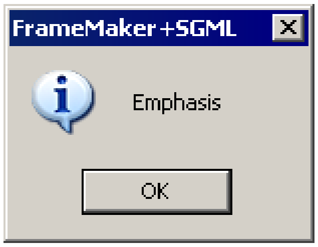
The same message box appears twice, once for each change. You can see that the character tag and angle have changed at both locations; to determine the actual property change, you need to make a text location at each **CharPropsChange** location and query the appropriate **TextLoc** properties.

Code Listing 5-55

```
// Make a variable for the paragraph at the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a text list of character property changes in the paragraph.
Get TextList InObject(vPgf) NewVar(vTextList) CharPropsChange;

// Get each member of the list using a loop.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  // Make a text location.
  New TextLoc NewVar(vTextLoc) Object(vPgf)
    Offset(vTextItem.TextOffset);
  // Display the angle and character tag properties.
  Display vTextLoc.FontAngle;
  Display vTextLoc.CharTag;
EndLoop
```

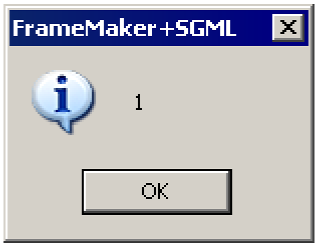


5

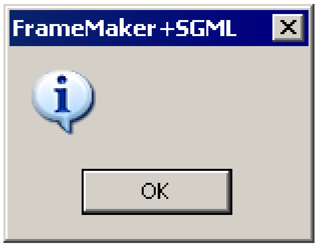


Emphasis

The first **CharPropsChange** shows the angle set to 5, which is the integer for Italic, while the character tag is set to Emphasis.



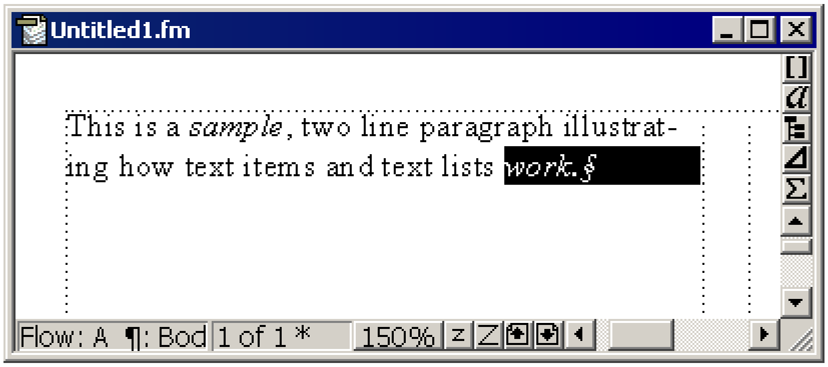
1



The second **CharPropsChange** shows the angle set to 1, which is the integer for Regular, while the character tag is set to an empty string, signifying a switch back to the default paragraph font.

Getting the Text Range of a Character Property Change

CharPropsChange text items mark the points where character properties change in the text object. You will sometimes need to locate the text range of changed text in a paragraph. To do this, you will have to retrieve the **TextRange** that is between two **CharPropChange** text items. This is easier to understand by working through an example. Make a new FrameMaker document and add a short paragraph. Italicize a word near the middle of the paragraph, using **Format** > **Style** > **Italic**. Italicize the last word in the paragraph, including any punctuation, using **Format** > **Style** > **Italic**.



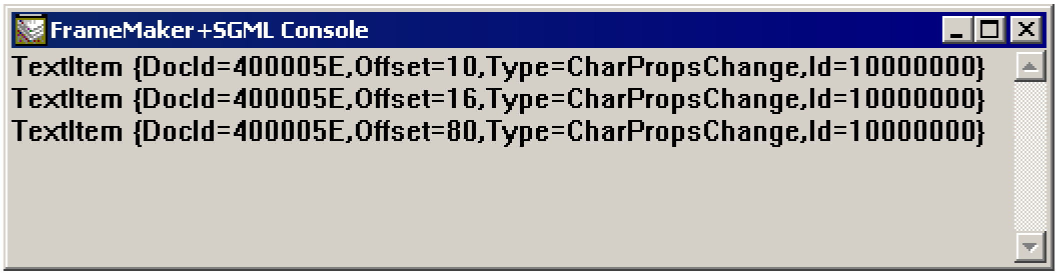
This is a *sample*, two line paragraph illustrating how text items and text lists *work.*§

We are using local formatting to make the text italic, instead of using a character format. Suppose we want to find the italicized text and apply the Emphasis character format to it. The script below shows how to get the **CharPropsChange** text items for the paragraph containing the insertion point.

Code Listing 5-56

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a list of character property changes.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);
// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // Write the text item to the Console.
  Write Console vCharPropsChange;
EndLoop
```



```
TextItem {DocId=400005E,Offset=10,Type=CharPropsChange,Id=10000000}
TextItem {DocId=400005E,Offset=16,Type=CharPropsChange,Id=10000000}
TextItem {DocId=400005E,Offset=80,Type=CharPropsChange,Id=10000000}
```

The Console window shows the text items, although the numbers will different for each document. In order to get the text ranges of each change, we will have to test each text location where the property changes and store the location. Once we have an "on" and "off" location pair, we can make a text range.

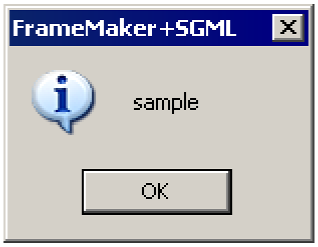
To test the italic property, we have to compare it to the correct integer value from the Session list of font angle names. See "Getting Text Properties" on page 5-27 for more information.

Code Listing 5-57

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the Italic integer from the session list.
Find Member('Italic') InList(FontAngleNames) ReturnPos(vPos);
Set vItalicInt = vPos-1;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Initialize variables to store the property changes.
Set vStartOffset = 0;
Set vItalicsOn = 0;
```

```
// Get a list of character property changes.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);
// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // Set a text location at the property change location.
  New TextLoc NewVar(vTextLoc) Object(vPgf)
    Offset(vCharPropsChange.TextOffset);
  // Test the italics value at the text location.
  If vTextLoc.FontAngle = vItalicInt // Italics is "on"
    // Store the offset.
    Set vStartOffset = vCharPropsChange.TextOffset;
    // Set the "on" variable.
    Set vItalicsOn = 1;
  EndIf
  // See if italics is "off" and the vItalicsOn variable is 1
  If (vTextLoc.FontAngle = 1) and (vItalicsOn = 1)
    // This means that italics was on, but it just switched to off.
    // Make a text range.
    New TextRange NewVar(vTextRange) Object(vPgf)
      Offset(vStartOffset) Offset(vCharPropsChange.TextOffset);
    // Display the text range text.
    Display vTextRange.Text;
    // Reset the italics on variable.
    Set vItalicsOn = 0;
  EndIf
EndLoop
```

When you run the script, you will see the first italicized word appear.

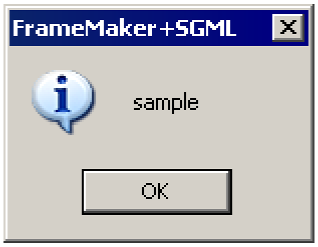


sample

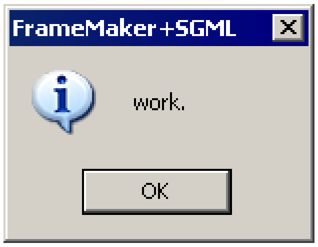
You don't see the second italicized word because it occurs at the end of the paragraph. A **CharPropsChange** text item never occurs at the end of a text object. We must test the **vItalicsOn** variable after all of the items in the text list have been processed. Add this code to the end of the last script and run it with your cursor in the sample paragraph.

Code Listing 5-58

```
// Test the end of the paragraph to see if italics is on.
If vItalicsOn = 1;
  // Make a text range up to the end of the paragraph.
  New TextRange NewVar(vTextRange) Object(vPgf)
    Offset(vStartOffset) Offset(ObjEndOffset);
  // Display the text range text.
  Display vTextRange.Text;
EndIf
```



sample



work.

If your code is correct, you will now see both italicized strings listed. Now we can format them with the Emphasis character format. Here is the completed script.

Code Listing 5-59

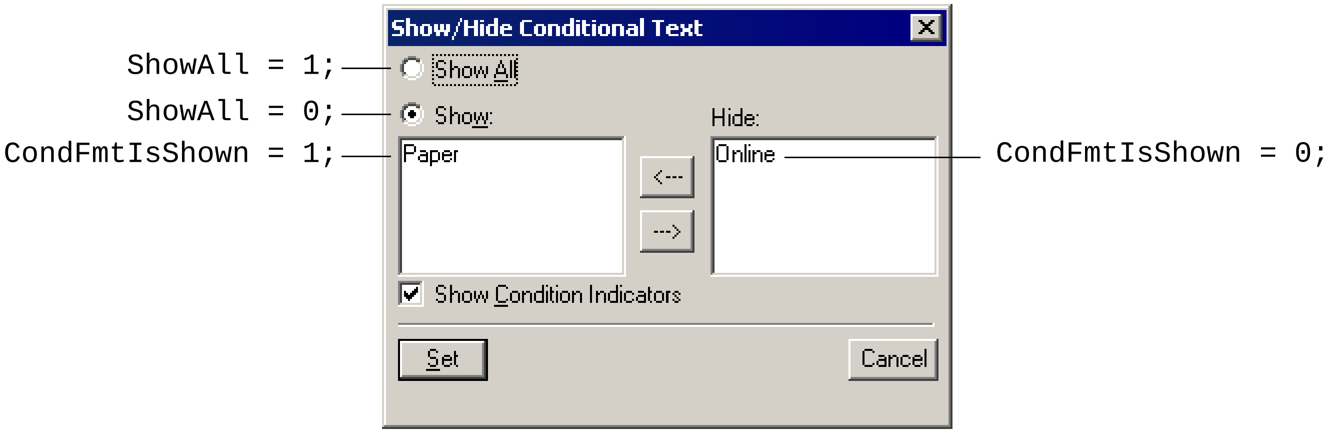
```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the Italic integer from the session list.
Find Member('Italic') InList(FontAngleNames) ReturnPos(vPos);
Set vItalicInt = vPos-1;

// Get the Emphasis character format object.
Get Object Type(CharFmt) Name('Emphasis') DocObject(vCurrentDoc)
  NewVar(vCharFmt);
// If the Emphasis character format does not exist, exit the script.
If vCharFmt = 0
  MsgBox 'Emphasis character format does not exist.    ';
  LeaveSub;
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Initialize variables to store the property changes.
Set vStartOffset = 0;
Set vItalicsOn = 0;
```

```
// Get a list of character property changes.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);
// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // Set a text location at the property change location.
  New TextLoc NewVar(vTextLoc) Object(vPgf)
    Offset(vCharPropsChange.TextOffset);
  // Test the italics value at the text location.
  If vTextLoc.FontAngle = vItalicInt // Italics is "on"
    // Store the offset.
    Set vStartOffset = vCharPropsChange.TextOffset;
    // Set the "on" variable.
    Set vItalicsOn = 1;
  EndIf
  // See if italics is "off" and the vItalicsOn variable is 1
  If (vTextLoc.FontAngle = 1) and (vItalicsOn = 1)
    // This means that italics was on, but it just switched to off.
    // Make a text range.
    New TextRange NewVar(vTextRange) Object(vPgf)
      Offset(vStartOffset) Offset(vCharPropsChange.TextOffset);
    // Apply the Emphasis character format to the text.
    Apply TextProperties TextRange(vTextRange)
      Properties(vCharFmt.Properties);
    // Reset the italics on variable.
    Set vItalicsOn = 0;
  EndIf
EndLoop

// Test the end of the paragraph to see if italics is on.
If vItalicsOn = 1;
  // Make a text range up to the end of the paragraph.
  New TextRange NewVar(vTextRange) Object(vPgf)
    Offset(vStartOffset) Offset(ObjEndOffset);
  // Apply the Emphasis character format to the text.
  Apply TextProperties TextRange(vTextRange)
    Properties(vCharFmt.Properties);
EndIf
```

Conditional Text

Showing and Hiding Conditions

Each condition format in a FrameMaker document has a **CondFmtIsShown** property that determines the visibility of the condition in the document. This property must be used in conjunction with the document's **ShowAll** property. If the document's **ShowAll** property is set to **1** (or **True**), all conditions will be shown, regardless of their **CondFmtIsShown** value. Therefore, to hide a condition, set the document's **ShowAll** property to **0** (or **False**), and the condition's **CondFmtIsShown** property to **0**. The screenshot below shows how each FrameScript property corresponds to settings in the FrameMaker interface.



Show/Hide Conditional Text

○ Show All

● Show:

Paper

Hide:

Online

☑ Show Condition Indicators

The following script hides the Online condition format in the active document.

Code Listing 5-60

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set the document's ShowAll property to 0.
Set vCurrentDoc.ShowAll = 0;

// Get the condition format object.
Get Object Type(CondFmt) Name('Online') DocObject(vCurrentDoc)
  NewVar(vCondFmt);

// Hide the condition format.
Set vCondFmt.CondFmtIsShown = 0;
```

Remember that **ShowAll** is a **Doc** property, while **CondFmtIsShown** is a **CondFmt** property.

Showing and Hiding Condition Indicators

Condition formats can have colors and formatting to distinguish them in a document; these are know as "condition indicators." You can show or hide condition indicators by setting the document's **ShowCondIndicators** property. This is the same as checking or unchecking the Show Condition Indicators checkbox in the Show/Hide Conditional Text dialog box (see previous screenshot). This script shows the condition indicators in the active document.

Code Listing 5-61

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Show the condition indicators in the document.
Set vCurrentDoc.ShowCondIndicators = 1;
```

Applying Condition Formats

You can use the **Apply TextProperties** command to apply a **CondFmt** to a range of text. Use the **CondFmt** parameter with the name of the condition format. The following code applies the "Online" condition to the paragraph containing the insertion point.

Code Listing 5-62

```
// Make a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Make a text range for the entire paragraph.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0)
  Offset(ObjEndOffset);

// Apply the Online condition format to the text range.
Apply TextProperties CondFmt('Online') TextRange(vTextRange);
```

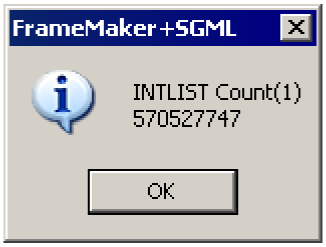
You can use the **CondFmt** parameter more than once to apply multiple conditions to a text range. If you use a null string on the **CondFmt** parameter, all condition formats will be removed from the text range.

The **Apply TextProperties** command is limited in that it does not preserve any existing conditions that may be applied to the text range. For example, if you have “Condition1” applied to a range of text and use the **Apply TextProperties** command to apply “Condition2” to the same text, it will remove the “Condition1” format from the text.

You can overcome this limitation by manipulating the text’s **InCond** property. The **InCond** property is an integer list of condition format integers that are applied to text. Table rows also have an **InCond** property that indicates which conditions are applied to the row. To see the **InCond** property, apply a condition format to a word in a document, select the word and run this code.

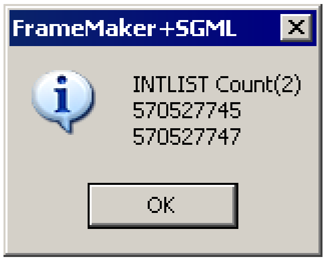
Code Listing 5-63

```
// See the InCond property of the selected text.
Display TextSelection.Begin.InCond;
```



```
INTLIST Count(1)
570527747
```

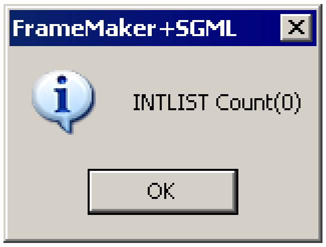
Now apply a second condition format to the word and rerun the code.



```
INTLIST Count(2)
570527745
570527747
```

The numbers you will see will differ from those in the screenshots. These numbers are integer representations of **CondFmt** objects. You will see how to identify the condition formats represented by these numbers shortly.

If you run the code on text that has no condition formats applied, the **InCond** integer list will be **0** (empty).

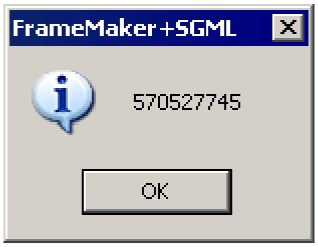


INTLIST Count(0)

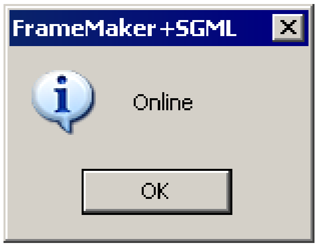
To convert the integers to objects, you must first extract them from the integer list. Then you use the **New Object** command to convert each integer to a **CondFmt** object. Select the text that is formatted with two conditions and run the following script.

Code Listing 5-64

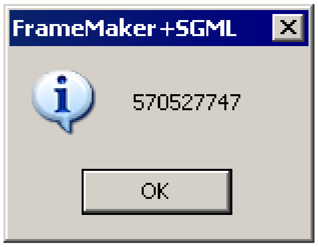
```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the InCond list at the text selection.
Set vInCond = TextSelection.Begin.InCond;

// Loop through the InCond list of the selected text.
Loop While(n <= vInCond.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vInCond) NewVar(vCondInt);
  Display vCondInt; // Display the integer.
  // Make an condition format object from the integer.
  New Object IntValue(vCondInt) DocObject(vCurrentDoc)
    NewVar(vCondFmt);
  Display vCondFmt.Name; // Display the condition format name.
EndLoop
```



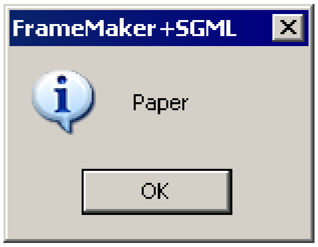
570527745



Online



570527747



Paper

You can use this method to identify condition formats that are applied at a text location. You can also use this with a **Row**'s **InCond** property to identify conditions that are applied to a table row.

To apply conditions to text or table rows, you work in the opposite direction; you start with the **CondFmt** object, convert it to an integer, add it to an integer list, and apply the list to the **InCond** property. Here is a script that illustrates this by applying a single condition to the selected text in the active document. *The following two example scripts do not attempt to preserve existing conditions that may be applied to the text or table rows.* See "Preserving Existing Conditions" on page 5-44.

Code Listing 5-65

```
// Make a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make an integer list.
New IntList NewVar(vInCond);

// Get the Online condition format object.
Get Object Type(CondFmt) Name('Online') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
// Convert the condition format object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);

// Add the integer to the integer list.
Add Member(vCondInt) To(vInCond);

// Make a property list containing the vInCond list.
New PropertyList NewVar(vProps) InCond(vInCond);

// Apply the properties to the text selection.
Apply TextProperties TextRange(TextSelection) Properties(vProps);
```

For table rows, you use the **Row**'s **InCond** property to apply conditions to the row. The following example, applies two conditions to the last row in the selected table.

Code Listing 5-66

```
// Make a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;

// Make an integer list.
New IntList NewVar(vInCond);

// Get the PaperUS condition object.
Get Object Type(CondFmt) Name('PaperUS') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
// Convert the condition format object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);
// Add the integer to the integer list.
Add Member(vCondInt) To(vInCond);
```

```
// Get the PaperUK condition object.
Get Object Type(CondFmt) Name('PaperUK') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
// Convert the condition format object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);
// Add the integer to the integer list.
Add Member(vCondInt) To(vInCond);

// Apply the  list to the last row in the table.
Set vTbl.LastRowInTbl.InCond = vInCond;
```

IMPORTANT: The previous code examples leave out some important tests: there is no test for an active document, a selected table, and we did not test for the existence of the the condition formats. Make sure you have sufficient testing in your production scripts.

Preserving Existing Conditions

The previous two examples used the **InCond** property to apply conditions to text and table rows. *However, existing conditions would be overwritten using those scripts.* To preserve existing conditions, you must add the new conditions to the existing ones. We will demonstrate this first with table rows. If you want to try this, follow these steps.

1. Make a new FrameMaker document.
2. Create two new conditions, Condition1 and Condition2. Make sure you assign a different color to each condition.
3. Insert a Format A table.
4. Apply Condition1 to the first row in the table.
5. Click in the table and run the following script.

Code Listing 5-67

```
// Make a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the Condition2 object.
Get Object Type(CondFmt) Name('Condition2') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
If vCondFmt = 0
  // If the condition doesn't exit, warn the user and exit.
  MsgBox 'Condition2 format does not exist in this document.    ';
  LeaveSub;
EndIf

// Convert the condition object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);

// Set a variable for the selected table.
Set vTbl = vCurrentDoc.SelectedTbl;

// Set a variable for the first row in the table.
Set vRow = vTbl.FirstRowInTbl;

// Make a variable for the current conditions in the row.
Set vInCond = vRow.InCond;
```

```
// See if the Condition2 integer is already in the list.
Find Member(vCondInt) InList(vInCond) ReturnStatus(vFound);
If vFound = 0
  // If the integer is not already in the list, add it.
  Add Member(vCondInt) To(vInCond);
EndIf

// Apply the list to the table row.
Set vRow.InCond = vInCond;
```

You should see Condition2 applied to the table row in addition to the existing Condition1. If you get an error, make sure your cursor is in the table and run the script again.

Preserving existing conditions in text is a little more complicated because existing conditions may be applied anywhere in the text. You must use the **Get TextList** command to identify and preserve existing conditions in the text range or object that you are applying the new condition to. To test the next script, follow these steps.

1. Make a new FrameMaker document.
2. Create two conditions in the document: Condition1 and Condition2. Make sure you assign a different color to each condition.
3. Enter a paragraph of text with a few sentences.
4. Apply Condition1 to a word or two in the middle of the paragraph.

The first example will apply the Condition2 format to the entire paragraph, while preserving the Condition1 format that is already applied. Make sure you click in the paragraph before running the script.

Code Listing 5-68

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get a list of character property changes in the paragraph.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);

// Set a variable for the offset where conditional text changes.
Set vCondOffset = 0;
```

```
// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // See if the property change is a condition format.
  Find Member('CONDITIONTAG') InList(vCharPropsChange.TextData)
    ReturnStatus(vFound);
  If vFound
    // Make a text range for the existing condition.
    New TextRange NewVar(vTextRange) Object(vPgf)
      Offset(vCondOffset) Offset(vCharPropsChange.TextOffset);
    // Write the text to the Console window.
    Write Console vTextRange.Text;
    // Set the variable indicating the start offset.
    Set vCondOffset = vCharPropsChange.TextOffset;
  EndIf
EndLoop

// Make a text range including the end of the paragraph.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(vCondOffset)
  Offset(ObjEndOffset);
// Write the text to the Console window.
Write Console vTextRange.Text;
```

Look at the Console window. At this point, the script simply divides the paragraph up into separate text ranges for each condition format combination, including text that has no condition applied. If you run the script on a paragraph that has no conditions applied, the entire paragraph will be written to the Console. We now have a mechanism to apply the new condition to each text range, while preserving any existing conditions. We will use a subroutine to apply the new condition to the current text range. Here is the finished script.

Code Listing 5-69

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get the Condition2 object.
Get Object Type(CondFmt) Name('Condition2') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
If vCondFmt = 0
  // If the condition doesn't exit, warn the user and exit.
  MsgBox 'Condition2 format does not exist in this document.    ';
  LeaveSub;
EndIf

// Convert the condition object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);

// Get a list of character property changes in the paragraph.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);

// Set a variable for the offset where conditional text changes.
Set vCondOffset = 0;
```

```
// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // See if the property change is a condition format.
  Find Member('CONDITIONTAG') InList(vCharPropsChange.TextData)
    ReturnStatus(vFound);
  If vFound
    // Make a text range for the existing condition.
    New TextRange NewVar(vTextRange) Object(vPgf)
      Offset(vCondOffset) Offset(vCharPropsChange.TextOffset);
    // Run a subroutine to apply the new condition.
    Run ApplyCondition;
    // Set the variable indicating the start offset.
    Set vCondOffset = vCharPropsChange.TextOffset;
  EndIf
EndLoop

// Make a text range including the end of the paragraph.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(vCondOffset)
  Offset(ObjEndOffset);
// Run a subroutine to apply the new condition.
Run ApplyCondition;

Sub ApplyCondition
//
// Set a variable for the current InCond setting for the text range.
Set vInCond = vTextRange.Begin.InCond;
// See if the new condition integer is already in the list.
Find Member(vCondInt) InList(vInCond) ReturnStatus(vFound);
If vFound = 0
  // If the condition is not in the list, add it.
  Add Member(vCondInt) To(vInCond);
  // Make a property list containing the updated integer list.
  New PropertyList NewVar(vProps) InCond(vInCond);
  // Apply the updated integer list to the text range.
  Apply TextProperties TextRange(vTextRange) Properties(vProps);
EndIf
//
EndSub
```

Removing Conditions from Text and Table Rows

Removing *all* conditions from text is simple; use the **Apply TextProperties** command with a null string in the **CondFmt** parameter.

Code Listing 5-70

```
// Remove ALL conditions from the selected text.
Apply TextProperties CondFmt('') TextRange(TextSelection);
```

You use a similar technique to remove *all* conditions from a table row. The script below removes all condition formats from all of the rows in the selected table.

Code Listing 5-71

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Make a variable for the selected table.
Set vTbl = SelectedTbl;

// Make an empty integer list.
New IntList NewVar(vInCond);

// Loop through the table rows.
Loop ForEach(Row) In(vTbl) LoopVar(vRow)
  // Apply the empty integer list to the table row.
  Set vRow.InCond = vInCond;
EndLoop
```

Preserving Existing Conditions

You can remove condition formats while preserving existing ones. You do this by manipulating the **InCond** property of the text or table rows. Here is a script that removes a condition from the table rows in the selected table and preserves existing conditions.

Code Listing 5-72

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make a variable for the selected table.
Set vTbl = SelectedTbl;

// Get the Condition2 object.
Get Object Type(CondFmt) Name('Condition2') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
If vCondFmt = 0
  // If the condition doesn't exit, warn the user and exit.
  MsgBox 'Condition2 format does not exist in this document.    ';
  LeaveSub;
EndIf

// Convert the condition object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);

// Loop through the table rows.
Loop ForEach(Row) In(vTbl) LoopVar(vRow)
  // Make a variable for the row's InCond integer list.
  Set vInCond = vRow.InCond;
  // See if Condition2 is in the list.
  Find Member(vCondInt) InList(vInCond) ReturnStatus(vFound);
  If vFound
    // Remove the member from the list.
    Remove Member(vCondInt) From(vInCond);
    // Apply the modified list to the table row.
    Set vRow.InCond = vInCond;
  EndIf
EndLoop
```

When removing a condition from text, we can use the same technique employed in Code Listing 5-69 on page 5-46. In fact, everything in the following script is the same, except the subroutine and the two lines calling the subroutine.

Code Listing 5-73

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Get the Condition2 object.
Get Object Type(CondFmt) Name('Condition2') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
If vCondFmt = 0
  // If the condition doesn't exit, warn the user and exit.
  MsgBox 'Condition2 format does not exist in this document.    ';
  LeaveSub;
EndIf

// Convert the condition object to an integer.
New Integer NewVar(vCondInt) IntValue(vCondFmt);

// Get a list of character property changes in the paragraph.
Get TextList InObject(vPgf) CharPropsChange NewVar(vTextList);

// Set a variable for the offset where conditional text changes.
Set vCondOffset = 0;

// Loop through the text list.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vCharPropsChange);
  // See if the property change is a condition format.
  Find Member('CONDITIONTAG') InList(vCharPropsChange.TextData)
    ReturnStatus(vFound);
  If vFound
    // Make a text range for the existing condition.
    New TextRange NewVar(vTextRange) Object(vPgf)
      Offset(vCondOffset) Offset(vCharPropsChange.TextOffset);
    // Run a subroutine to apply the new condition.
    Run RemoveCondition;
    // Set the variable indicating the start offset.
    Set vCondOffset = vCharPropsChange.TextOffset;
  EndIf
EndLoop

// Make a text range including the end of the paragraph.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(vCondOffset)
  Offset(ObjEndOffset);
// Run a subroutine to apply the new condition.
Run RemoveCondition;
```

```
Sub RemoveCondition
//
// Set a variable for the current InCond setting for the text range.
Set vInCond = vTextRange.Begin.InCond;
// See if the condition integer is in the list.
Find Member(vCondInt) InList(vInCond) ReturnStatus(vFound);
If vFound
  // If the condition is not in the list, add it.
  Remove Member(vCondInt) From(vInCond);
  // Make a property list containing the updated integer list.
  New PropertyList NewVar(vProps) InCond(vInCond);
  // Apply the updated integer list to the text range.
  Apply TextProperties TextRange(vTextRange) Properties(vProps);
EndIf
//
EndSub
```

Creating Condition Formats

FrameScript allows you to create condition formats by using the **New ConditionFormat** command. You supply the name of the condition format in the **Name** parameter. You should check to see if the format already exists before creating it. The following script creates a condition called “SecondEdition” in the active document.

Code Listing 5-74

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// See if the condition format already exists.
Get Object Type(CondFmt) Name('SecondEdition')
  DocObject(vCurrentDoc) NewVar(vCondFmt);
If vCondFmt
  LeaveSub; // Exit the script.
EndIf

// Make the condition format.
New ConditionFormat Name('SecondEdition') DocObject(vCurrentDoc);
```

Condition formats usually have a character style and a color to identify them in the document. You can set these properties after you create the condition format, or you can change them in an existing format. The following script creates a condition, and then sets its style to Strikethrough and its color to Green.

Code Listing 5-75

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// See if the condition format already exists.
Get Object Type(CondFmt) Name('SecondEdition')
  DocObject(vCurrentDoc) NewVar(vCondFmt);
If vCondFmt = 0
  // If it doesn't exist, create the format.
  New ConditionFormat Name('SecondEdition') DocObject(vCurrentDoc)
    NewVar(vCondFmt);
EndIf
```

```
// Set the condition's style to Strikethrough.
Set vCondFmt.StyleOverride = CnStrikeThrough;

// Set the condition's color to Green.
Get Object Type(Color) Name('Green') DocObject(vCurrentDoc)
  NewVar(vColor);
Set vCondFmt.SepOverride = vColor;
```

Note that we didn't have to test for the existence of the Green color, because Green is a reserved color that exists in every FrameMaker document. If you are using a non-reserved color, make sure you test for its existence in the document before attempting to use it.

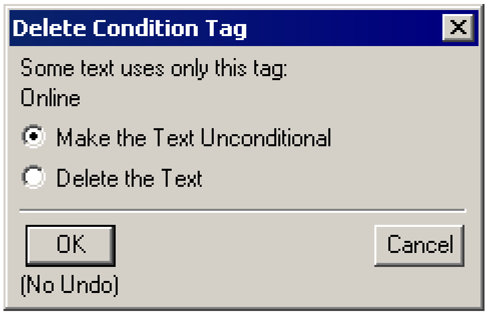
Deleting Condition Formats

FrameScript allows you to delete condition formats by getting the **CondFmt** object and deleting it.

Code Listing 5-76

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the condition format object.
Get Object Type(CondFmt) Name('PaperUK') DocObject(vCurrentDoc)
  NewVar(vCondFmt);
If vCondFmt
  // Delete the condition format.
  Delete Object(vCondFmt);
EndIf
```

When you delete a condition format in the FrameMaker interface, you will get the Delete Condition Tag dialog box if some of the text or table rows in the document is formatted with only that condition. This gives you a choice to keep the text and make it unconditional, or to delete the conditional text and table rows along with the condition.



Some text uses only this tag:
Online

● Make the Text Unconditional

○ Delete the Text

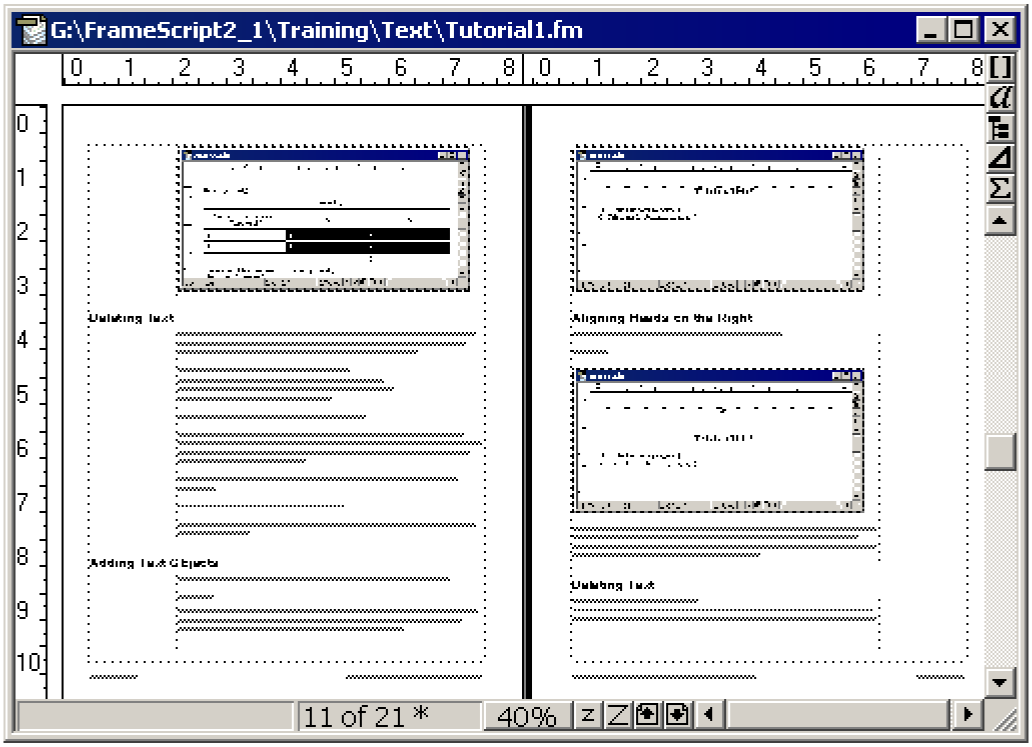
(No Undo)

When you delete a condition format with FrameScript, *any text or table rows with that condition format applied is deleted*. The only exception is text or table rows that have additional conditions applied to it. If you want to delete the condition but keep the text and table rows, you must remove the condition format from the text and table rows before you delete the condition. See "Removing Conditions from Text and Table Rows" on page 5-47.

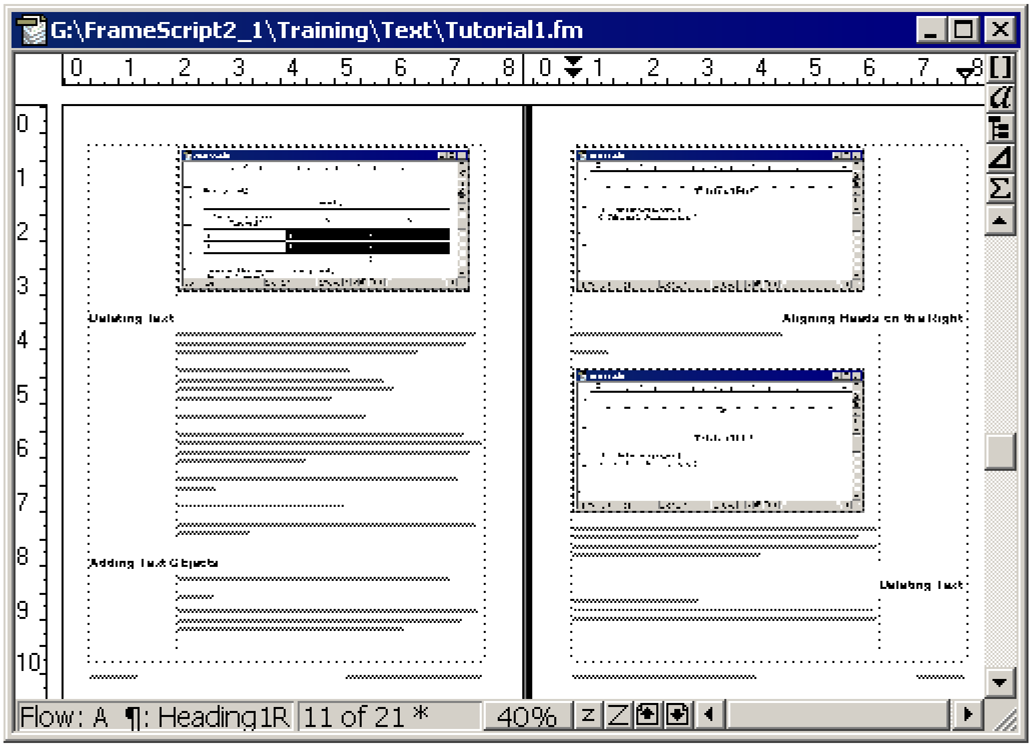
Tutorial 5-1: Applying Paragraph Formatting

This tutorial will illustrate how to apply paragraph formats to paragraphs in a document. We will write a complete script that includes error checking and comments.

We will imagine that a client has given us requirements for a custom script. They have double-sided documents with side-head areas that are opposite the binding. On left-hand pages the side-head area is on the left side of the text frame; on right-hand pages, the side-head area is on the right. Heading1 paragraphs are set to straddle across the side-head area and the main text column.


G:\FrameScript2_1\Training\Text\Tutorial1.fm
Deleting Text
Adding Text Objects
Aligning Heads on the Right
Deleting Text
11 of 21 *
40%

Heading1 paragraphs are normally left-aligned (see screenshot above), but on right-hand pages, the client wants the Heading1 paragraphs right-aligned so that they are visible in the side-head area (see screenshot below).


G:\FrameScript2_1\Training\Text\Tutorial1.fm
Deleting Text
Adding Text Objects
Aligning Heads on the Right
Deleting Text
Flow: A ¶: Heading1R 11 of 21 *
40%

For this tutorial, make a new document that has a double-sided page layout. Put several Heading1 paragraphs on left-hand pages, and several on right-hand pages.

Testing for an Active Document

Since this script runs on a document, we need to make sure there is an active document when the script is run. An active document (**ActiveDoc**) is an open document in FrameMaker that has the focus and is the front-most window.

Code Listing 5-77

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set vCurrentDoc = ActiveDoc;
EndIf
```

If there is no active document, the script will warn the user and exit. If there is an active document, we will set the **vCurrentDoc** variable to the **ActiveDoc**. We do this so that the script can later be modifed to run on documents that are not active; for example, on all of the files in a book.

Now we want to start at the top of the main text flow in the document and test each paragraph. When making blocks of code, such as loops, it is sometimes helpful to write a shell for the block and then fill in the details. That way, we can test the shell before adding more code. Here is the shell for the loop through our paragraphs.

Code Listing 5-78

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Write the paragraph name to the Console.
  Write Console vPgf.Name;
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Run the code and you will see that the name of each paragraph in the main text flow is written to the Console window. The script may take a few seconds, or longer with a large document, but it should run successfully without error. If you get errors, go to each line that gives an error and fix the code. Be careful with loops; if you leave out the second last line above, or if it has a syntax error, your script may stay in an endless loop. If you think this is the case, try hitting the Escape key to abort the script.

Once the loop works correctly, we will add more code inside the loop. We need to test each paragraph to see if it is a Heading1 paragraph. We will use an **If/EndIf** statement to do the test.

Code Listing 5-79

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Heading1' // Names are case-sensitive
    // Write the paragraph name to the Console.
    Write Console vPgf.Name;
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

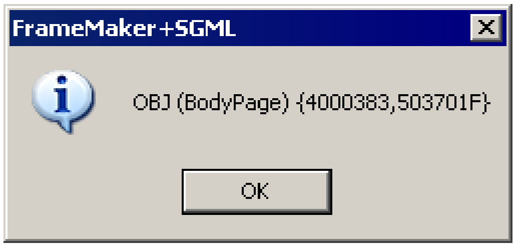
The script will now only write Heading1 paragraph names to the Console. Change **vPgf.Name** to **vPgf.Text** and rerun the script. It will write the text of each Heading1 paragraph to the Console.

Now that our script locates the Heading1 paragraphs, we have to determine the page that each one is on. Many objects in FrameMaker have a special property called **Page** that returns the **Page** object where the object is located. You can use the **Page** property on a **Pgf** object to see what page it is on.

When you are exploring a property that you might not be familiar with, it is often helpful to leave the main script and experiment with the property in a temporary script. This helps you to figure out the syntax needed in the main script. Let’s do that with the Page property. Click in a paragraph in your document, and run this code.

Code Listing 5-80

```
Set vPgf = TextSelection.Begin.Object;
Display vPgf.Page;
```

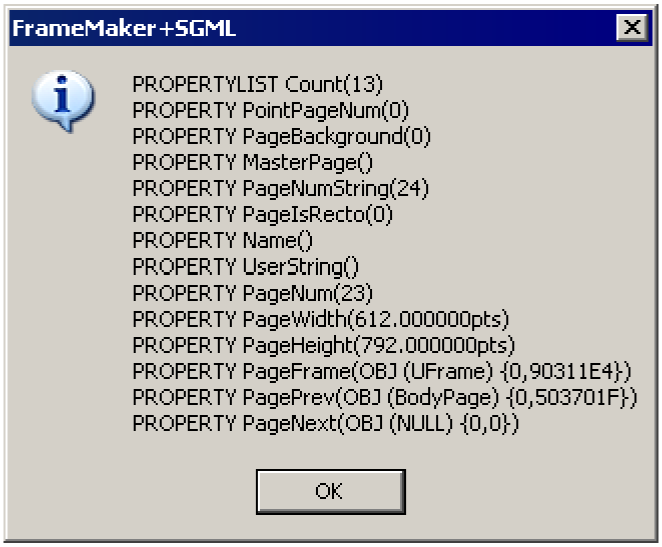


```
OBJ (BodyPage) {4000383,503701F}
```

The BodyPage object by itself is not very useful, but we can look it up in the *FrameScript Scriptwriter's Reference* to see if there is a property that will determine if it is a left or right page. Or, we can "cheat" and use code to display the properties.

Code Listing 5-81

```
Set vPgf = TextSelection.Begin.Object;
Display vPgf.Page.Properties;
```



```
PROPERTYLIST Count(13)
PROPERTY PointPageNum(0)
PROPERTY PageBackground(0)
PROPERTY MasterPage()
PROPERTY PageNumString(24)
PROPERTY PageIsRecto(0)
PROPERTY Name()
PROPERTY UserString()
PROPERTY PageNum(23)
PROPERTY PageWidth(612.000000pts)
PROPERTY PageHeight(792.000000pts)
PROPERTY PageFrame(OBJ (UFrame) {0,90311E4})
PROPERTY PagePrev(OBJ (BodyPage) {0,503701F})
PROPERTY PageNext(OBJ (NULL) {0,0})
```

The property we are looking for is **PageIsRecto**. *Recto* is an old printing term that refers to a right-hand page (*verso* is a left-hand page). If the **PageIsRecto** property is 1, the paragraph is on a right-hand page, if **PageIsRecto** = 0, it is on a left-hand page. Let's modify our main code to test this property.

Code Listing 5-82

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Heading1' // Names are case-sensitive
    // See what kind of page the paragraph is on.
    If vPgf.Page.PageIsRecto
      Write Console 'Right-hand page.';
    Else
      Write Console 'Left-hand page.';
    EndIf
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

With the main logic of the script in place, we now have to determine the best way to right-align the Heading1 paragraphs on the right-hand pages. Again, let’s leave the main script to explore the possibilities.

We could make a PropertyList with the alignment property and apply that to the right page paragraphs.

```
New PropertyList NewVar(vRightProps) PgfAlignment(PgfRight);
```

There is no sense in making the property list over and over again inside the loop, so we can put it before the loop. Then we can apply the properties where appropriate inside the loop.

Code Listing 5-83

```
New PropertyList NewVar(vRightProps) PgfAlignment(PgfRight);

// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Heading1' // Names are case-sensitive
    // See what kind of page the paragraph is on.
    If vPgf.Page.PageIsRecto
      Set vPgf.Properties = vRightProps;
    EndIf
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

This script works, but there is one glaring problem. By setting the paragraph properties this way, we are creating an override on the Heading1 paragraphs on right-hand pages. We want to avoid overrides, so instead we can apply a special paragraph format called Heading1Right.

Code Listing 5-84

```
// Get the Heading1Right paragraph format object.
Get Object Type(PgfFmt) Name('Heading1Right')
  DocObject(vCurrentDoc) NewVar(vRightPgfFmt);
If vRightPgfFmt = 0
  // If the paragraph format does not exist, exit the script.
  MsgBox 'Heading1Right does not exist in this document.    ';
  LeaveSub;
EndIf

// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Heading1' // Names are case-sensitive
    // See what kind of page the paragraph is on.
    If vPgf.Page.PageIsRecto
      Set vPgf.Properties = vRightPgfFmt.Properties;
    EndIf
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

We are checking to make sure that the Heading1Right paragraph format exists before processing the paragraphs. The script works, but can you see a potential problem? What if the document reflows after the script is run? This will be likely because of edits that the client makes to the document. Because Heading1Right paragraphs could end up on a left-hand page, we should test for these also and reset them to Heading1 paragraphs.

We also need to account for the possibility that your client has used manual page breaks on Heading1 paragraphs. If we apply a Heading1Right paragraph format to a paragraph with a page break override, the override will be lost and the document may reflow. Here is the modified code.

Code Listing 5-85

```
// Get the Heading1 paragraph format object.
Get Object Type(PgfFmt) Name('Heading1')
  DocObject(vCurrentDoc) NewVar(vLeftPgfFmt);
If vLeftPgfFmt = 0
  // If the paragraph format does not exist, exit the script.
  MsgBox 'Heading1 does not exist in this document.    ';
  LeaveSub;
EndIf

// Get the Heading1Right paragraph format object.
Get Object Type(PgfFmt) Name('Heading1Right')
  DocObject(vCurrentDoc) NewVar(vRightPgfFmt);
If vRightPgfFmt = 0
  // If the paragraph format does not exist, exit the script.
  MsgBox 'Heading1Right does not exist in this document.    ';
  LeaveSub;
EndIf
```

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If (vPgf.Name = 'Heading1') or (vPgf.Name = 'Heading1Right')
    // Store the page break setting in a variable.
    Set vStart = vPgf.Start;
    // See what kind of page the paragraph is on.
    If vPgf.Page.PageIsRecto
      Set vPgf.Properties = vRightPgfFmt.Properties;
    Else
      Set vPgf.Properties = vLeftPgfFmt.Properties;
    EndIf
    // Restore the original page break setting.
    Set vPgf.Start = vStart;
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Before the loop, we are getting both **PgfFmt** objects. Inside the loop, we have expanded the test for the **vPgf.Name** to include Heading1Right paragraphs. We have also added the **Else** statement back into the page side test to handle left-hand pages.

There is one enhancement that we can add to the script that will surely please your client. Since the script requires the Heading1Right paragraph format, we can put code in the script to add it if it doesn’t exist. Heading1Right right will have exactly the same properties as Heading1 except for the alignment. This means that the properties on the new format will be correct no matter what template the client is using. The only requirement will be that the Heading1 format exists. Let’s use a temporary script to develop the code.

Code Listing 5-86

```
// Get the Heading1 paragraph format object.
Get Object Type(PgfFmt) Name('Heading1')
  DocObject(vCurrentDoc) NewVar(vLeftPgfFmt);
If vLeftPgfFmt = 0
  // If the paragraph format does not exist, exit the script.
  MsgBox 'Heading1 does not exist in this document.    ';
  LeaveSub;
EndIf
```

We can start with this existing code, because Heading1 needs to be in the document in order to proceed. We want to create a new format based on this format. First of all, create a new PropertyList and apply the Heading1 paragraph properties to it.

Code Listing 5-87

```
New PropertyList NewVar(vRightProps);
Set vRightProps = vLeftPgfFmt.Properties;
```

Now, we need to remove the two properties that will be different between the two formats—the **Name** property and the **PgfAlignment** property. After we remove the **PgfAlignment** property, we can add it back with the right aligned value.

Code Listing 5-88

```
Remove Property(Name) From(vRightProps);
Remove Property(PgfAlignment) From(vRightProps);
Add Property To(vRightProps) PgfAlignment(PgfRight);
```

The **Name** property will be supplied when we create the new paragraph format.

Code Listing 5-89

```
New PgfFmt Name('Heading1Right') DocObject(vCurrentDoc)
  NewVar(vRightPgfFmt);
Set vRightPgfFmt.Properties = vRightProps;
```

The last line applies the correct properties to the new paragraph format. To test the code, let's first put it in a subroutine called MakeFormat.

Code Listing 5-90

```
Sub MakeFormat
//
New PropertyList NewVar(vRightProps);
Set vRightProps = vLeftPgfFmt.Properties;
Remove Property(Name) From(vRightProps);
Remove Property(PgfAlignment) From(vRightProps);
Add Property To(vRightProps) PgfAlignment(PgfRight);
New PgfFmt Name('Heading1Right') DocObject(vCurrentDoc)
  NewVar(vRightPgfFmt);
Set vRightPgfFmt.Properties = vRightProps;
//
EndSub
```

Open a new, portrait FrameMaker document and run the following code.

Code Listing 5-91

```
Set vCurrentDoc = ActiveDoc;

// Get the Heading1 paragraph format object.
Get Object Type(PgfFmt) Name('Heading1')
  DocObject(vCurrentDoc) NewVar(vLeftPgfFmt);
If vLeftPgfFmt = 0
  // If the paragraph format does not exist, exit the script.
  MsgBox 'Heading1 does not exist in this document.    ';
  LeaveSub;
EndIf

Run MakeFormat;
```

```
Sub MakeFormat
//
New PropertyList NewVar(vRightProps);
Set vRightProps = vLeftPgfFmt.Properties;
Remove Property(Name) From(vRightProps);
Remove Property(PgfAlignment) From(vRightProps);
Add Property To(vRightProps) PgfAlignment(PgfRight);
New PgfFmt Name('Heading1Right') DocObject(vCurrentDoc)
  NewVar(vRightPgfFmt);
Set vRightPgfFmt.Properties = vRightProps;
//
EndSub
```

Since the Heading1 format exists in FrameMaker's portrait template, the script should make the Heading1Right paragraph format. Open the Paragraph Designer, choose the Heading1Right paragraph tag, and look at the Alignment properties on the Basic tab.

Once you have tested this code, copy the **MakeFormat** subroutine to the END of the main script; subroutines must go at the end of a script after any code that is not in a subroutine. Put the code to call the subroutine in the following block.

Code Listing 5-92

```
// Get the Heading1Right paragraph format object.
Get Object Type(PgfFmt) Name('Heading1Right')
  DocObject(vCurrentDoc) NewVar(vRightPgfFmt);
If vRightPgfFmt = 0
  // If the Heading1Right paragraph format does not exist, call the
  // MakeFormat subroutine to create it.
  Run MakeFormat;
EndIf
```

Test the finished script to see how it works. As an additional exercise, add your own comments to the commands in the MakeFormat subroutine.

Tutorial 5-2: Applying Character Formatting

This tutorial will show how to apply character formatting to text based on certain conditions. Our imaginary client has a large glossary with terms and definitions. Each glossary entry is in a single paragraph called Glossary and begins with the term, followed by a colon and a space (:). The client wants the term and the colon formatted with a character format called Bold.

Isolating the Script's Tasks

Before we begin, let's make an outline of what the script needs to do.

- Test for an active document.
- See if the Bold character format exists; if not, create it.
- Loop through all of the paragraphs in the main document flow.
- Test each paragraph, looking for paragraphs with the Glossary format applied.
- Isolate the term in the paragraph, and apply the Bold character format to it.

If you worked through the Paragraph Formatting tutorial, it should be evident that most of the tasks in this script will be the same as the other one. As you gain experience, you will find much of your scripting will consist of code you have already written. To take advantage of this, let's duplicate the previous script, and modify it to suit our new purposes.

Code Listing 5-93

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set vCurrentDoc = ActiveDoc;
EndIf
```

Since this script will also work on an active document, this code is the same.

Code Listing 5-94

```
// Get the Bold character format object.
Get Object Type(CharFmt) Name('Bold')
  DocObject(vCurrentDoc) NewVar(vCharFmt);
If vCharFmt = 0
  // If the Bold character format does not exist, call
  // the MakeFormat subroutine to create it.
  Run MakeFormat;
EndIf

Sub MakeFormat
//
// Get the FontWeight integer for the Bold font weight.
Find Member('Bold') InList(FontWeightNames) ReturnPos(vPos);
// Make a property list and add the Bold weight to it.
New PropertyList NewVar(vProps);
Add Property To(vProps) FontWeight(vPos-1);
// Make the Bold character format.
New CharFmt Name('Bold') DocObject(vCurrentDoc) NewVar(vCharFmt);
// Apply the Bold font weight to the Bold character format.
Set vCharFmt.Properties = vProps;
//
EndSub
```

Instead of testing for the existence of a paragraph format, we modify the test to see if the Bold character format exists in the document. If it doesn't exist, we call the modified **MakeFormat** subroutine to create the format. Compare this **MakeFormat** subroutine with previous one to see how it differs. The code we have so far is self-contained so you can test it with a new, portrait FrameMaker document.

Now we come to the part of the script that does the bulk of the work—the loop that moves through the paragraphs. Here is the shell with the unneeded parts stripped out.

Code Listing 5-95

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Glossary'
    // Process this paragraph.
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

We are back to a simple test for the paragraph format name—in this case, Glossary. Our character formatting code will go where it says // **Process this paragraph**. In the previous script, we applied the paragraph formatting inside the loop. Because, applying the character formatting in this script will be a little more complex, we do it in a subroutine. Let's move away from the main script and work on the subroutine.

Finding the Glossary Term

Once we find a Glossary paragraph, we need a way to isolate the term in the paragraph. The term should end with a colon and a space (:), and this colon and space should be the first occurrence of these characters in the paragraph. Click in a Glossary paragraph and run the following code.

Code Listing 5-96

```
Set vPgf = TextSelection.Begin.Object;

Find String(': ') InObject(vPgf) ReturnStatus(vFound)
  ReturnRange(vRange);
If vFound
  Display vRange.Text;
EndIf
```

If your paragraph has a colon+space in it, it will be displayed in a dialog box; if not, nothing will happen. We can reverse this, and exit the subroutine if it is NOT found.

Code Listing 5-97

```
Find String(': ') InObject(vPgf) ReturnStatus(vFound)
  ReturnRange(vRange);
If vFound = 0
  LeaveSub;
EndIf
```

Applying the Character Format

To apply a character format to text, we need a valid **TextRange** object. We can use the information provided in the **vRange** variable to help us make the correct **TextRange** for the Glossary term and the colon.

Code Listing 5-98

```
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0)
  Offset(vRange.Begin.Offset+1);
```

Below is the completed code so far. With your cursor in a Glossary paragraph, give it a try.

Code Listing 5-99

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
// Find the colon and space in the paragraph.
Find String(': ') InObject(vPgf) ReturnStatus(vFound)
  ReturnRange(vRange);
// If the string is not found, exit the script.
If vFound = 0
  LeaveSub;
EndIf
// Make a text range from the beginning of the paragraph through
// the colon and space.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0)
  Offset(vRange.Begin.Offset+1);
// Select the text range.
Set TextSelection = vTextRange;
```

The last line is not necessary for the finished script; it is simply there so you can see that the Glossary term is highlighted. Delete the last line and add the code to apply the character format to the **TextRange**.

Code Listing 5-100

```
// Apply the character format properties to the text range.
Apply TextProperties TextRange(vTextRange)
  Properties(vCharFmt.Properties);
```

Delete the first line, wrap the code in the **Sub**/**EndSub** statements, and put the whole subroutine at the end of the main script.

Code Listing 5-101

```
Sub FormatTerm
//
// Find the colon and space combination.
Find String(': ') InObject(vPgf) ReturnStatus(vFound)
  ReturnRange(vRange);
// If it’s not found, exit the subroutine.
If vFound = 0
  LeaveSub;
EndIf
// Make a text range containing the Glossary term.
New TextRange NewVar(vTextRange) Object(vPgf) Offset(0)
  Offset(vRange.Begin.Offset+1);
// Apply the character format properties to the text range.
Apply TextProperties TextRange(vTextRange)
  Properties(vCharFmt.Properties);
//
EndSub
```

Now we can add a call to the subroutine inside the main loop.

Code Listing 5-102

```
// Find the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through all of the paragraphs in the main text flow.
Loop While(vPgf)
  // Test the paragraph format name.
  If vPgf.Name = 'Glossary'
    // Process this paragraph by calling a subroutine.
    Run FormatTerm;
  EndIf
  // Move to the next paragraph in the flow (don’t forget this!).
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Give the completed script a try on your sample document.

Bonus Task: Making the Script “Look Better”

When a script makes changes to an active document, the screen will usually “flicker” as the changes are made. This is because FrameMaker is refreshing the display as each change is made. We can turn off the property that refreshes the screen while the script is running. Not only will the display look “cleaner” but the script will run faster because it doesn’t have to refresh the screen after each change.

You need to add code in two places. Add these lines before the block containing the main loop.

Code Listing 5-103

```
// Turn off the document display to prevent flicker.
Set Displaying = 0;
```

And, after the main loop, and but before the subroutines, add these lines.

Code Listing 5-104

```
// Turn on the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;
```

When using the Displaying property, there are a couple things to keep in mind.

- If the script exits after you set **Displaying = 0** and before you turn it back on, your screen will not refresh and you will see a question mark (**?**) in the Page Status area. That is why you should turn it off after any tests that might cause a **LeaveSub** to be executed. If that is not possible, use **Set Displaying = 1;** before your **LeaveSub** statements.
- If your script crashes before it reaches the **Set Displaying = 1;** command, the screen will become white and a question mark (**?**) will appear in the Page Status area. If that happens, choose **FrameScript > Script Window**, type **Set Displaying = 1;** in the Script Window, click the **Run** button. This will restore the display.

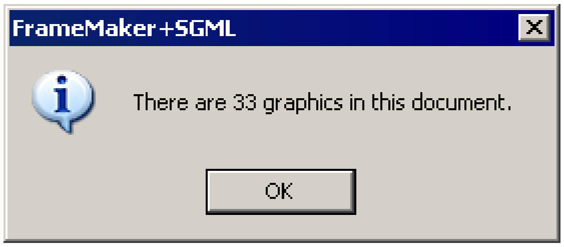
6 Working With Graphics

Graphics are pervasive in a FrameMaker document. Even the text in a document is contained by graphics—text frames and text lines. The exception is text in tables, but tables are anchored in paragraphs, which are contained by text frames. Graphics can contain other graphics; for example, an anchored frame can contain other graphics, such as imported graphics and lines. Most of your scripts will deal with graphics in one way or another.

Make a new Portrait document and run the following script. It counts the graphics in the document. You will see that even a "blank" document contains many graphics.

Code Listing 6-1

```
Set vCurrentDoc = ActiveDoc;

// Initialize a counter.
Set n = 0;

// Loop through the graphics in the document and count them.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
Loop While(vGraphic)
  Set n = n + 1;
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop

// Display a message.
MsgBox 'There are '+ n +' graphics in this document.    ';
```



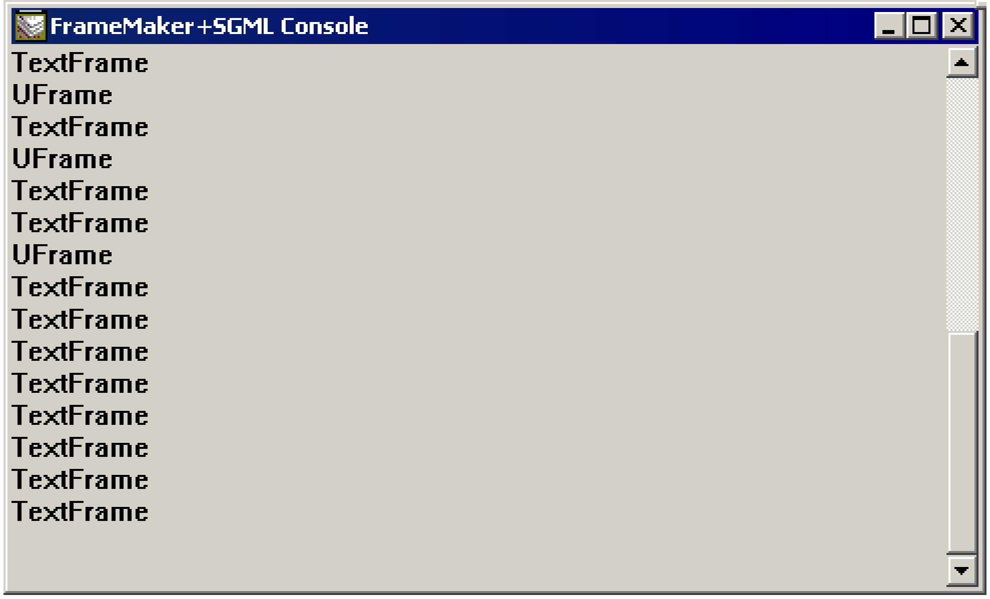
There are 33 graphics in this document.

FrameMaker has different kinds of graphics, such as text frames, lines, and arcs. You identify the type of graphic by using its **ObjectName** property. Here is a script that loops through all of the graphics in the document and writes each graphic's **ObjectName** to the Console window. Run the script and look at the Console.

Code Listing 6-2

```
Set vCurrentDoc = ActiveDoc;
```

```
// Loop through the graphics in the document.
Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
Loop While(vGraphic)
  // Write the graphic’s object name to the Console.
  Write Console vGraphic.ObjectName;
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```



```
TextFrame
UFrame
TextFrame
UFrame
TextFrame
TextFrame
UFrame
TextFrame
TextFrame
TextFrame
TextFrame
TextFrame
TextFrame
TextFrame
TextFrame
```

Many of the graphic types can be drawn with tools in the FrameMaker tool box. The following table lists each type of graphic, its **ObjectName**, how the graphic is created in the interface, and a description. The table is sorted by the FrameScript **ObjectName**.

Graphic Name	ObjectName	Tool or Menu	Description
Anchored frame	**AFrame**	**Special > Anchored Frame** or **File > Import > File**	A container for graphic objects that is tied to a specific location in text.
Arc	**Arc**	⌒	Open quarter-oval
Oval	**Ellipse**	⬭	Oval or circle
Group	**Group**	**Graphics > Group**	An invisible graphic object that holds together a set of other graphics.
Imported graphic	**Inset**	**File > Import > File**	An external graphic imported by reference or copy.
Line	**Line**	╲	Line
Math equation	**Math**	Equation palette	A math equation, which describes a formatted equation.

Graphic Name	ObjectName	Tool or Menu	Description
Polyline	**Polyline**		An open-ended polygon
Freehand curve	**Polyline**		An open-ended polygon that is “smoothed.”
Rectangle	**Rectangle**	▭	Rectangle or square
Rounded rectangle	**RoundedRect**	⬭	A rectangle with rounded corners.
Text frame	**TextFrame**	▤	A container for text in a flow.
Text line	**TextLine**	A	A single line of text that is not in a paragraph or flow.
Unanchored frame	**UFrame**		A container for graphic objects that is not tied to a specific location in text. Also used for named reference page frames.

IMPORTANT: ObjectNames are case-sensitive when used in your scripts. The **ObjectName** property is a string value, so you must use single quotes around the **ObjectName**s in your scripts.

Working With Selected Graphics

A document has a **FirstSelectedGraphicInDoc** property that indicates if one or more graphics are currently selected in the document. If no graphics are selected, the **FirstSelectedGraphicInDoc** property will return 0 (zero). You can use this property to experiment with selected graphics and their properties. If there is more than one selected graphic, you can use a loop to move through them.

Code Listing 6-3

```
Set vCurrentDoc = ActiveDoc;

// Get the first selected graphic in the document.
Set vGraphic = vCurrentDoc.FirstSelectedGraphicInDoc;

// Loop through the selected graphics.
Loop While(vGraphic)
  // Write the object name and Y location to the Console.
  Write Console vGraphic.ObjectName;
  Write Console vGraphic.LocY;
  // Get the next selected graphic.
  Set vGraphic = vGraphic.NextSelectedGraphicInDoc;
EndLoop
```

Graphic Properties

All graphic objects have a set of properties that are common to all graphic types. Some properties do not affect some graphic types; for example, a **Rectange** object has an **ArrowType** property. This property can have a value, but it will not the affect the appearance of the rectangle. In addition to common properties, there are properties that are specific to certain types of graphic objects. For instance, **AFrame** objects have an **AnchorType** property that corresponds to the Anchoring Position property in the Anchored Frame dialog box.

You can query and set a graphic's properties by using dot notation and the property name. The following script sets a border around all of the anchored frames in the document except those that are positioned at the insertion point.

Code Listing 6-4

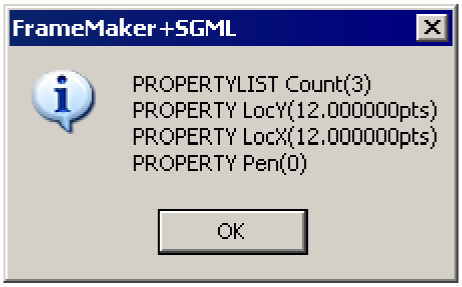
```
Set vCurrentDoc = ActiveDoc;

Set vGraphic = vCurrentDoc.FirstGraphicInDoc;
Loop While(vGraphic)
  // Test for an anchored frame. AFrame is case-sensitive and
  // requires single quotes.
  If vGraphic.ObjectName = 'AFrame'
    // Make sure the anchored frame is not at the insertion point.
    If vGraphic.AnchorType not= AnchorInline
      // Set the border to black.
      Set vGraphic.Pen = FillBlack;
    EndIf
  EndIf
  Set vGraphic = vGraphic.NextGraphicInDoc;
EndLoop
```

Note that when you test for the **ObjectName**, you must put it in single-quotes.

You can also use property lists (**PropertyList**) to apply properties to graphics. This can be useful for changing multiple properties at one time, particularly when you are adding new graphic objects. You can make a **PropertyList** and add properties to it with a single command.

Code Listing 6-5

```
New PropertyList NewVar(vProps) LocY(12pt) LocX(12pt)
  Pen(FillBlack);
Display vProps;
```

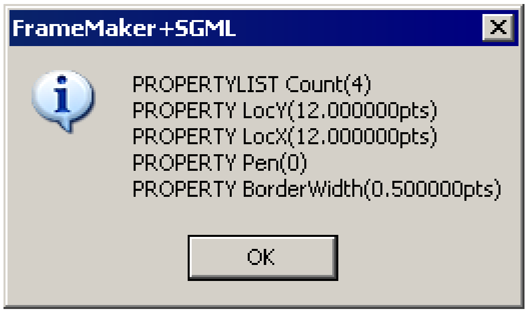


```
PROPERTYLIST Count(3)
PROPERTY LocY(12.000000pts)
PROPERTY LocX(12.000000pts)
PROPERTY Pen(0)
```

You can also make a **PropertyList** and add properties to it with the **Add Property** command.

Code Listing 6-6

```
New PropertyList NewVar(vProps);
Add Property To(vProps) LocY(12pt);
Add Property To(vProps) LocX(12pt);
Add Property To(vProps) Pen(FillBlack) BorderWidth(.5pt);
Display vProps;
```



```
PROPERTYLIST Count(4)
PROPERTY LocY(12.000000pts)
PROPERTY LocX(12.000000pts)
PROPERTY Pen(0)
PROPERTY BorderWidth(0.500000pts)
```

Here is how to apply a **PropertyList** to a graphic; in this case, the selected graphic.

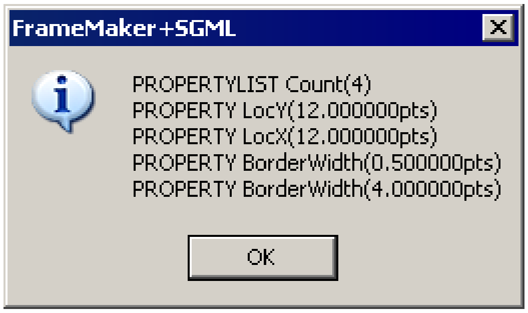
Code Listing 6-7

```
Set vCurrentDoc = ActiveDoc;

// Make a property list and add properties.
New PropertyList NewVar(vProps);
Add Property To(vProps) LocY(12pt);
Add Property To(vProps) LocX(12pt);
Add Property To(vProps) Pen(FillBlack) BorderWidth(8pt);

// See if there is a selected graphic.
If vCurrentDoc.FirstSelectedGraphicInDoc
  // Apply the property list to the selected graphic.
  Set vCurrentDoc.FirstSelectedGraphicInDoc.Properties = vProps;
EndIf
```

It is possible to add a property to a **PropertyList** more than once; for example, in the code below, the **BorderWidth** property is added twice.

Code Listing 6-8

```
New PropertyList NewVar(vProps);
Add Property To(vProps) LocY(12pt);
Add Property To(vProps) LocX(12pt);
Add Property To(vProps) BorderWidth(.5pt);
Add Property To(vProps) BorderWidth(4pt);
Display vProps;
```



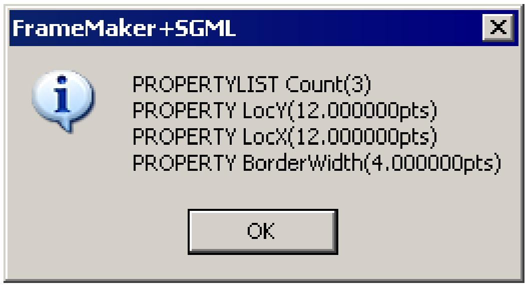
```
PROPERTYLIST Count(4)
PROPERTY LocY(12.000000pts)
PROPERTY LocX(12.000000pts)
PROPERTY BorderWidth(0.500000pts)
PROPERTY BorderWidth(4.000000pts)
```

When there are multiple instances of the same property, the first instance of the property will be used when it is applied to an object.

You can remove a property from a **PropertyList** by using the **Remove Property** command.

Code Listing 6-9

```
New PropertyList NewVar(vProps);
Add Property To(vProps) LocY(12pt);
Add Property To(vProps) LocX(12pt);
Add Property To(vProps) BorderWidth(.5pt);
Add Property To(vProps) BorderWidth(4pt);
Remove Property(BorderWidth) From(vProps);
Display vProps;
```



```
PROPERTYLIST Count(3)
PROPERTY LocY(12.000000pts)
PROPERTY LocX(12.000000pts)
PROPERTY BorderWidth(4.000000pts)
```

Notice that if the property occurs more than once in the **PropertyList**, the first occurrence is removed.

Inserting Graphics

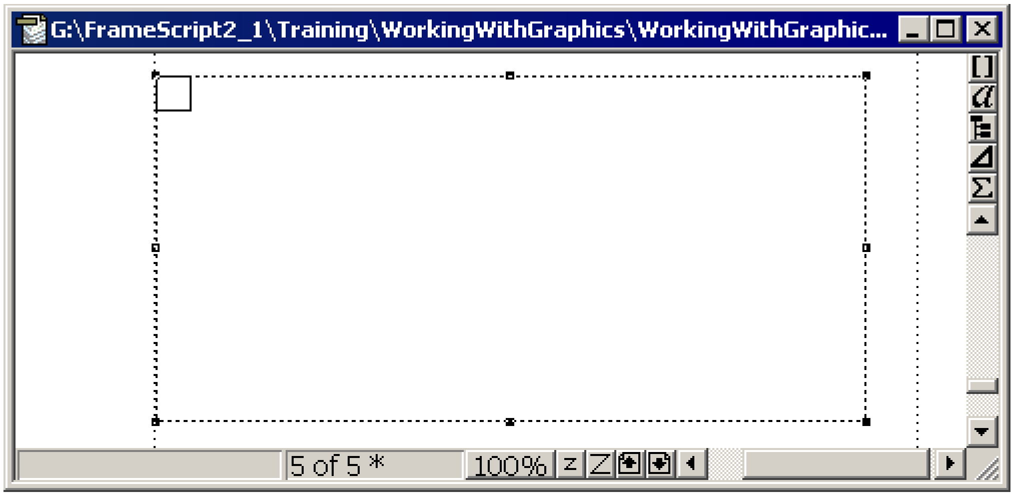
The method you use to add graphics to your documents will depend on the type of graphic. Unanchored graphics, such as geometric shapes, text frames, and unanchored frames are created with the **New** command and the **ParentObject** parameter. Anchored frames are also added with the **New** command, but you use the **TextLoc** parameter to specify a text location in a text flow. Imported graphics are added by using the **Import File** command. Imported graphics can be added at a text location or into a selected anchored frame. They can also be added directly on a page.

Unanchored Graphics

When you add an unanchored graphic to a document, you must use the **New** command with the object name of the graphic (**New Line**, **New Rectangle**, **New TextFrame**, etc.). You use the required **ParentObject** parameter to specify the frame that will contain the new graphic. Here is a script that makes a rectangle inside the selected anchored frame. To try it, put an anchored frame in a document and make sure it is selected when you run the code.

Code Listing 6-10

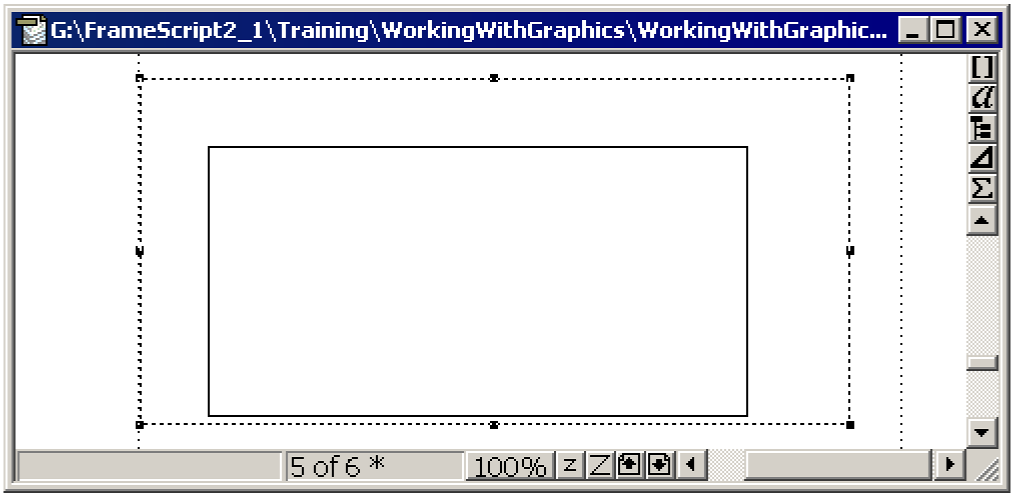
```
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected anchored frame.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Add a rectangle to the anchored frame.
New Rectangle ParentObject(vAFrame) NewVar(vRectangle);
```


G:\FrameScript2_1\Training\WorkingWithGraphics\WorkingWithGraphic...
5 of 5 *
100%

The **Rectangle** is added to the **AFrame** using default values for its properties, including size and location. Since the new **Rectangle** object is returned in a variable by the **NewVar** parameter, you can use this to change its properties after it is inserted.

Code Listing 6-11

```
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected anchored frame.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Add a rectangle to the anchored frame.
New Rectangle ParentObject(vAFrame) NewVar(vRectangle);

// Change the location and size of the rectangle.
Set vRectangle.LocX = .5in;
Set vRectangle.LocY = .5in;
Set vRectangle.Height = 2in;
Set vRectangle.Width = 4in;
```


G:\FrameScript2_1\Training\WorkingWithGraphics\WorkingWithGraphic...
5 of 6 *
100%

You can also assign properties to a graphic when you create it by using property parameters with the **New** command.

Code Listing 6-12

```
Set vCurrentDoc = ActiveDoc;

// Set a variable for the selected anchored frame.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;

// Add a rectangle to the anchored frame.
New Rectangle ParentObject(vAFrame) NewVar(vRectangle)
  LocX(.5in) LocY(.5in) Height(2in) Width(4in);
```

When you want to insert a graphic directly on a page and not in a frame, you still must use the **ParentObject** parameter. Every page in a document has a **PageFrame** object, which is an invisible **UFrame** object that contains all of the objects on the page. Below is a script that adds a text frame directly to the current page in the document.

Code Listing 6-13

```
Set vCurrentDoc = ActiveDoc;

// Set a variable for the current page in the document.
Set vPage = vCurrentDoc.CurrentPage;

// Add the text frame to the current page.
New TextFrame ParentObject(vPage.PageFrame) NewVar(vTextFrame)
  LocX(.5in) LocY(.5in) Height(2in) Width(4in);

// Select the text frame so it's easier to see.
Set vTextFrame.GraphicIsSelected = 1;
```

IMPORTANT: Do not attempt to get a **PageFrame** object's properties. Doing so will crash FrameMaker.

Anchored Frames

There are two ways to add anchored frames to a document. One, is to use the **New AFrame** command and specify a **TextLoc** for the frame. The second is to import a graphic at a text location; an anchored frame will be created automatically to hold the imported graphic. The first method will be discussed in this section.

To insert an **AFrame**, you will usually need a valid **TextLoc** object. If you do not use the **TextLoc** parameter, the **AFrame** will be inserted at the insertion point of the active document. Here is a script that inserts an anchored frame at the end of the paragraph containing the insertion point.

Code Listing 6-14

```
Set vCurrentDoc = ActiveDoc;

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Make a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);

// Insert the anchored frame at the text location.
New AFrame NewVar(vAFrame) TextLoc(vTextLoc) Height(3in)
  AnchorType(AnchorBelow);

// Set the width of the anchored frame to match the column width.
Set vAFrame.Width = vPgf.InTextObj.Width;
```

If you use a **Pgf** object for the **TextLoc** parameter, the **AFrame** will be inserted at the beginning of the paragraph. This eliminates the necessity of creating a **TextLoc** object ahead of time.

Like unanchored graphics, you can set properties on the anchored frame on the **New AFrame** command, as well as after by using the **AFrame** variable value, in this case **vAFrame**. You can also use a **PropertyList** to assign values to the **AFrame** after it is inserted.

Imported Graphics

To import external graphic files into FrameMaker, you use the **Import File** command. The **Import File** command is used to import text and graphics, so it has parameters that deal with both. In this section, we will limit our discussion to the graphics parameters.

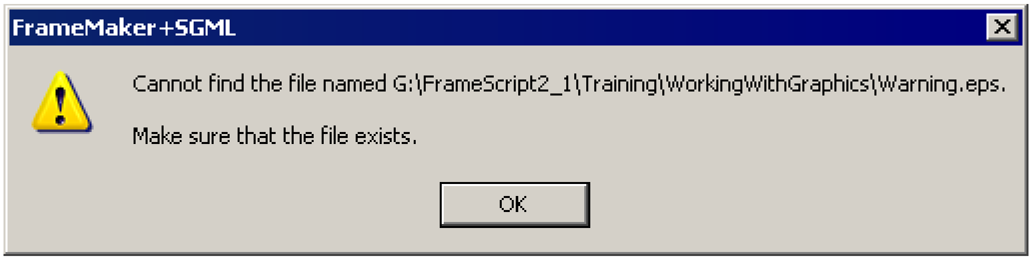
If you use the **Import File** command without any parameters, you will be prompted for a file to import, just as if you choose **File** > **Import** > **File** in the interface. You will usually provide a file name in the **File** parameter. Here is a script that imports a file called Warning.eps that is in the same folder as the active document.

Code Listing 6-15

```
// Test for an active document and make sure it is named.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  If ActiveDoc.Name = ''
    MsgBox 'Save the file and run the script again.    ';
    LeaveSub;
  Else
    Set vCurrentDoc = ActiveDoc;
EndIf

// Import the graphic.
Import File File('Warning.eps');
```

As you can see, we are using a relative path to the graphic file. That is why we require that the document be saved (not untitled) before running the script. You can also provide an absolute path to the graphic, which is often preferred.

If the graphic file does not exist, you will get the standard interface warning.



Cannot find the file named G:\FrameScript2_1\Training\WorkingWithGraphics\Warning.eps.

Make sure that the file exists.

In some scripts, you may want to suppress the interface warning and alert the user in some other way. If you set the **AlertUserAboutFailure** parameter to **False**, the interface warning will not appear. You can test the value of the built-in **ErrorCode** value to see if the graphic imported successfully.

Code Listing 6-16

```
// Test for an active document and make sure it is named.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  If ActiveDoc.Name = ''
    MsgBox 'Save the file and run the script again.    ';
    LeaveSub;
  Else
    Set vCurrentDoc = ActiveDoc;
EndIf

// Initialize the ErrorCode variable to 0 (Success).
Set ErrorCode = 0;

// Import the graphic.
Import File File('Warning.eps') AlertUserAboutFailure(False);
```

```
// Test the ErrorCode value.
If ErrorCode not= 0
  Write Console 'The graphic could not be imported.';
  Write Console 'Error message: '+ErrorMsg;
EndIf
```

If **ErrorCode** is equal to 0 (zero), the built-in **ErrorMsg** variable will be set to "Success"; otherwise, it will be set to an error message (which is usually vague).

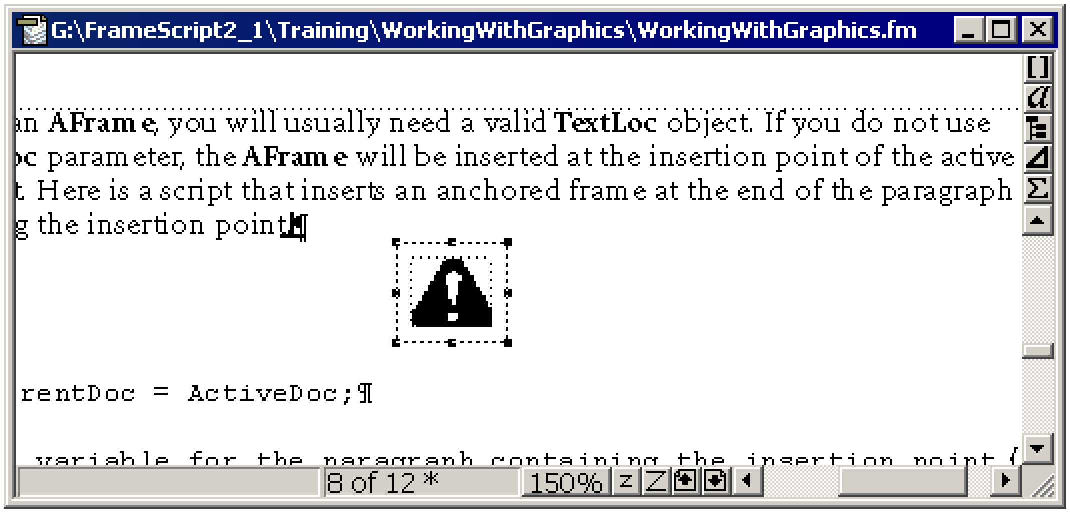
You will usually want to specify a location for the imported graphic. If you use the **TextLoc** parameter, an anchored frame will be created at the text location and the graphic will be imported into the anchored frame. Here is a script that imports a graphic at the end of the paragraph containing the insertion point in the active document.

Code Listing 6-17

```
Set vCurrentDoc = ActiveDoc;

// Make sure the insertion point is in a paragraph.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Please click in a paragraph and run the script again.';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = vCurrentDoc.TextSelection.Begin.Object;

// Make a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);

// Initialize the ErrorCode variable to 0 (Success).
Set ErrorCode = 0;

// Import the graphic at the text location.
Import File File('Warning.eps') AlertUserAboutFailure(False)
  TextLoc(vTextLoc);

// Test the ErrorCode value.
If ErrorCode not= 0
  Write Console 'The graphic could not be imported.';
  Write Console 'Error message: '+ErrorMsg;
EndIf
```

If the graphic is successfully imported, the anchored frame will remain selected (see the illustration below). You can use the document's **FirstSelectedGraphicInDoc** property to modify the anchored frame after you import the graphic.

```
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
```



G:\FrameScript2_1\Training\WorkingWithGraphics\WorkingWithGraphics.fm

n **AFrame**, you will usually need a valid **TextLoc** object. If you do not use
c parameter, the **AFrame** will be inserted at the insertion point of the active
t. Here is a script that inserts an anchored frame at the end of the paragraph
g the insertion point.¶

```
rentDoc = ActiveDoc;¶

variable for the paragraph containing the insertion point.
```

8 of 12 * 150%

If you want to modify the imported graphic, you can use the anchored frame variable (**vAFrame**) to access it. The new anchored frame will only contain the imported graphic, so you can use **vAFrame.FirstGraphicInFrame** to get the graphic object.

```
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
Set vGraphic = vAFrame.FirstGraphicInFrame;
```

This is necessary because the **Import File** command does not return a variable for the imported graphic.

Here is an example where both the anchored frame and the graphic objects are used after the graphic is imported. This code “shrinkwraps” the anchored frame to the size of the graphic, and places it outside of the text column at the left of the paragraph.

Code Listing 6-18

```
Set vCurrentDoc = ActiveDoc;

// Make sure the insertion point is in a paragraph.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Please click in a paragraph and run the script again.';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = vCurrentDoc.TextSelection.Begin.Object;

// Make a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);

// Initialize the ErrorCode variable to 0 (Success).
Set ErrorCode = 0;

// Import the graphic at the text location.
Import File File('Warning.eps') AlertUserAboutFailure(False)
  TextLoc(vTextLoc);
```

```
// Test the ErrorCode value.
If ErrorCode not= 0
  Write Console 'The graphic could not be imported.';
  Write Console 'Error message: '+ErrorMsg;
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the selected anchored frame.
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
// Set a variable for the imported graphic.
Set vGraphic = vAFrame.FirstGraphicInFrame;

// Shrinkwrap the anchored frame and change its location.
Set vAFrame.Height = vGraphic.Height+2pt;
Set vAFrame.Width = vGraphic.Width+2pt;
Set vGraphic.LocX = 1pt;
Set vGraphic.LocY = 1pt;
Set vAFrame.AnchorType = AnchorSubColLeft;
Set vAFrame.SideOffset = 12pt;
Set vAFrame.BaselineOffset = 20pt;
```

Notice that we used the dimensions of the imported graphic (**vGraphic**) to determine the height and width of the anchored frame.

Importing Bitmaps

There are two parameters on the **Import File** command that are of interest when importing bitmapped graphics, such as BMP or TIF files. One, is the **GraphicDpi** parameter. If you don't specify a value, the graphic will be imported at 72 dpi (dots per inch). The second is the **FitGraphicInSelectedRect** parameter. You must set this to **False** or else the graphic will be scaled to 1 inch square when it is imported. The script below imports a TIF file at the beginning of the paragraph containing the insertion point, and preserves its original size.

Code Listing 6-19

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Import the graphic.
Import File File('Warning.tif') TextLoc(vPgf)
  AlertUserAboutFailure(False)
  GraphicDpi(300) FitGraphicInSelectedRect(False);
```

Importing By Reference or By Copy

The **HowToImport** parameter in the **Import File** command determines if the graphic is imported by reference or by copy. Use **DoByRef** to import by reference; use **DoByCopy** to import by copy.

Importing a Graphic into an Existing Frame

In the examples in this section, we have been importing a graphic at a text location in a paragraph. This automatically creates the anchored frame to hold the graphic. There may be cases where you want to import a graphic into an existing frame. To do this, you must get the frame object and select it. Then the graphic will import into the selected frame. Here is an example, assuming that **vAFrame** is the frame where you want the graphic to go.

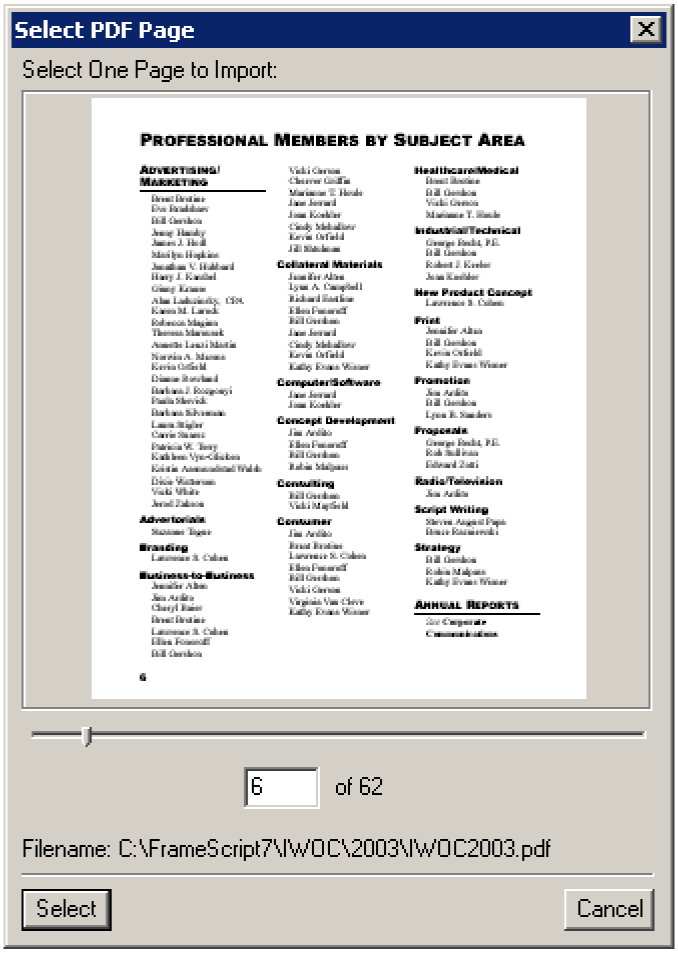
Code Listing 6-20

```
// Select the anchored frame.
Set vAFrame.GraphicIsSelected = 1;

// Import the graphic into the frame.
Import File File('Warning.tif') AlertUserAboutFailure(False)
  GraphicDpi(300) FitGraphicInSelectedRect(False);
```

Because we want the graphic to import into the selected frame, we do not use the **TextLoc** parameter on the **Import File** command.

Importing Multi-page PDF files

When you import a multi-page PDF file into FrameMaker, you are prompted with the Select PDF Page dialog box. This allows you to select which page of the PDF file that you want to import.



Select PDF Page

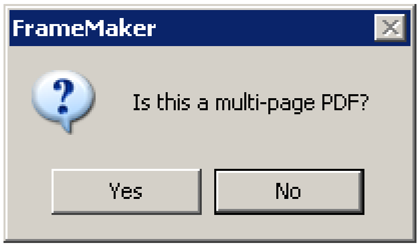
Select One Page to Import:

6 of 62

Filename: C:\FrameScript7\IWOC\2003\IWOC2003.pdf

This dialog box cannot be scripted with FrameScript, and you there is no parameter on the **Import File** command to determine which page is imported. However, you can partially script this by making a loop that will continually prompt you with the Select PDF Page dialog until you click Cancel.

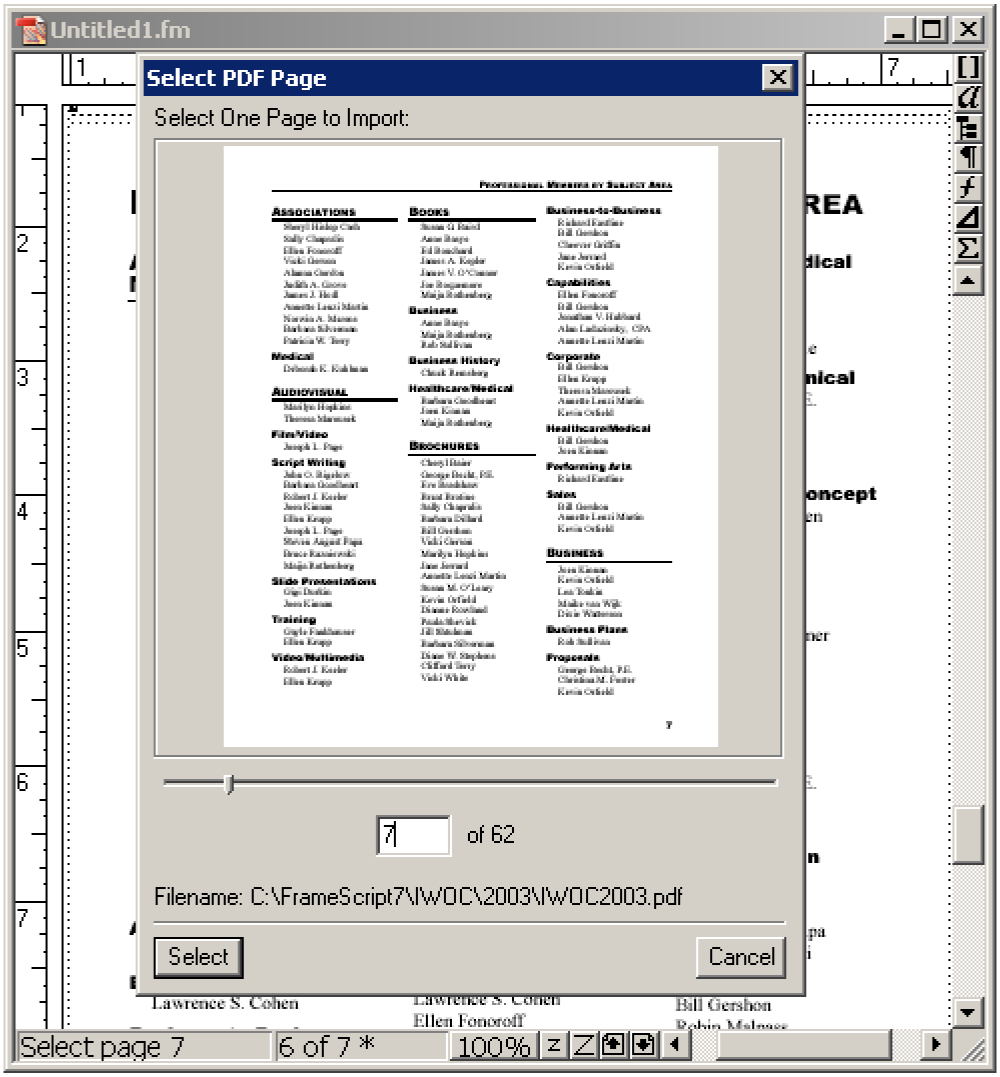
This script begins with the paragraph at the insertion point. It prompts the user to choose a PDF file to import. It then asks the user if the PDF contains more than one page.



FrameMaker

Is this a multi-page PDF?

If the user chooses Yes, the script will use a loop to continually prompt the user with the Select PDF Page dialog box. The loop will continue to prompt and import pages until the user clicks Cancel. Each time the Select PDF Page dialog box appears, the page number field defaults back to 1. We will use a counter to keep track of the next page to import. The value of the counter will be displayed in the document's status area in the lower, left-hand corner. The user will have to enter the page number in the dialog box, but they can refer to the status area to see what number should be input.



Untitled1.fm

Select PDF Page

Select One Page to Import:

7 of 62

Filename: C:\FrameScript7\IWOC\2003\IWOC2003.pdf

Select page 7 | 6 of 7 * | 100%

Here is the complete script.

Code Listing 6-21

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the active document.
  Set vDoc = ActiveDoc;
EndIf

// Set a variable for the current insertion point.
Set vPgf = TextSelection.Begin.Object;

// Prompt the user for a PDF file.
DialogBox Type(ChooseFile) Mode(SelectFile)
  Title('Please choose a PDF file:')
  Init('*.pdf') NewVar(vPdfFile) Button(vButton);
If vButton = CancelButton
  LeaveSub;
EndIf

// Ask the user if this is a multipage PDF.
MsgBox Mode(NoYes) Button(vButton)
  'Is this a multi-page PDF?    ';

// Initialize a counter variable.
Local n = 1;

// Attempt to import the graphic.
Set ErrorCode = 0
Loop While(ErrorCode = 0) LoopVar(n) Init(1) Incr(1)
  If n > 1
    // Insert a paragraph for the graphic.
    New Paragraph NewVar(vPgf) PrevObject(vPgf);
  EndIf
  // Display the current PDF page number to import.
  Set vDoc.StatusLine = 'Select page '+n;
  // Import the PDF.
  Import File File(vPdfFile) TextLoc(vPgf);
  // Here is where you will get the PDF page prompt.
  If ErrorCode = 0
    If vButton = NoButton
      LeaveLoop;
    EndIf
  EndIf
EndLoop

// Restore the status line.
Set vDoc.StatusLine = '';
```

Working With Graphics In Frames

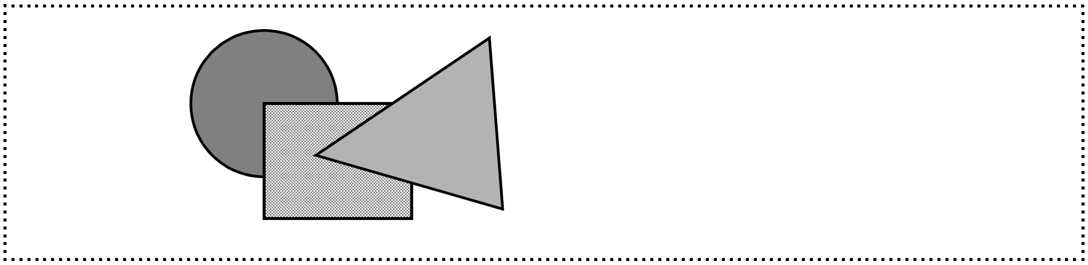
Each FrameMaker document contains a list of the graphic objects that it contains; the first code example in this chapter shows how to loop through all of the graphics in a document. In addition, each *frame* contains a list of graphics that are inside of it. A frame can be an **AFrame** (anchored frame), **UFrame** (unanchored frame), or the special **PageFrame** object on each page. Objects drawn directly on a page are contained by the page's **PageFrame** object.

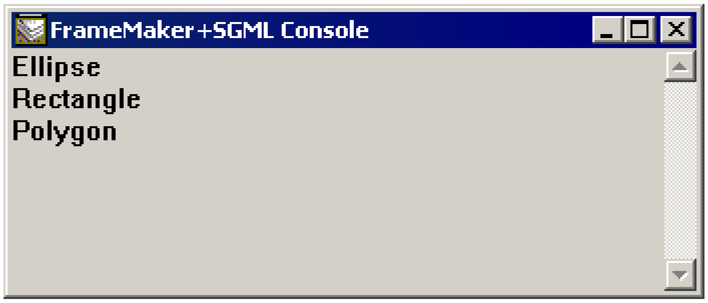
You will often want to work with graphics that are in a particular frame. To do this, you must first get the frame object, and then its **FirstGraphicInFrame** object. Here is a script that loops through all of the graphics in the selected anchored frame. It writes each graphic's **ObjectName** to the Console.

Code Listing 6-22

```
Set vCurrentDoc = ActiveDoc;

// Test for a selected anchored frame.
If vCurrentDoc.FirstSelectedGraphicInDoc.ObjectName not= 'AFrame'
  // Display a message and exit the script.
  MsgBox 'Please select an anchored frame and rerun the script.    ';
  LeaveSub;
EndIf

Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
// See if the anchored frame is empty.
If vAFrame.FirstGraphicInFrame = 0
  // Display a message and exit the script.
  MsgBox 'There are no graphics in the selected frame.    ';
  LeaveSub;
EndIf

Set vGraphic = vAFrame.FirstGraphicInFrame;
// Loop through the graphics in the frame.
Loop While(vGraphic)
  Write Console vGraphic.ObjectName;
  Set vGraphic = vGraphic.NextGraphicInFrame;
EndLoop
```

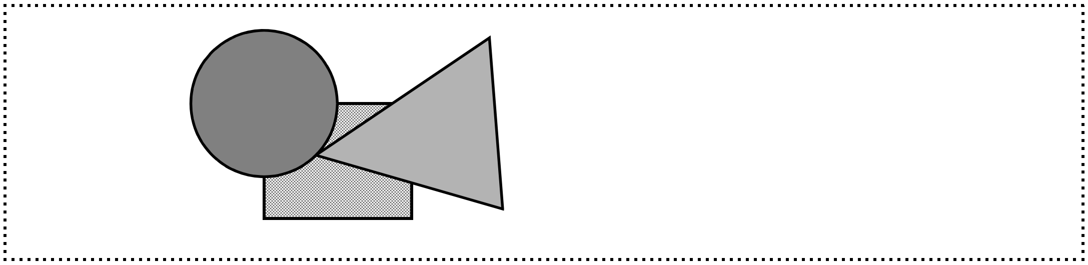
Once you get the **FirstGraphicInFrame**, the loop moves from graphic to graphic using the **NextGraphicInFrame** property. The order of graphics in a frame corresponds to the "draw order" of the objects, from the back to the front. To illustrate this, add a few objects to an anchored frame and run the previous script. You will see that the first object you added will be the **FirstGraphicInFrame** and that the others will follow in the order they were drawn.

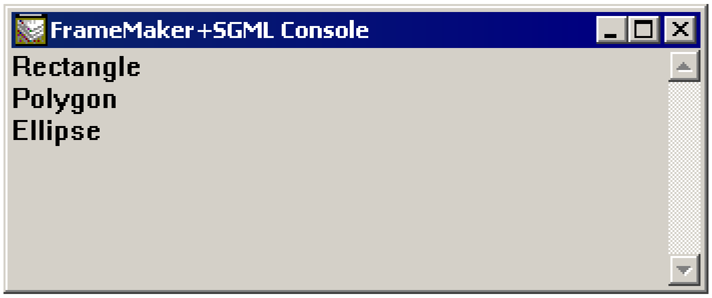




Ellipse
Rectangle
Polygon

It is possible to change the draw order of objects after they are created. You can do this in the interface by selecting a graphic and choosing **Graphic** > **Bring to Front** or **Graphic** > **Send to Back**. In the illustration below, the **Ellipse** has been brought to the front and is now considered last in the draw order.





Rectangle
Polygon
Ellipse

You can also start with the **LastGraphicInFrame** property (last in the draw order) and move through the graphics by using **PrevGraphicInFrame** property. Simply change the following lines in the script.

Code Listing 6-23

```
Set vGraphic = vAFrame.LastGraphicInFrame;
// Loop through the graphics in the frame.
Loop While(vGraphic)
  Write Console vGraphic.ObjectName;
  Set vGraphic = vGraphic.PrevGraphicInFrame;
EndLoop
```

We mentioned the interface commands to change the order of graphics in a frame. This can also be done with FrameScript code. The following script makes sure that the largest **Inset** (imported graphic) is the backmost object in the selected frame.

Code Listing 6-24

```
Set vCurrentDoc = ActiveDoc;
Set vAFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
```

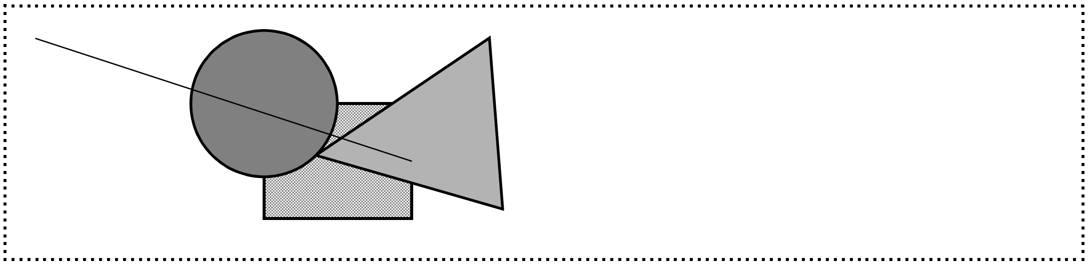
```
// Initialize a variable for the largest inset.
Set vLargestInset = 0;
// Initialize a variable for the size.
Set vLargestSize = 0;

Set vGraphic = vAFrame.FirstGraphicInFrame;
// Loop through the graphics in the frame.
Loop While(vGraphic)
  // Test for an imported graphic.
  If vGraphic.ObjectName = 'Inset'
    // Calculate the size in square points.
    Set vSize = vGraphic.Height * vGraphic.Width;
    // Compare the size with the current largest size.
    If vSize > vLargestSize
      // Set a variable for the inset.
      Set vLargestInset = vGraphic;
      // Store the size as the current largest size.
      Set vLargestSize = vSize;
    EndIf
  EndIf
  // Move to the next graphic in the frame.
  Set vGraphic = vGraphic.NextGraphicInFrame;
EndLoop

// If Insets were found, move the largest to the back.
If vLargestInset
  Set vLargestInset.PrevGraphicInFrame = 0;
EndIf
```

You make an object the “back-most” object in its parent frame by setting its **PrevGraphicInFrame** property to **0** (zero). To make an object the “front-most” object, set its **NextGraphicInFrame** property to **0** (zero). Note that you do not have the parent frame object to reorder an object; you only need the object of the graphic you want to move. The following code works with the **FirstSelectedGraphicInDoc** object.

Code Listing 6-25

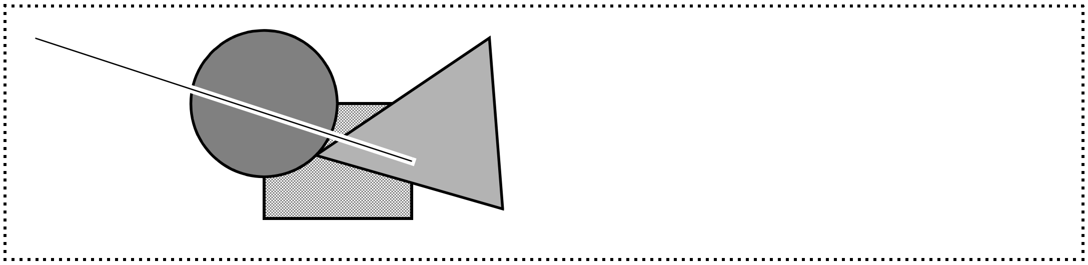
```
// Set a variable for the first selected graphic in doc.
Set vGraphic = FirstSelectedGraphicInDoc;

// Move it to the front of its parent frame.
Set vGraphic.NextGraphicInFrame = 0;
```

You can also position an object directly in front of or behind another object in a frame. Here is a script that makes a white shadow of a selected line and places is directly behind the line.



Code Listing 6-26

```
// Set a variable for the selected line.
Set vLine = FirstSelectedGraphicInDoc;

// Make a property list, starting with the line properties.
New PropertyList NewVar(vShadowProps);
Set vShadowProps = vLine.Properties;

// Remove the pen and border width properties.
Remove Property(Pen) From(vShadowProps);
Remove Property(BorderWidth) From(vShadowProps);

// Add the specific shadow properties.
Add Property To(vShadowProps) BorderWidth(3pt) Pen(FillWhite);

// Make the shadow line.
New Line NewVar(vShadow) ParentObject(vLine.FrameParent);

// Apply the shadow properties to the new line.
Set vShadow.Properties = vShadowProps;

// Move the shadow directly behind the line.
Set vShadow.NextGraphicInFrame = vLine;
```



7 Working With Tables

The ability to make tables is a powerful feature of FrameMaker. This chapter provides information on using FrameScript to work with FrameMaker tables.

To work with the examples in this chapter, make a new FrameMaker document and choose **FrameScript** > **Script Window**. To run code in the Script Window, choose **Run** > **Script**, press Ctrl+R, or click the Run button (the last button on the right).

Inserting Tables

To insert a table, use the **New Table** command. Click in the document and try the following code.

Code Listing 7-1

```
// Insert a Format A table at the insertion point.
New Table NewVar(vTbl) Format('Format A');
```

This will insert a Format A table at the insertion point of the document. If there is no insertion point in the document, no table will be inserted. You use a valid table format name for the **Format** parameter. For example, if the Format A table format does not exist in your document, no table will be inserted.

You can specify a location the a new table by using the **TextLoc** parameter. Add a few paragraphs to your sample document, click in the margin so that there is no insertion point, and run the following script. It will add a table at the end of the first paragraph of the document.

Code Listing 7-2

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Set a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);
// Insert the table at the text location.
New Table NewVar(vTbl) Format('Format A') TextLoc(vTextLoc);
```

You can specify the number of columns, and the number and type of rows, when you use the **New Table** command. Delete the table from your document and run the code below.

Code Listing 7-3

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
// Set a variable for the first paragraph in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Set a text location at the end of the paragraph.
New TextLoc NewVar(vTextLoc) Object(vPgf) Offset(ObjEndOffset-1);
// Insert the table at the text location.
New Table NewVar(vTbl) Format('Format A') TextLoc(vTextLoc)
  NumCols(3) BodyRows(2) HeaderRows(1) FooterRows(1);
```

If you leave off any of the row or column parameters, the default value stored with the table format will be used. You should specify values for these parameters in your scripts. Because a FrameMaker table must have at least one body row, you cannot use **BodyRows(0)** in your code; if you do, the table will not be created.

You can use FrameScript's built-in **ErrorCode** variable to see if your code sucessfully inserted the table.

Code Listing 7-4

```
// Initialize ErrorCode to 0 before using it.
Set ErrorCode = 0;
// Attempt to insert a table at the insertion point.
New Table NewVar(vTbl) Format('Format A')
  NumCols(3) BodyRows(0) HeaderRows(1) FooterRows(1);
If ErrorCode not= 0
  // Display an error message.
  MsgBox 'The table was not inserted.'+CHARLF+
    'Error code: '+ErrorCode+CHARLF+'Error message: '+ErrorMsg;
Else
  // Display a sucess message.
  MsgBox 'The table was inserted.'+CHARLF+
    'Error code: '+ErrorCode+CHARLF+'Error message: '+ErrorMsg;
EndIf
```

Try the above code to see what happens with the **BodyRows** parameter set to 0 (zero). Then change **BodyRows** to a legal value, change the **Format** parameter to an invalid name, and try the script. Finally, make sure you have all legal values and run the code.

Admittedly, the **ErrorMsg** message does not always give useful information. For more information, you can look up the **ErrorCode** number in the *FrameScript Scriptwriter's Reference*.

A simpler approach is to test for the **NewVar** variable to see if a table was successfully inserted.

Code Listing 7-5

```
// Attempt to insert a table at the insertion point.
New Table NewVar(vTbl) Format('Format A')
  NumCols(3) BodyRows(0) HeaderRows(1) FooterRows(1);
// See if the new table exists.
If vTbl = 0
  // Display an error message.
  MsgBox 'The table was not inserted.    ';
Else
  // Display a sucess message.
  MsgBox 'The table was inserted.    ';
EndIf
```

Working With a Selected Table

FrameMaker documents have a property called **SelectedTbl** (selected table) that indicates if a table is selected in the document. A selected table can either contain the insertion point or have one or more cells selected. Click in a table in your document and run this code.

Code Listing 7-6

```
Set vCurrentDoc = ActiveDoc;
// See if there is a selected table.
If vCurrentDoc.SelectedTbl = 0
  // If there is no selected table, display a message.
  MsgBox 'There is no selected table.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the selected table.
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Display the table object.
Display vTbl;
```

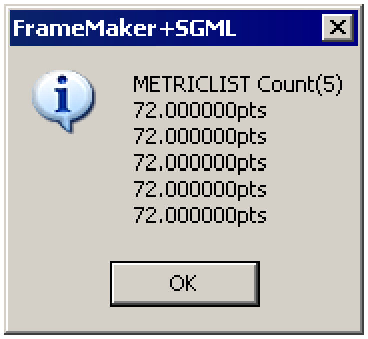
You can use the **SelectedTbl** property to help you test much of the remaining code in this chapter. It allows you to test command on a selected table in your document.

Resizing Table Columns

A table's column widths are stored in a metric list property called **TblColWidths**. Click in a table in your document and run this code.

Code Listing 7-7

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

Display vTbl.TblColWidths;
```



METRICLIST Count(5)
72.000000pts
72.000000pts
72.000000pts
72.000000pts
72.000000pts

To change the width of one or more cells, you apply an updated MetricList to the table's **TblColWidth** property. The following example will change each of the table column widths to 1/2 inch. We are assuming that our sample table has five columns.

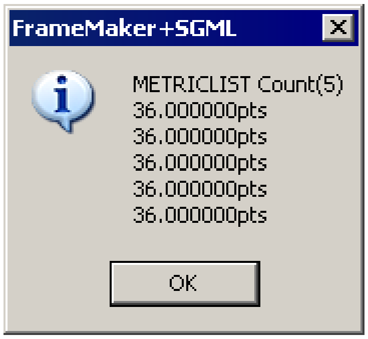
Code Listing 7-8

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Make a new metric list.
New MetricList NewVar(vTblColWidths);
// Add the width of each column to the list.
Add Member(.5in) To(vTblColWidths);
Add Member(.5in) To(vTblColWidths);
Add Member(.5in) To(vTblColWidths);
Add Member(.5in) To(vTblColWidths);
Add Member(.5in) To(vTblColWidths);

// Apply the table column widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
```

You can display the new widths; notice that metric lists always uses points as the display unit (.5 inch = 36 points).

Code Listing 7-9

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

Display vTbl.TblColWidths;
```



```
METRICLIST Count(5)
36.000000pts
36.000000pts
36.000000pts
36.000000pts
36.000000pts
```

In this case, we assumed that the table had five columns. The number of members in your metric list does not have to match the number of columns in the table. For example, if your table has six columns and your metric list has five members, only the first five columns of the table will be resized. Your metric list can also contain more members than the number of columns in the table.

If you want to make sure that all of your table columns are the same width, regardless of the number of columns, you can use the **TblNumCols** property to make sure that your metric list has the correct number of members. **TblNumCols** is the number of columns in the table. Try this code on a table.

Code Listing 7-10

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Set a variable for number of columns in the table.
Set vTblNumCols = vTbl.TblNumCols;
// Make a new metric list.
New MetricList NewVar(vTblColWidths);
// Use a loop to add the width of each column to the list.
Loop While(n <= vTblNumCols) LoopVar(n) Init(1) Incr(1)
  Add Member(.5in) To(vTblColWidths);
EndLoop

// Display the metric list.
Display vTblColWidths;

// Apply the table column widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
```

You will see that when the metric list is displayed, the number of members will be the same as the number of columns in the table.

There are times when you only want to change the widths of certain columns in the table, while leaving the rest as they are. Changing the first column’s width is easy; just make a one-member metric list and apply that to the table.

Code Listing 7-11

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Make a new metric list with one member.
New MetricList NewVar(vTblColWidths) Value(.5in);

// Apply the table column widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
```

To change the widths of columns to the right of the first one, you have to figure out how to maintain the current widths of the columns that you don't want to change. The best approach is make a metric list from the existing table column widths, and manipulate the appropriate values. For example, let's assume that we want to change the width of the fourth column to 1/2 inch, while leaving the rest of the column widths as they are.

Code Listing 7-12

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Make a new metric list with the existing table column widths.
Set vTblColWidths = vTbl.TblColWidths;

// Replace the fourth member of the list with the new value.
Replace Member Number(4) In(vTblColWidths) With(.5in);

// Apply the table column widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
```

There is one pitfall to this code: if the selected table only has three columns, the **Replace Member** command will fail. We can test for this before the command, and exit if necessary.

Code Listing 7-13

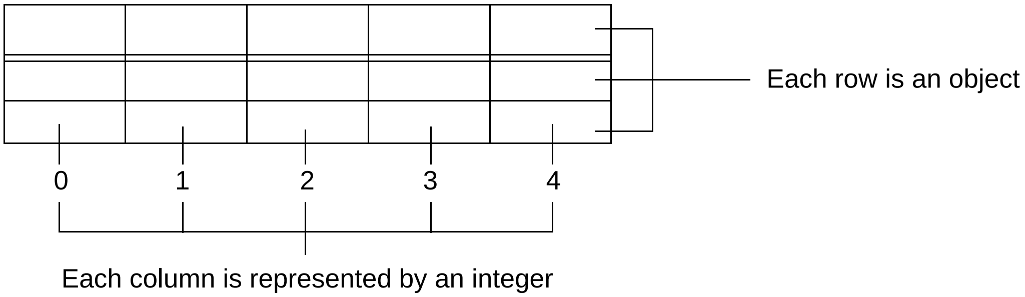
```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf
```

```
// If the table has less than four columns, exit the script.
If vTbl.TblNumCols < 4
  LeaveSub;
EndIf

Set vTblColWidths = vTbl.TblColWidths;
Replace Member Number(4) In(vTblColWidths) With(.5in);
Set vTbl.TblColWidths = vTblColWidths;
```

Adding Rows and Columns

Before we look at the syntax for adding rows and columns to tables, let's discuss how FrameMaker represents rows and columns in a table. Each row in a table is a separate object made up of cell objects. Conversely, table columns are not objects, but are represented by integers starting with 0 (zero). The first column is 0, the second column is 1, etc. The last column is the number of columns in the table minus 1 (**TblNumCols-1**).

Table 1:



This is important because when you add body rows to a table, you must supply the **Row** object where the new rows will be inserted. When you add columns, you supply the number of the column where the new columns will be inserted.

New TableRows

To add rows to a table use the **New TableRows** command. Here is the basic syntax.

Code Listing 7-14

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Set a variable for the second row in the table.
Set vRow = vTbl.FirstRowInTbl.NextRowInTbl;

// Add one row below the second row in the table.
New TableRows TableObject(vTbl) RowObject(vRow)
  HeaderRows(1) Direction(Below);
```

In this script we are adding a row below the second row in the table, represented by the **vRow** variable. The **vRow** variable is used as the **RowObject** parameter. The **Direction** parameter indicates where the new rows are added, either **Above** or **Below** the **RowObject**. The **BodyRows** parameter is the number of new rows that are added to the table.

To add header or footer rows to a table, you supply integer values to the **HeaderRows** or **FooterRows** parameter. Here is a script that adds two header rows and one footer row to the selected table.

Code Listing 7-15

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Add two header rows and one footer row to the table.
New TableRows TableObject(vTbl) HeaderRows(2) FooterRows(1);
```

Notice that you do not use the **RowObject** or **Direction** parameters when adding new header or footer rows to a table. New header rows are always added below any existing header rows, and footer rows are added below existing footer rows. The **RowObject** and **Direction** parameters are only used with the **BodyRows** parameter.

New TableCols

Use the **New TableCols** command to add new columns to a table. Here is a script that adds a new column at the right the second column in the selected table.

Code Listing 7-16

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Add a table column to the right of the second column.
New TableCols TableObject(vTbl) StartCol(1) Direction(Right)
  NumCols(1);
```

Remember that columns in a table are numbered from the left, starting with 0 (zero), so the second column in the table will be column 1. This is what we used in the **StartCol** parameter. We used **Right** for the **Direction** parameter; the other choice is **Left** when you want to add the new columns to the left of the start column.

Deleting Rows and Columns

Deleting table rows and columns is similar to adding them. For rows, you supply the **Row** object of the first row you want to delete, and number of rows to delete. For columns, you supply the starting column number, and how many columns you want to delete.

Delete TableRows

The following script deletes the last two rows in the table.

Code Listing 7-17

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Set a variable for the second last row in the table.
Set vRow = vTbl.LastRowInTbl.PrevRowInTbl;

// Delete the last two rows in the table.
Delete TableRows TableObject(vTbl) RowObject(vRow) NumRows(2);
```

Note that a table must have at least one body row, so this code will fail if the selected table only has two body rows. If your code would leave the table with no body rows, the **Delete TableRows** command will fail completely, and no rows will be deleted from the table.

Delete TableColumns

This script deletes the first column from the selected table.

Code Listing 7-18

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Delete the first column in the table.
Delete TableColumns TableObject(vTbl) StartCol(0) NumCols(1);
```

There are several things to note about this command.

- If **StartCol** = 0 and **NumCols** is equal to the number of columns in the table, the entire table will be deleted.
- If **StartCol** = 0 and **NumCols** is greater than the number of columns in the table, the command will fail and no columns will be deleted.
- If **NumCols** is greater than the number of columns in the table to the right of **StartCol**, the command will fail and no columns will be deleted. For example, if you have a 5-column table, and use **StartCol(1)** and **NumCols(5)**, the command will fail. In general, **StartCol** + **NumCols** must be less than or equal to the total number of columns in the table (**TblNumCols**) in order for the **Delete TableColumns** command to work.

Working with Table Properties

Tables have four kinds of objects that you use to manipulate table properties:

- **Tbl** is the table object itself, which gives you access to general table properties, such as those that are available in the Table Designer.
- A table can have an optional table title, which is represented by the table's **FirstPgf** property.
- Each table has one or more **Row** objects, where you can control row properties, such as minimum and maximum row height.
- Each row has one or more **Cell** objects. There are many properties associated with cells, including custom ruling and shading, rotation, and straddles. Cells also contain **Pgf** (paragraph) objects that contain the table's content, including text and graphics.

The following sections discuss table, table title, and row properties. Cell properties will be covered separately in a later section.

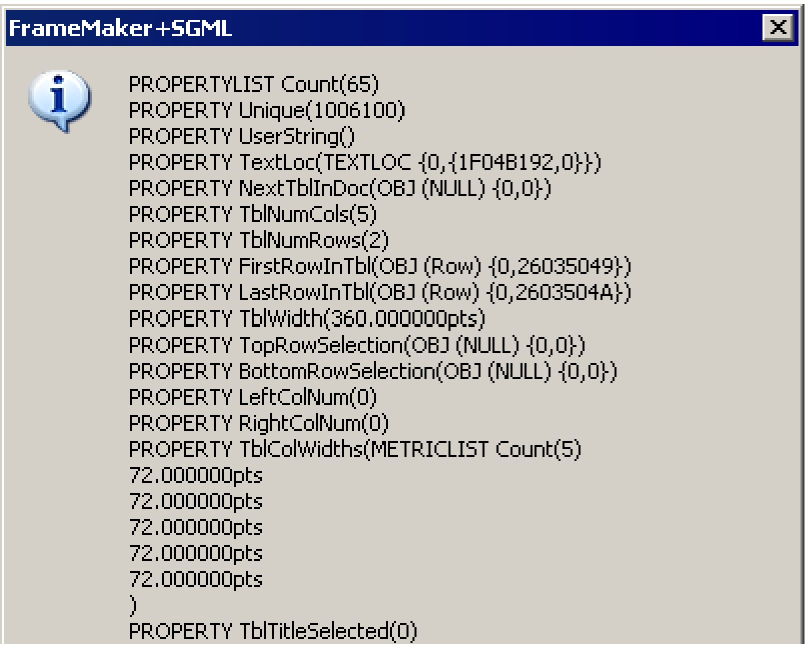
Getting and Setting Table Properties

To get a table's properties, you first need to get the **Tbl** object. As in the previous sections, we will use the document's **SelectedTbl** property to work on the selected table. See "Finding Tables in a Document" on page 7-24 to see how to work with any table in the document.

You can see a selected table's properties by using this code.

Code Listing 7-19

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Display the table's properties.
Display vTbl.Properties;
```



```
PROPERTYLIST Count(65)
PROPERTY Unique(1006100)
PROPERTY UserString()
PROPERTY TextLoc(TEXTLOC {0,{1F04B192,0}})
PROPERTY NextTblInDoc(OBJ (NULL) {0,0})
PROPERTY TblNumCols(5)
PROPERTY TblNumRows(2)
PROPERTY FirstRowInTbl(OBJ (Row) {0,26035049})
PROPERTY LastRowInTbl(OBJ (Row) {0,2603504A})
PROPERTY TblWidth(360.000000pts)
PROPERTY TopRowSelection(OBJ (NULL) {0,0})
PROPERTY BottomRowSelection(OBJ (NULL) {0,0})
PROPERTY LeftColNum(0)
PROPERTY RightColNum(0)
PROPERTY TblColWidths(METRICLIST Count(5)
72.000000pts
72.000000pts
72.000000pts
72.000000pts
72.000000pts
)
PROPERTY TblTitleSelected(0)
```

The entire property list will not fit on the screen; press the Enter key to dismiss the dialog box. Instead, you can use the following code to write the properties to the Console window.

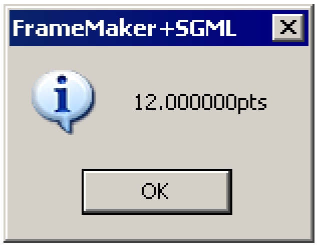
Code Listing 7-20

```
Write Console vTbl.Properties;
```

To display a particular property, use the property name in place of **.Properties**. For example, to display the Space Above setting of the selected table, use this.

Code Listing 7-21

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Display the table's Space Above property.
Display vTbl.TblSpaceAbove;
```



```
12.000000pts
```

To change a table property, use the **Set** command to assign the new property value. The following script changes the selected table's Space Above property to zero.

Code Listing 7-22

```
Set vCurrentDoc = ActiveDoc;
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Change the space above to 0.
Set vTbl.TblSpaceAbove = 0pt;
```

Note that when you change a single table like this, you are usually creating an override of the table's Table Format. If you want to make a change to all the tables with a particular Table Format, make the change to the **TblFmt** (Table Format) object, and to all of the tables in the document that use that format. Here is an example that changes the space above property on all of the Format A tables.

Code Listing 7-23

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  Set vCurrentDoc = ActiveDoc;
EndIf

// Get the object for the Format A table format.
Get Object Type(TblFmt) Name('Format A') DocObject(vCurrentDoc)
  NewVar(vTblFmt);
// Make sure the table format exists.
If vTblFmt
  // Set the Space Above setting to zero.
  Set vTblFmt.TblSpaceAbove = 0pt;
EndIf

// Apply the change to all of the Format A tables in the document.
// Loop through all of the tables in the document.
Loop ForEach(Tbl) In(vCurrentDoc) LoopVar(vTbl)
  // Test for the Format A format.
  If vTbl.TblTag = 'Format A'
    // Apply the change to the table.
    Set vTbl.TblSpaceAbove = 0pt;
  EndIf
EndLoop
```

Some table properties are read-only, in other words, they can't be directly changed using the **Set** command. One example is the **TblNumCols** property, which gives the number of columns in the table. You cannot use this to change the number of columns in the table.

```
// Bad code; attempting to set a read-only property.
Set vTbl.TblNumCols = 6;
```

Instead, you must use the **New TableCols** or **Delete TableColumns** commands to change the **TblNumCols** property.

Getting and Setting the Table Title Properties

A table's title is represented by the table's **FirstPgf** property. If a table has no title, its **FirstPgf** property will be **0** (zero). Before attempting the change table title properties, you should test that the **FirstPgf** exists.

Code Listing 7-24

```
If SelectedTbl.FirstPgf
  MsgBox 'The selected table has a title.    ';
Else
  MsgBox 'The selected table does not have a title.    ';
EndIf
```

The **FirstPgf** property of a table is actually a **Pgf** (paragraph) object. As such, you can manipulate its properties like you do with any other paragraph object.

You can use a table's **TblTitlePosition** property to add or remove the table title, or to change its position. Here are the possible values.

- **TblNoTitle**. The table has no title.
- **TblTitleAbove**. The table title is above the table.
- **TblTitleBelow**. The table title is below the table.

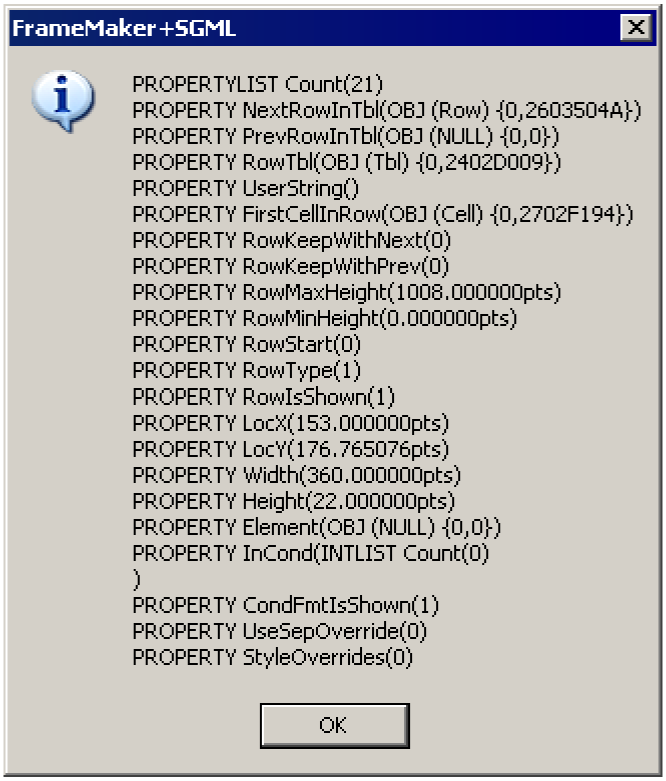
A table's **TblTitleGap** property determines the distance between the table title and the table.

Getting and Setting Row Properties

To get and set a row's properties in a table, you must get the **Row** object. For information on how to find a particular Row in a table, see "Finding Objects in a Table" on page 15. Here is a script that shows the Row properties of the first row of the selected table.

Code Listing 7-25

```
Set vCurrentDoc = ActiveDoc;

If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Set a variable for the first row in the table.
Set vRow = vTbl.FirstRowInTbl;

// Display the row properties.
Display vRow.Properties;
```



```
PROPERTYLIST Count(21)
PROPERTY NextRowInTbl(OBJ (Row) {0,2603504A})
PROPERTY PrevRowInTbl(OBJ (NULL) {0,0})
PROPERTY RowTbl(OBJ (Tbl) {0,2402D009})
PROPERTY UserString()
PROPERTY FirstCellInRow(OBJ (Cell) {0,2702F194})
PROPERTY RowKeepWithNext(0)
PROPERTY RowKeepWithPrev(0)
PROPERTY RowMaxHeight(1008.000000pts)
PROPERTY RowMinHeight(0.000000pts)
PROPERTY RowStart(0)
PROPERTY RowType(1)
PROPERTY RowIsShown(1)
PROPERTY LocX(153.000000pts)
PROPERTY LocY(176.765076pts)
PROPERTY Width(360.000000pts)
PROPERTY Height(22.000000pts)
PROPERTY Element(OBJ (NULL) {0,0})
PROPERTY InCond(INTLIST Count(0)
)
PROPERTY CondFmtIsShown(1)
PROPERTY UseSepOverride(0)
PROPERTY StyleOverrides(0)
```

You can use the **Set** command to change **Row** properties, except for read-only properties, such as **Width** and **Height**. Here is a script that sets the third row in the selected table to start at the top of the page.

Code Listing 7-26

```
Set vCurrentDoc = ActiveDoc;

If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Find the third row in the table.
Set vRow = vTbl.FirstRowInTbl;
Loop While(n < 3) LoopVar(n) Init(1) Incr(1)
  Set vRow = vRow.NextRowInTbl;
  If vRow = 0
    // If there are less than three rows in the table, exit.
    LeaveSub;
  EndIf
EndLoop

// Set the row to start at the top of the page.
Set vRow.RowStart = RowTopOfPage;

// Update the document's hyphenation so that the change is applied.
Update DocObject(vCurrentDoc) Hyphenating;
```

You may find it curious that we update the document's hyphenation with the last line of the script. This is to overcome a bug that prevents the row from moving after we change its **RowStart** setting.

Finding Objects in a Table

Before discussing table cells, it will be helpful to see how to navigate to rows and cells in a table. Once you have a **Tbl** object, there are two "entrance points" to the rows in the table. One is the **FirstRowInTbl** property; the other is the **LastRowInTbl** property. When you have one of these object, you can move row-to-row through the table. Here is a loop that moves through the table rows of a selected table, beginning with the first row.

Code Listing 7-27

```
Set vCurrentDoc = ActiveDoc;

If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;
Loop While(vRow)
  // Write the row object to the Console window.
  Write Console vRow;
  // Move to the next row in the table.
  Set vRow = vRow.NextRowInTbl;
EndLoop
```

You can also start with the last row in the table and move up through the table rows.

Code Listing 7-28

```
Set vCurrentDoc = ActiveDoc;

If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the last row in the table.
Set vRow = vTbl.LastRowInTbl;
Loop While(vRow)
  // Write the row object to the Console window.
  Write Console vRow;
  // Move to the previous row in the table.
  Set vRow = vRow.PrevRowInTbl;
EndLoop
```

A table row can be one of three types: header, body, or footer. You can test what kind of **Row** object you have by checking its **RowType** property. Often, you will only want to process one kind of row in your scripts. Here is a script that skips heading rows, but processes body and footer rows in the selected table.

Code Listing 7-29

```
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;
// Find the first body row in the table.
Loop While(vRow.RowType = RowHeading)
  // Move to the next row in the table.
  Set vRow = vRow.NextRowInTbl;
EndLoop

// Now loop through the rest of the rows in the table.
Loop While(vRow)
  Write Console vRow;
  Set vRow = vRow.NextRowInTbl;
EndLoop
```

When you have a **Row** object, you can then move to the **Cell** objects in the row. Here is a script that gets the first **Cell** object in the first row of the selected table. It then loops through all of the cells in the row.

Code Listing 7-30

```
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;

// Get the first cell in the row.
Set vCell = vRow.FirstCellInRow;
// Loop through all of the cells in the row.
Loop While(vCell)
  // Write the cell object to the Console.
  Write Console vCell;
  // Move to the next cell in the row.
  Set vCell = vCell.NextCellInRow;
EndLoop
```

You can combine a loop through the rows with a loop through the cells to touch every cell in the table. You do this by nesting the **Cell** loop inside the **Row** loop.

Code Listing 7-31

```
Set vCurrentDoc = ActiveDoc;
```

```
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;
Loop While(vRow)
  // Get the first cell in the row.
  Set vCell = vRow.FirstCellInRow;
  // Loop through the cells in the current row.
  Loop While(vCell)
    // Write the cell object to the Console.
    Write Console vCell;
    // Move to the next cell in the current row.
    Set vCell = vCell.NextCellInRow;
  EndLoop
  // Move to the next row in the table.
  Set vRow = vRow.NextRowInTbl;
EndLoop
```

Here is a shorthand loop that starts with the first **Cell** and moves from cell-to-cell through the selected table. This uses the cell's **NextCellInTbl** property.

Code Listing 7-32

```
Set vCurrentDoc = ActiveDoc;

If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;
// Get the first cell in the row.
Set vCell = vRow.FirstCellInRow;
// Loop through all the cells in the table.
Loop While(vCell)
  // Write the cell object to the Console.
  Write Console vCell;
  // Move to the next cell in the table.
  Set vCell = vCell.NextCellInTbl;
EndLoop
```

Once you have a **Cell** object, you can move down and up in the same table column by using the cell's **CellBelowInCol** or **CellAboveInCol** properties. The following script starts at the second cell in the first row and moves cell-to-cell down the second column of the selected table.

Code Listing 7-33

```
Set vCurrentDoc = ActiveDoc;
```

```
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub;
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Get the first row in the table.
Set vRow = vTbl.FirstRowInTbl;
// Get the second cell in the row.
Set vCell = vRow.FirstCellInRow.NextCellInRow;
// Loop through all the cells in the column.
Loop While(vCell)
  // Write the cell object to the Console.
  Write Console vCell;
  // Move to the next cell in the column.
  Set vCell = vCell.CellBelowInCol;
EndLoop
```

The previous code has shown how to move from a table down to the cell level. It is also possible to move "outward" from a **Cell** object to the **Row** and **Tbl** that contains the cell. To get the **Row** object of the current **Cell**, use this code.

```
// Get the row object of the current cell.
Set vRow = vCell.CellRow;
```

To get the **Tbl** object of the current **Cell**, use this.

```
// Get the table object of the current row.
Set vTbl = vRow.RowTbl;
```

You can combine the code to go directly from the **Cell** object to the **Tbl** object.

```
// Get the table object of the cell.
Set vTbl = vCell.CellRow.RowTbl;
```

If your insertion point is in a table cell, you can go directly to the **Cell** object without going through the **Tbl** or **Row** objects containing the **Cell**.

Code Listing 7-34

```
// Make a text location at the insertion point.
Set vTextLoc = TextSelection.Begin;
// Find the text object containing the insertion point.
Set vCell = vTextLoc.InTextObj;
```

A table cell can contain one or more paragraphs (unless the cell is straddled). You can navigate through a cell's paragraphs in order by using its **FirstPgf** property and a loop. This code loops through the paragraphs in the selected cell.

Code Listing 7-35

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;
```

```
// Find the first paragraph in the cell.
Set vPgf = vCell.FirstPgf;
// Loop through the paragraphs in the cell.
Loop While(vPgf)
  // Write the paragraph’s text to the Console.
  Write Console vPgf.Text;
  // Go to the next paragraph in the cell.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

The following script uses the cell’s **LastPgf** property and moves backwards through the cell’s paragraphs.

Code Listing 7-36

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Find the last paragraph in the cell.
Set vPgf = vCell.LastPgf;
// Loop through the paragraphs in the cell.
Loop While(vPgf)
  // Write the paragraph’s text to the Console.
  Write Console vPgf.Text;
  // Go to the previous paragraph in the cell.
  Set vPgf = vPgf.PrevPgfInFlow;
EndLoop
```

Working with Cells

Cells are the building blocks of tables and contain a majority of a table’s content. This section provides information on working with table cells.

Adding Text

Every cell that is not straddled contains at least one paragraph. Non-straddled cells have **FirstPgf** and **LastPgf** objects; in a cell that only has one paragraph, its **FirstPgf** and **LastPgf** objects are the same.

Adding text to a table cell is the same as adding text to any paragraph in FrameMaker; you use the **New Text** command with either the **Object** or **TextLoc** parameter. If the table cell is empty, it is easiest to use the **FirstPgf** property with the **Object** parameter. Click an insertion point in an empty table cell to try the following code.

Code Listing 7-37

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Add text to the cell.
New Text Object(vCell.FirstPgf) 'Adding text to a table cell.';
```

The text is always added at the beginning of the paragraph when using the **Object** parameter. If your paragraph already contains text, the new text will be added in front of the existing text.

You can add text at any location in an existing paragraph by using the **TextLoc** parameter on the **New Text** command.

Code Listing 7-38

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Make a text location at the end of the cell’s first paragraph.
New TextLoc NewVar(vTextLoc) Object(vCell.FirstPgf)
  Offset(ObjEndOffset-1);
// Add the text at the text location.
New Text TextLoc(vTextLoc) ' New text at the end of the paragraph.';
```

You can add paragraphs to a table cell by using the **New Pgf** command. To add a new paragraph to the cell containing the insertion point, use this script.

Code Listing 7-39

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Add a new paragraph after the first one.
New Pgf NewVar(vPgf) PrevObject(vCell.FirstPgf);
```

If you want to make sure that your new paragraph is the last one in the cell, use **vCell.LastPgf** for the **PrevObject** value.

There is no direct way to add a paragraph before the existing **FirstPgf** in a cell, but here is a workaround that uses the clipboard.

Code Listing 7-40

```
// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Make a text range of the existing first paragraph.
New TextRange NewVar(vTextRange) Object(vCell.FirstPgf) Offset(0)
  Offset(ObjEndOffset);

// Select the paragraph.
Set TextSelection = vTextRange;

// Copy the paragraph to the clipboard.
Copy Text;

// Put the insertion point at the beginning of the first paragraph.
Set TextSelection = vCell.FirstPgf;

// Paste the copied paragraph at the beginning of the cell.
Paste Text;

// Make a text range of the first paragraph's text.
New TextRange NewVar(vTextRange) Object(vCell.FirstPgf) Offset(0)
  Offset(ObjEndOffset-1);
```

```
// Delete the duplicated text.
Delete Text TextRange(vTextRange);
```

Rotating Cells

You can rotate a table cell by setting its **CellAngle** property to one of four valid settings: **0**, **90** (clockwise), **180** (upside down), or **270** (counter-clockwise).

Straddling Cells

To straddle or unstraddle table cells, you use the **Straddle TableCells** command. The **CellObject** parameter specifies the cell where the straddle begins. **NumRows** and **NumCols** specifies the number of rows and columns, respectively, in the straddle. The **On** and **Off** options determines whether the command will straddle or unstraddle the specified cells.

The following script straddles two rows and two columns starting with the cell containing the insertion point. Put your cursor in the first cell of the first body row and try the script.

Code Listing 7-41

```
Set vCurrentDoc = ActiveDoc;

// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Straddle the cells.
Straddle TableCells CellObject(vCell) NumRows(2) NumCols(2) On;

// Use this to refresh the screen.
Update DocObject(vCurrentDoc) Hyphenating;
```

To unstraddle cells, use the **Off** option. Keep the cursor in the cell and run this code to remove the straddle.

Code Listing 7-42

```
Set vCurrentDoc = ActiveDoc;

// Find the cell containing the insertion point.
Set vTextLoc = TextSelection.Begin;
Set vCell = vTextLoc.InTextObj;

// Straddle the cells.
Straddle TableCells CellObject(vCell) NumRows(2) NumCols(2) Off;

// Use this to refresh the screen.
Update DocObject(vCurrentDoc) Hyphenating;
```

When you use the **On** option to straddle cells, make sure you specify a combination of cells that would result in a valid straddle. For example, you cannot have a single straddle containing both heading and body row cells.

Although straddled cells appear to be merged into one, all of the original cells still exist as **Cell** objects. You can tell if a cell is straddled by checking its **CellIsStraddled** property. This property will be **True** (1) for all of the cells in a straddle except for the first one (the visible cell).

You can identify the first cell in a straddle by querying the cell's **CellNumRowsStraddled** and **CellNumColsStraddled** properties. If either of these property values are greater than one, then you know that the cell is the visible cell in a straddle. Here is a script that uses these properties to unstraddle all of the cells in a table. Add a few straddles to a selected table and try the script.

Code Listing 7-43

```
Set vCurrentDoc = ActiveDoc;
Set vTbl = SelectedTbl;

// Find the first cell in the table.
Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
// Loop through the cells in the table.
Loop While(vCell)
  // Test the cell to see if it is straddling any rows or columns.
  If (vCell.CellNumRowsStraddled > 1) or
    (vCell.CellNumColsStraddled > 1)
    // Unstraddle the cells.
    Straddle TableCells CellObject(vCell)
      NumRows(vCell.CellNumRowsStraddled)
      NumCols(vCell.CellNumColsStraddled) Off;
  EndIf
  // Move to the next cell in the table.
  Set vCell = vCell.NextCellInTbl;
EndLoop

// Use this to refresh the screen.
Update DocObject(vCurrentDoc) Hyphenating;
```

See "Getting the Correct Objects" on page 2-6 for another example of straddling table cells.

Applying Custom Ruling and Shading

Custom ruling and shading can be applied at the cell level in a table. Custom ruling and shading overrides ruling and shading settings defined by the Table format of the table. Each **Cell** object has the following properties related to ruling and shading.

- **CellOverrideShading.** This is the **Color** object of the cell's custom shading. You use the **Get Object** command to get the object of the color you want.
- **CellOverrideFill.** This is an integer representing the cell's custom fill. For example, use **3** for 50% shading.
- **CellOverrideTopRuling.** This is **RulingFmt** object of the cell's custom top ruling. You use the **Get Object** command to get the object of the ruling format you want.
- **CellOverrideBottomRuling.** This is **RulingFmt** object of the cell's custom bottom ruling. You use the **Get Object** command to get the object of the ruling format you want.
- **CellOverrideLeftRuling.** This is **RulingFmt** object of the cell's custom left ruling. You use the **Get Object** command to get the object of the ruling format you want.

- **CellOverrideRightRuling.** This is **RulingFmt** object of the cell's custom right ruling. You use the **Get Object** command to get the object of the ruling format you want.

The following properties act as toggles for the six properties above. You set the above property or properties to the values you want, and then you set the corresponding property below to **1** (or **True**).

- **CellUseOverrideShading**
- **CellUseOverrideFill**
- **CellUseOverrideTRuling**
- **CellUseOverrideBRuling**
- **CellUseOverrideLRuling**
- **CellUseOverrideRRuling**

Here is a script that sets the custom shading for the selected cell to 50% Red and applies the Double ruling format to the cell's borders.

Code Listing 7-44

```
Set vCurrentDoc = ActiveDoc;
Set vCell = TextSelection.Begin.InTextObj;

// Get the Color object for Red.
Get Object Type(Color) Name('Red') DocObject(vCurrentDoc)
  NewVar(vColor);

// Get the RulingFmt object for Double.
Get Object Type(RulingFmt) Name('Double') DocObject(vCurrentDoc)
  NewVar(vRulingFmt);

// Set the cell’s properties.
Set vCell.CellOverrideShading = vColor; // Red color
Set vCell.CellOverrideFill = 3; // 50% fill
Set vCell.CellOverrideTopRuling = vRulingFmt; // Double ruling fmt
Set vCell.CellOverrideBottomRuling = vRulingFmt;
Set vCell.CellOverrideLeftRuling = vRulingFmt;
Set vCell.CellOverrideRightRuling = vRulingFmt;

// Set the override toggles.
Set vCell.CellUseOverrideShading = 1;
Set vCell.CellUseOverrideFill = 1;
Set vCell.CellUseOverrideTRuling = 1;
Set vCell.CellUseOverrideBRuling = 1;
Set vCell.CellUseOverrideLRuling = 1;
Set vCell.CellUseOverrideRRuling = 1;

// Refresh the screen.
Update DocObject(vCurrentDoc) Redisplay;
```

Many times, you will want to remove custom ruling and shading from table cells. The easiest way is to loop through the cells, and set the override toggles to **0** (zero) for each cell. Below is a script that does that. It uses a subroutine to set the override toggle settings.

Code Listing 7-45

```
Set vCurrentDoc = ActiveDoc;
Set vTbl = SelectedTbl;

Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
Loop While(vCell)
  // Call the subroutine to remove the overrides.
  Run RemoveCellOverrides;
  Set vCell = vCell.NextCellInTbl;
EndLoop

// Refresh the screen.
Update DocObject(vCurrentDoc) Redisplay;

Sub RemoveCellOverrides
//
// Set the override toggles.
Set vCell.CellUseOverrideShading = 0;
Set vCell.CellUseOverrideFill = 0;
Set vCell.CellUseOverrideTRuling = 0;
Set vCell.CellUseOverrideBRuling = 0;
Set vCell.CellUseOverrideLRuling = 0;
Set vCell.CellUseOverrideRRuling = 0;
//
EndSub
```

Finding Tables in a Document

Thus far, we have worked with the **SelectedTbl** object, which is the selected table in the active document. Most of the time, your scripts will need to work on tables that are not selected. You can use loops to find tables in a document. Here is a simple loop through all of the tables in the active document.

Code Listing 7-46

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set vCurrentDoc = ActiveDoc;
EndIf

// Loop through all of the tables in the document.
Loop ForEach(Tbl) In(vCurrentDoc) LoopVar(vTbl)
  // Write the table object to the Console window.
  Write Console vTbl;
EndLoop
```

If you want to delete any tables in your loop, you have to use a longer version of the loop. In the following example, we want to delete Format A tables from the document.

Code Listing 7-47

```
Set vCurrentDoc = ActiveDoc;
```

```
// Find the first table in the document.
Set vTbl = vCurrentDoc.FirstTblInDoc;
// Loop through the tables.
Loop While(vTbl)
  // Set a variable for the next table in the document.
  Set vNextTbl = vTbl.NextTblInDoc;
  // See if the current table is a Format A table.
  If vTbl.TblTag = 'Format A'
    // Delete the table.
    Delete Object(vTbl);
  EndIf
  // Move to the next table in the document.
  Set vTbl = vNextTbl;
EndLoop
```

Note that the two previous loops do not necessarily move through the tables in the document order. In addition, they also process tables that are on master pages and references pages. You can restrict your loop to tables on specific kinds of pages by adding a test to the loop. The following script only processes tables that are on body pages.

Code Listing 7-48

```
Set vCurrentDoc = ActiveDoc;

// Loop through all of the tables in the document.
Loop ForEach(Tbl) In(vCurrentDoc) LoopVar(vTbl)
  // See if the current table is on a body page.
  If vTbl.Page.ObjectName = 'BodyPage'
    // Write the table object to the Console window.
    Write Console vTbl;
  EndIf
EndLoop
```

When you want to process tables in document order, you need to use a different approach. You loop through the paragraphs in document order, and test each paragraph to see if it contains table anchors. First, here is how to loop through paragraphs in document order. This code assumes that your document has a main text flow, which a document does by default.

Code Listing 7-49

```
Set vCurrentDoc = ActiveDoc;

// Find the first paragraph in the flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through the paragraphs in document order.
Loop While(vPgf)
  // Test for table anchors will go here.
  // ...
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

To test each paragraph in the flow, we use the **Get TextList** command to get a list of table anchors in the paragraph.

```
Get TextList InObject(vPgf) TblAnchor NewVar(vTextList);
```

If the paragraph contains any tables, the **vTextList** variable will contain a list of the table anchors. You will need to loop through the **TextList**, get each **TblAnchor**, and get the **Tbl** object associated with the **TblAnchor**.

```
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
  Set vTbl = vTblAnchor.TextData;
  // Process the table here.
EndLoop
```

Here is the entire code.

Code Listing 7-50

```
Set vCurrentDoc = ActiveDoc;

// Find the first paragraph in the flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through the paragraphs in document order.
Loop While(vPgf)
  // Test for table anchors in the paragraph.
  Get TextList InObject(vPgf) TblAnchor NewVar(vTextList);
  // Loop through the tables.
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
    Set vTbl = vTblAnchor.TextData;
    // Process the table here.
    // ...
  EndLoop
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

You can also get a **TextList** of table anchors in a text flow without looping through each paragraph. This will return the tables in the main text flow in document order.

Code Listing 7-51

```
Set vCurrentDoc = ActiveDoc;
Set vMainFlow = vCurrentDoc.MainFlowInDoc;

// Test for table anchors in the main text flow.
Get TextList InObject(vMainFlow) TblAnchor NewVar(vTextList);
// Loop through the tables.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
  Set vTbl = vTblAnchor.TextData;
  // Process the table here.
  // ...
EndLoop
```

Tutorial 7-1: Scaling Tables to Fit the Text Column Width

Our fictitious client uses two different templates for his documents. His online template has a wider text column than his paper template. When importing formats from one template to another, the width of the tables does not change to fit the new text column width. He would like a simple script that will proportionately scale each table to the same width as the text column. He can then run this script after importing formats.

Devising a Strategy

It is usually best to devise a strategy for solving the problem before thinking about the actual FrameScript syntax. This helps you to develop an *algrorithm* for the solution. An algorithm is a plan for solving the problem; it is the logic of your solution without the programming language syntax. Once you have a plan, you can fill in the appropriate FrameScript commands. This approach is especially useful when you are learning the language because it allows you to develop a solution without getting bogged down with unfamiliar language syntax. As you get more familiar with FrameScript, you will find yourself combining these steps by writing code as you develop the algorithm.

You will need some mathematics to determine how to scale the table. You can determine a scaling factor by dividing the column width by the table's current width. Here is the formula.

Scale factor = Column width / Table width

For example, if the column width is 5 inches and the table is 4 inches, the table will have to be scaled by a factor of 1.25 (125%). If the table is wider than the text column, the scale factor will be less than 1, because the table width needs to be reduced.

Once we have the scale factor, we can determine the new width of the table.

New table width = Current table width * Scale factor

You can try the formula with the example we used above.

4 (current table width) * 1.25 (scale factor) = 5 (new table width)

As you can see, this gives the desired result: the new table width is equal to the text column width. You may want to verify the formulas with a few other example values.

Getting the Appropriate Data

Now that we have the basic algorithm, we can attempt to find the FrameScript commands and properties that we need. This will take a trip to the *FrameScript Scriptwriter's Reference* (and some persistance). Here are the necessary values for the first formula.

Code Listing 7-52

```
// The selected table.
Set vTbl = SelectedTbl;

// The current width of the table.
Set vTblWidth = vTbl.TblWidth;

// The width of the column containing the table.
Set vColWidth = vTbl.TextLoc.Object.InTextObj.Width;
```

The last line is a little tricky, so here it is step-by-step.

Code Listing 7-53

```
// The text location where the table is anchored.
Set vTextLoc = vTbl.TextLoc;
// The paragraph containing the text location.
Set vPgf = vTextLoc.Object;
// The text object (sub column) containing the paragraph.
Set vSubCol = vPgf.InTextObj;
// The width of the sub column.
Set vColWidth = vSubCol.Width;
```

Admittedly, this isn't very intuitive on the surface. Sometimes it's easier to start with the desired object and work backwards to the object you already have. Here, we need the width of the text column, which is a **SubCol** (subcolumn) object. If you look up **SubCol** properties, you will see that there is a **Width** property. Subcolumns contain paragraphs, so you can look at **Pgf** properties to see if there is a property which determines which subcolumn the paragraph is in. This is the **InTextObj** property. We can determine the **Pgf** containing the table by looking at the **Object** property of the text location containing the table anchor. The text location of the table anchor is indicated by the **TextLoc** property of the table.

As it turns out, a **TextLoc** also has an **InTextObj** property, so we can skip the **Pgf** object altogether. This revised code

Code Listing 7-54

```
// The text location where the table is anchored.
Set vTextLoc = vTbl.TextLoc;
// The text object (sub column) containing the text location.
Set vSubCol = vTextLoc.InTextObj;
// The width of the sub column.
Set vColWidth = vSubCol.Width;
```

which gives us this.

```
// The width of the column containing the table.
Set vColWidth = vTbl.TextLoc.InTextObj.Width;
```

It may take some time to figure out how to get to the correct property. Don't be discouraged; with experience you will become familiar with finding what you need.

With this information, we can now fill in the first formula with FrameScript syntax. Click in a sample table and run this code.

Code Listing 7-55

```
// The selected table.
Set vTbl = SelectedTbl;

// The current width of the table.
Set vTblWidth = vTbl.TblWidth;

// The width of the column containing the table.
Set vColWidth = vTbl.TextLoc.InTextObj.Width;

// The first formula: find a scale factor.
Set vScaleFactor = vColWidth / vTblWidth;
```

Scaling the Table

The table’s **TblWidth** property is read-only, so you can’t use this to scale the table.

```
// This won’t work.
Set vTbl.TblWidth = vTblWidth * vScaleFactor;
```

Instead, you have to change the width of each individual column of the table. These widths are stored in a metric list called **TblColWidths**. A metric list is a list of measurement values; in this case, there is a member of the list for each column of the table. If a table has five columns, the **TblColWidth** list will contain five members. Each member contains the width of the corresponding table column.

What we need to do is scale each member of the list by the scale factor. We do this by following these steps.

1. Duplicate the **vTbl.TblColWidths** list by assigning it to a **vTblColWidths** variable.
2. Loop through the **vTblColWidths** list and do the following for each member.
   a. Extract the member from the list.
   b. Multiple the value by the scale factor.
   c. Replace the member with the new value.
3. Finally, apply the **vTblColWidths** variable to the **vTbl.TblColWidth** property.

Here is the completed script. Give it a try on the selected table.

Code Listing 7-56

```
Set vTbl = SelectedTbl;
Set vTblWidth = vTbl.TblWidth;
Set vColWidth = vTbl.TextLoc.InTextObj.Width;

// Find the scale factor.
Set vScaleFactor = vColWidth / vTblWidth;

// Set a variable for the table column widths.
Set vTblColWidths = vTbl.TblColWidths;

// Loop through the list and calculate each column’s new width.
Loop While(n <= vTblColWidths.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTblColWidths) NewVar(vWidth);
  // Scale the individual width.
  Set vWidth = vWidth * vScaleFactor;
  // Put the new width back into the list.
  Replace Member Number(n) In(vTblColWidths) With(vWidth);
EndLoop

// Apply the new widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
```

Making it Work for All Tables in a Document

The easiest way to do this is to put the above code in a subroutine. Then you can loop through the tables in the document and call the subroutine for each one. Here is the completed script that scales all of the tables on body pages in the active document.

Code Listing 7-57

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  Set vCurrentDoc = ActiveDoc;
EndIf

// Turn off displaying to prevent flicker and speed the script.
Set Displaying = 1;

// Loop through the tables in the document.
Loop ForEach(Tbl) In(vCurrentDoc) LoopVar(vTbl)
  // Make sure the table is on a body page.
  If vTbl.Page.ObjectName = 'BodyPage'
    // Scale the table.
    Run ScaleTable;
  EndIf
EndLoop

// Restore the display and refresh the screen.
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;

Sub ScaleTable
//
// Set a variable for the current table width.
Set vTblWidth = vTbl.TblWidth;
// Set a variable for the subcolumn width.
Set vColWidth = vTbl.TextLoc.InTextObj.Width;
// Determine the scale factor.
Set vScaleFactor = vColWidth / vTblWidth;

// Set a variable for the table column widths.
Set vTblColWidths = vTbl.TblColWidths;
// Loop through the table column widths.
Loop While(n <= vTblColWidths.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTblColWidths) NewVar(vWidth);
  // Multiply the current width by the scale factor.
  Set vWidth = vWidth * vScaleFactor;
  // Update the list with the new width.
  Replace Member Number(n) In(vTblColWidths) With(vWidth);
EndLoop

// Apply the updated widths to the table.
Set vTbl.TblColWidths = vTblColWidths;
//
EndSub
```

8 Markers, Cross-References, and Variables

FrameScript provides commands and options for working with FrameMaker markers, cross-references, and variables. A marker is considered an *anchored* object because it is anchored at a location in text. Cross-references and variables are *anchored formatted* objects because you must specify a format when you insert them. You can also work with marker types, and cross-reference and variable formats.

Working With Markers and Marker Types

Inserting Markers

You use the **New Marker** command to insert a marker. The optional **TextLoc** parameter determines where the marker is inserted; if you don't specify a location, the marker is inserted at the current insertion point by default. The **NewVar** parameter returns the object variable of the new marker.

Code Listing 8-1

```
// Insert a marker at the insertion point.
New Marker NewVar(vMarker);
```

The optional **MarkerName** parameter is used to specify the marker type. If you don't specify a marker type, an Index marker is created by default. The **MarkerText** parameter specifies the marker text.

Code Listing 8-2

```
// Insert a Hypertext marker at the insertion point.
New Marker NewVar(vMarker) MarkerName('Hypertext')
  MarkerText('alert Hypertext marker    ');
```

You can either set the parameters at the time you use the **New Marker** command (as above), or you can set them after you insert the marker. The next script shows how to set parameters after the marker has been created.

Code Listing 8-3

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Insert a marker at the insertion point.
New Marker NewVar(vMarker);
// Set the marker type.
Get Object Type(MarkerType) Name('Hypertext') NewVar(vMarkerType)
  DocObject(vCurrentDoc);
Set vMarker.MarkerTypeId = vMarkerType;
// Set the marker text.
Set vMarker.MarkerText = 'alert Hypertext marker    ';
```

IMPORTANT: Be careful not to set a new marker to a marker type that doesn't exist in the document. FrameMaker will crash when you attempt to edit the marker if the marker type doesn't exist. To avoid this, you should test that the marker type exists before creating the marker.

Markers are limited to 255 characters. While you can use FrameScript to insert more than 255 characters in a marker, only the first 255 characters will be used.

Making New Marker Types

You can use FrameScript to make custom marker types by using the **New MarkerType** command. Here is a script that inserts a marker with a custom marker type. If the marker type does not exist in the document, the script creates it.

Code Listing 8-4

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Insert a marker at the insertion point.
New Marker NewVar(vMarker);

// Get the marker type object.
Get Object Type(MarkerType) Name('Command') NewVar(vMarkerType)
  DocObject(vCurrentDoc);
// Make sure the marker type exists; if not, create it.
If vMarkerType = 0
  New MarkerType Name('Command') NewVar(vMarkerType);
EndIf

// Set the marker type.
Set vMarker.MarkerTypeId = vMarkerType;
// Set the marker text.
Set vMarker.MarkerText = 'Commands:New Marker';
```

Changing Marker Types

You can use FrameScript to change a marker from one marker type to another. In the following example, we are changing all Command markers to Index markers in the active document.

Code Listing 8-5

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the object variable for the Index marker type.
Get Object Type(MarkerType) Name('Index') DocObject(vCurrentDoc)
  NewVar(vMarkerType);

// Loop through the markers in the document.
Loop ForEach(Marker) In(vCurrentDoc) LoopVar(vMarker)
  // See if the marker is a Command marker.
  If vMarker.MarkerTypeId.Name = 'Command'
    // Change the marker type to Index.
    Set vMarker.MarkerTypeId = vMarkerType;
  EndIf
EndLoop
```

Working With Cross-References

We will discuss two types of FrameMaker cross-references: paragraph and spot cross-references. (The third type, element cross-references, is not discussed in this book.) When you insert a spot cross-reference in the FrameMaker interface, you have to insert a marker first, followed by the cross-reference. When you insert a paragraph cross-reference, FrameMaker inserts the cross-reference marker for you. Spot and paragraph cross-references also use different settings in the FrameMaker Cross-Reference dialog box.

With FrameScript, both types of cross-references are treated basically the same, in that you have to insert the Cross-Ref marker (if it doesn't already exist) regardless of what kind of cross-reference you are creating. Then you insert a cross-reference object (**XRef**) that points to the Cross-Ref marker (**Marker**). The **XRef** object has an **XRefSrcText** property that must *exactly* match the marker text (including case) of the Cross-Ref marker. The only difference between the cross-reference types in FrameScript is the syntax of the Cross-Ref **MarkerText** (as you will see later).

Spot Cross-References

Spot cross-references are a little a simpler to understand, so we will start with them. As an example, let's assume that we want to insert a set of cross-references to Heading1 paragraphs in a document. This list of cross-references will be used as chapter table of contents. For simplicity, the cross-references will be inserted below the paragraph containing the insertion point.

If you want to try these examples, open a document that contains a few Heading1 paragraphs. To start, you should make sure that the insertion point is in a paragraph. Then, you can make a loop to process the Heading1 paragraphs in document order. The following code simply writes the text of each Heading1 paragraph to the Console.

Code Listing 8-6

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make sure there is an insertion point.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'There is no insertion point in a paragraph.    ';
  LeaveSub; // Exit the script.
Else
  // Make a variable for the paragraph.
  Set vXRefPgf = vCurrentDoc.TextSelection.Begin.Object;
EndIf

// Loop through the paragraphs in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.Name = 'Heading1'
    // Write the Heading1 text to the Console.
    Write Console vPgf.Text;
  EndIf
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Inserting each cross-reference will consist of three steps: First, we will insert a cross-reference marker in the Heading1 paragraph. The marker text will be the text of the heading paragraph. Second, we will insert a new paragraph for the cross-reference. Finally, we will insert the cross-reference into the new paragraph. Here is the code in a subroutine.

Code Listing 8-7

```
Sub InsertXRef
//
// Insert the cross-reference marker in the Heading1 paragraph.
New Marker NewVar(vMarker) TextLoc(vPgf) MarkerName('Cross-Ref')
  MarkerText(vPgf.Text);

// Insert the cross-reference paragraph.
New Pgf NewVar(vXRefPgf) PrevObject(vXRefPgf)
  PgfFmtName('Bulleted');

// Insert the cross-reference into the new paragraph.
New XRef NewVar(vXRef) TextLoc(vXRefPgf) Format('Heading & Page');
// Set the cross-reference source text to match the marker text.
Set vXRef.XRefSrcText = vMarker.MarkerText;
//
EndSub
```

To put it together, add the subroutine to the bottom of the first script and replace these lines

```
// Write the Heading1 text to the Console.
Write Console vPgf.Text;
```

with a call to the subroutine.

Code Listing 8-8

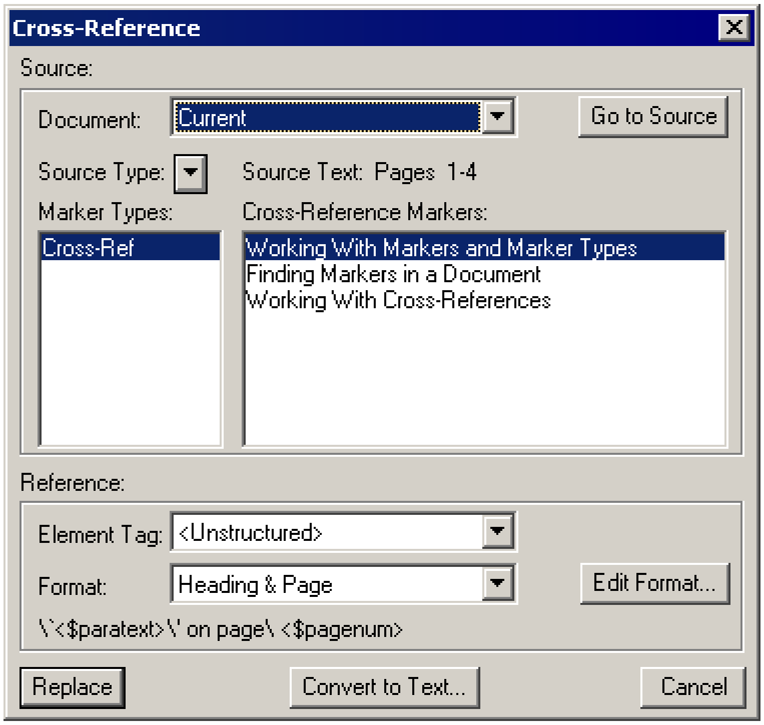
```
// Call the subroutine to insert the marker and cross-reference.
Run InsertXRef;
```

At the end of the body of the script, add a line to update the cross-references.

Code Listing 8-9

```
Update DocObject(vCurrentDoc) XRefs Everything;
```

Run the script on a sample document containing several Heading1 paragraphs. If you double-click on one of the cross-references, you will see that FrameMaker recognizes it as a spot cross-reference.



Cross-Reference

Source:

Document: Current

Source Type:

Source Text: Pages 1-4

Marker Types:

Cross-Ref

Cross-Reference Markers:

Working With Markers and Marker Types
Finding Markers in a Document
Working With Cross-References

Reference:

Element Tag: <Unstructured>

Format: Heading & Page

\`<$paratext>\' on page\ <$pagenum>

As you work with this code, there are a few things to remember.

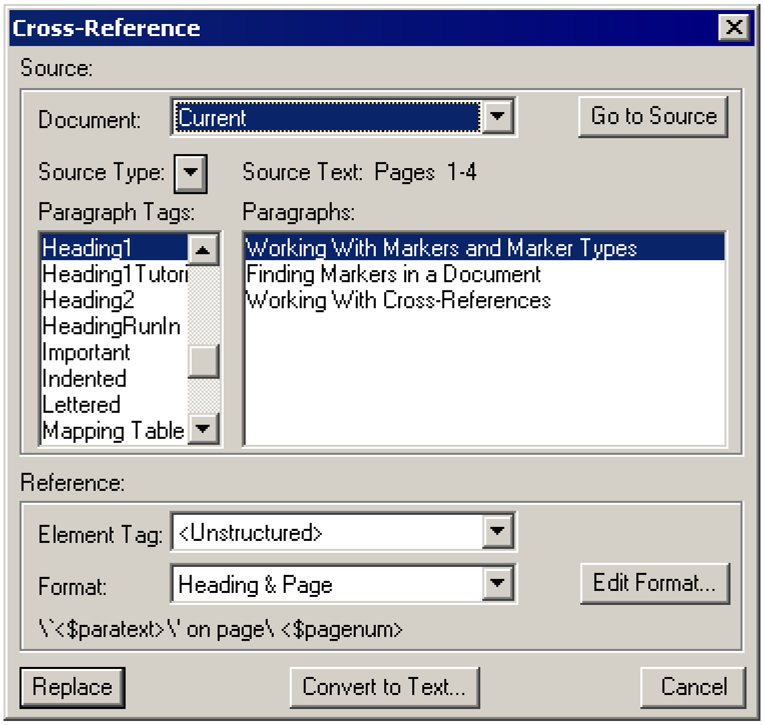
- The **XRefSrcText** property of the cross-reference must *exactly* match the **MarkerText** of the cross-reference marker. This is the same for both spot and paragraph cross-references.
- Each time you run the script, additional redundant markers will be added to the Heading1 paragraphs. In a production script, you should test for the existence of a cross-reference marker before adding a new one.
- We used the Heading & Page cross-reference format, but you might want to create and use a different format for a chapter table of contents.
- New cross-references will be added every time you run the script. In a production script, you would want to delete existing cross-references before adding new ones.

Paragraph Cross-References

As we mentioned earlier, spot and paragraph cross-references differ in the FrameMaker interface. When you insert a spot cross-reference with FrameMaker, you have to insert a marker *before* you insert the cross-reference. The marker text is the "spot" that appears in the Cross-Reference dialog box.

In contrast, paragraph cross-references do not require the insertion of the marker ahead of time; instead, you point to the desired paragraph in the Cross-Reference dialog box, and FrameMaker inserts both the marker and cross-reference in the appropriate places.

Insert a paragraph cross-reference in a document and double-click on it to display the Cross-Reference dialog box. Compare it to the previous screenshot to see how spot and paragraph cross-references are displayed differently.



Document: Current

Source Type:

Source Text: Pages 1-4

Paragraph Tags:

- Heading1
- Heading1Tutori
- Heading2
- HeadingRunIn
- Important
- Indented
- Lettered
- Mapping Table

Paragraphs:

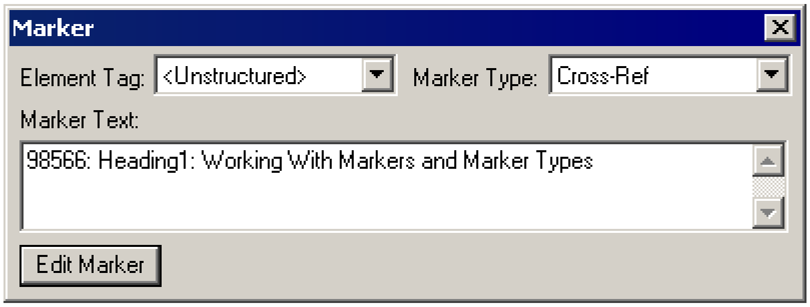
- Working With Markers and Marker Types
- Finding Markers in a Document
- Working With Cross-References

Element Tag: <Unstructured>

Format: Heading & Page

\‘<$paratext>\’ on page\ <$pagenum>

What makes the difference between the cross-reference types? It is the different syntax in the cross-reference markers. To see this, Control+Alt+Click (Control+Option+Click on the Macintosh) to go to the source and open the marker window. You will see the syntax of the marker that FrameMaker inserts with a paragraph cross-reference.



Element Tag: <Unstructured>

Marker Type: Cross-Ref

Marker Text:

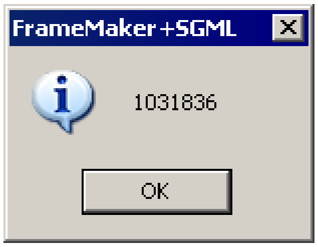
98566: Heading1: Working With Markers and Marker Types

Like the spot cross-references we inserted earlier, you can see the text of the source paragraph in the marker. However, in a paragraph cross-reference, the paragraph text is preceded by a five-digit number (followed by a colon and space), and the name of the paragraph format of the source paragraph (followed by a colon and space). It is these two additional components in the XRefSrcText property of the XRef that causes FrameMaker to “see” this as a paragraph cross-reference, as opposed to a spot cross-reference.

In order to insert paragraph cross-references with FrameScript, we must duplicate this special syntax when inserting the markers and cross-references. To experiment with the code to do this, click in a Heading1 paragraph. Almost every object in FrameMaker has a **Unique** property, which is like a serial number for the object. With your cursor in the paragraph, run this code.

Code Listing 8-10

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
// Display the unique "serial number" for the paragraph.
Display vPgf.Unique;
```

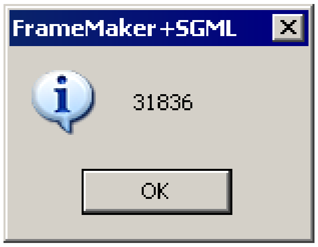


1031836

Because these numbers are unique, the actual number you see will be different. The Unique property is usually six or seven digits long, but never less than five digits. To get the 5-digit number for the cross-reference marker, use the following code. It will convert the integer to a string and drop the first characters. This code will work with any number greater than 5 digits.

Code Listing 8-11

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;

// Convert the Unique integer to a string.
New String NewVar(vMarkerText) Value(vPgf.Unique);
// Drop the first character to ensure that there are 5 digits.
Get String FromString(vMarkerText) NewVar(vMarkerText)
  StartPos(vMarkerText.Size-4);
Display vMarkerText;
```



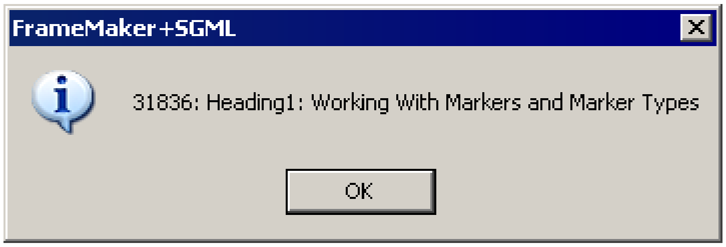
31836

Note that dropping the first digit or two introduces the remote possibility of making the number the same as another paragraph's number. Even if that were to occur, any difference in the paragraphs' text would still make the marker text unique. We can now add the rest of the marker syntax by adding a couple more lines.

Code Listing 8-12

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object;
```

```
// Convert the Unique integer to a string.
New String NewVar(vMarkerText) Value(vPgf.Unique);
// Drop the first character to ensure that there are 5 digits.
Get String FromString(vMarkerText) NewVar(vMarkerText)
  StartPos(vMarkerText.Size-4);
// Add the paragraph tag and text.
Set vMarkerText = vMarkerText + ': ' + vPgf.Name + ': ' + vPgf.Text;
Display vMarkerText;
```



31836: Heading1: Working With Markers and Marker Types

We will put the completed code in a subroutine that we can add to the original spot cross-reference script. We can insert a call to this subroutine from the **InsertXRef** subroutine.

Code Listing 8-13

```
Sub SetMarkerText
//
// Convert the Unique integer to a string.
New String NewVar(vMarkerText) Value(vPgf.Unique);
// Drop the first character to ensure that there are 5 digits.
Get String FromString(vMarkerText) NewVar(vMarkerText)
  StartPos(vMarkerText.Size-4);
// Add the paragraph tag and text.
Set vMarkerText = vMarkerText + ': ' + vPgf.Name + ': ' + vPgf.Text;
//
EndSub
```

Here is the entire script. Before you try it your sample document, make sure you delete the existing cross-references AND cross-reference markers.

Code Listing 8-14

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make sure there is an insertion point.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'There is no insertion point in a paragraph.    ';
  LeaveSub; // Exit the script.
Else
  // Make a variable for the paragraph.
  Set vXRefPgf = vCurrentDoc.TextSelection.Begin.Object;
EndIf
```

```
// Loop through the paragraphs in the main text flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.Name = 'Heading1'
    // Call the subroutine to insert the marker and cross-reference.
    Run InsertXRef;
  EndIf
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop

// Update the cross-references.
Update DocObject(vCurrentDoc) XRefs Everything;

Sub InsertXRef
//
// Call the subroutine to make the paragraph cross-reference syntax.
Run SetMarkerText;

// Insert the cross-reference marker in the Heading1 paragraph.
New Marker NewVar(vMarker) TextLoc(vPgf) MarkerName('Cross-Ref')
  MarkerText(vMarkerText);

// Insert the cross-reference paragraph.
New Pgf NewVar(vXRefPgf) PrevObject(vXRefPgf)
  PgfFmtName('Bulleted');

// Insert the cross-reference into the new paragraph.
New XRef NewVar(vXRef) TextLoc(vXRefPgf) Format('Heading & Page');
// Set the cross-reference source text to match the marker text.
Set vXRef.XRefSrcText = vMarkerText;
//
EndSub

Sub SetMarkerText
//
// Convert the Unique integer to a string.
New String NewVar(vMarkerText) Value(vPgf.Unique);
// Drop the first character to ensure that there are 5 digits.
Get String FromString(vMarkerText) NewVar(vMarkerText)
  StartPos(vMarkerText.Size-4);
// Add the paragraph tag and text.
Set vMarkerText = vMarkerText + ': ' + vPgf.Name + ': ' + vPgf.Text;
//
EndSub
```

External Cross-References

Cross-references from one FrameMaker document to another are known as *external* cross-references. To make a cross-reference point to another document, you set the **XRef**'s **XRefSrcFile** property to the absolute path of the FrameMaker file. On an *internal* cross reference, the **XRefSrcFile** property is an empty string.

Unresolved Cross-References

When you use FrameScript to insert cross-references in a document, you must update the cross-references before they will appear. You can use the **Update** command with the **XRefs** option. For an example, see the end of the body of the previous script. There are several other options that determine which cross-references are updated.

XRefs Update Option	Description
`ForceUpdate`	Updates all cross-references, whether they have changed or not.
`Internal`	Updates internal cross-references.
`OpenDocs`	Updates cross-references to open documents.
`ClosedDocs`	Updates cross-references to closed documents.
`Everything`	Updates all cross-references to all documents.

After you update cross-references, you query each one to see if it is unresolved. Here is a loop that counts resolved and unresolved cross-references in the active document and reports the results to the Console.

Code Listing 8-15

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first cross-reference in the document.
Set vXRef = vCurrentDoc.FirstXRefInDoc;

// If there are no cross-references, display a message and exit.
If vXRef = 0
  MsgBox 'There are no cross-references in this document.    ';
  LeaveSub;
EndIf

// Initialize variables for the cross-reference counts.
Set vResolved = 0;
Set vUnresolved = 0;

// Test the cross-references.
Loop While(vXRef)
  If vXRef.XRefIsUnresolved
    Set vUnresolved = vUnresolved + 1;
  Else
    Set vResolved = vResolved + 1;
  EndIf
  Set vXRef = vXRef.NextXRefInDoc;
EndLoop

// Write the information to the Console.
Write Console 'Unresolved cross-references: '+vUnresolved;
Write Console 'Resolved cross-references: '+vResolved;
```

Unresolved cross-references are usually a result of Cross-References markers that have been accidently deleted. External cross-references can become unresolved because of missing or renamed files. When you use FrameScript to create cross-references, it is up to you to make sure that all of the properties are set correctly.

Working with Cross-Reference Formats

Making Cross-Reference Formats

You can make new cross-reference formats in a document by using the **New XRefFmt** command. You must supply a name for the format using the required **Name** parameter. The **Fmt** property defines the building blocks for the **XRefFmt**.

Code Listing 8-16

```
// Make a new cross-reference format and change its definition.
New XRefFmt Name('ChapterTOC') NewVar(vXRefFmt);
Set vXRefFmt.Fmt = '<$paratext>\t<$pagenum>';
```

To change the definition of an existing cross-reference format, you use the **Get Object** command to get the **XRefFmt** object and then change its **Fmt** property.

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the variable for the cross-reference format object.
Get Object Type(XRefFmt) Name('See Heading & Page')
  DocObject(vCurrentDoc) NewVar(vXRefFmt);

// If the format doesn't exist, then make it.
If vXRefFmt = 0
  New XRefFmt Name('See Heading & Page') NewVar(vXRefFmt);
EndIf

// Set the definition.
Set vXRefFmt.Fmt =
  'For more information, see <$paratext> on page <$pagenum>.';

// Update the cross-references in the document.
Update DocObject(vCurrentDoc) XRefs ForceUpdate Everything;
```

When you change cross-reference formats, you must use the **ForceUpdate** option on the **Update** command to update cross-references that are already resolved. Otherwise, only unresolved cross-references will be updated.

Changing Cross-Reference Formats

You can use FrameScript to change a cross-reference from one cross-reference format to another. In the following example, we are changing all “Heading & Page” cross-references to “See Heading & Page” cross-references in the active document.

Code Listing 8-17

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Get the object variable for the See Heading & Page xref type.
Get Object Type(XRefFmt) Name('See Heading & Page')
  DocObject(vCurrentDoc) NewVar(vXRefFmt);
```

```
// Loop through the cross-references in the document.
Loop ForEach(XRef) In(vCurrentDoc) LoopVar(vXRef)
  // See if the xref is a Heading & Page xref.
  If vXRef.XRefFmt.Name = 'Heading & Page'
    // Change the xref fmt to See Heading & Page.
    Set vXRef.XRefFmt = vXRefFmt;
  EndIf
EndLoop
```

Working with Variables

Inserting Variables

To insert a variable, you use the **New Variable** command with the required **Format** parameter. You will normally use the **TextLoc** parameter to put the variable at a particular location in the text.

Code Listing 8-18

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// See if a table is selected.
If vCurrentDoc.SelectedTbl = 0
  MsgBox 'There is no selected table.    ';
  LeaveSub; // Exit the script.
Else
  Set vTbl = vCurrentDoc.SelectedTbl;
EndIf

// Make sure the table has a title.
If vTbl.FirstPgf = 0
  MsgBox 'The table does not have a title.    ';
  LeaveSub; // Exit the script.
EndIf

// Insert a Table Continuation variable at end of the table title.
// Make a text location.
New TextLoc NewVar(vTextLoc) Object(vTbl.FirstPgf)
  Offset(ObjEndOffset-1);
New Variable Format('Table Continuation') TextLoc(vTextLoc);
```

FrameMaker has two different kinds of variables: system variables and user variables. System variable formats always exist in a document and cannot be deleted. User variable formats must be created before they can be used. For this reason, you should always test for the existence of a user variable format before attempting to insert it in a document. To do this, you use the **Get Object** command.

Code Listing 8-19

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// See if the ProductName user variable format exists.
Get Object Type(VarFmt) Name('ProductName') DocObject(vCurrentDoc)
  NewVar(vVarFmt);
```

```
// If the format doesn't exist, warn the user and exit.
If vVarFmt = 0
  MsgBox 'The ProductName variable format does not exist.    ';
  LeaveSub; // Exit the script.
EndIf

// Insert the variable at the insertion point.
New Variable Format('ProductName');
```

IMPORTANT: Some system variables can only be inserted in background text frames on master pages. See the FrameMaker documentation for more information.

Working With Variable Formats

There are variable formats for two types of variables: system variables and user variables. You can create, edit, and delete user variable formats. You cannot create or delete system variable formats but you can edit their definitions. When you edit system variable formats, you will use the same building blocks that are available in the FrameMaker interface.

Making User Variable Formats

You use the **New VariableFormat** command to create a new user variable format. The **Fmt** parameter is used to set the definition for the variable.

Code Listing 8-20

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// See if the ProductName user variable format exists.
Get Object Type(VarFmt) Name('ProductName') DocObject(vCurrentDoc)
  NewVar(vVarFmt);

// If the format doesn't exist, create it.
If vVarFmt = 0
  New VariableFormat NewVar(vVarFmt) Name('ProductName')
    Fmt('FrameScript 2.1R3');
Else
  // If it exists, change the definition.
  Set vVarFmt.Fmt = 'FrameScript 2.1R3';
EndIf
```

Editing Variable Formats

To edit the definition of an existing variable format, change its **Fmt** property to the desired string. See the previous example script.

Changing Variable Formats

You can use FrameScript to change a variable from one variable format to another. In the following example, we are changing all “Running H/F 1” variables to “Running H/F 2” variables.

Code Listing 8-21

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Get the object variable for the Running H/F 2 variable.
Get Object Type(VarFmt) Name('Running H/F 2')
  DocObject(vCurrentDoc) NewVar(vVarFmt);

// Loop through the variables in the document.
Loop ForEach(Var) In(vCurrentDoc) LoopVar(vVar)
  // See if the variable is a Running H/F 1 variable.
  If vVar.VarFmt.Name = 'Running H/F 1'
    // Change the variable's format to Running H/F 2.
    Set vVar.VarFmt = vVarFmt;
  EndIf
EndLoop
```

Finding Markers, Cross-References, and Variables

In this section, we refer to markers, cross-references, and variables as objects or text items. You will usually need to find objects in a document before you can manipulate them. The easiest way is to use a **Loop ForEach** loop with the appropriate object. Here is an example that loops through all the markers in a document.

Code Listing 8-22

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Loop through all of the markers in the document.
Loop ForEach(Marker) In(vCurrentDoc) LoopVar(vMarker)
  // Write the text of index markers to the Console.
  If vMarker.MarkerTypeId.Name = 'Index'
    Write Console vMarker.MarkerText;
  EndIf
EndLoop
```

If you need to delete objects, you will have to use the long-hand version of the loop. (See “For another method of testing multiple conditions, See “Using a Loop to Handle Multiple Conditions” on page 3-13.” on page 3-3 for more information on loops.) Here is a script that deletes all of the ProductName user variables in the active document.

Code Listing 8-23

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;
```

```
// Set a variable for the first variable in the document.
Set vVar = vCurrentDoc.FirstVarInDoc;
// Loop through all of the variables in the document.
Loop While(vVar)
  // Set a variable for the next variable in the document.
  Set vNextVar = vVar.NextVarInDoc;
  // Test for a ProductName variable.
  If vVar.VarFmt.Name = 'ProductName'
    // Delete the variable.
    Delete Object(vVar);
  EndIf
  // Move to the next variable in the document.
  Set vVar = vNextVar;
EndLoop
```

If you use the previous loops to process markers, cross-references, and variables, they will not necessarily be processed in document order. If you need to process objects in document order, you need to get a text list of the objects in the main flow in the document. With this method, objects that are not in the main flow will be skipped. Here is an example that processes all of the markers in the main text flow.

Code Listing 8-24

```
// Set a variable for the active document and the main flow.
Set vCurrentDoc = ActiveDoc;
Set vMainFlowInDoc = vCurrentDoc.MainFlowInDoc;

// Get a list of the markers in the main flow.
Get TextList InObject(vMainFlowInDoc) MarkerAnchor
  NewVar(vTextList);
// Use a loop to process the list in order.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  // Get the marker anchor from the text list.
  Get Member Number(n) From(vTextList) NewVar(vMarkerAnchor);
  Set vMarker = vMarkerAnchor.TextData;
  // Do something with the marker here.
  // ...
EndLoop
```

A text list (**TextList**) returns a list of text items (**TextItem**); in this case, we are looking for **MarkerAnchor** text items. To find cross-references, you use the **XRefBegin** (or **XRefEnd**) option on the **Get TextList** command. For variables, use the **VarBegin** (or **VarEnd**) option. A text item has three properties:

- **TextData** — For object text items, it is the object variable. For the **MarkerAnchor** text item, it is the **Marker** object associated with the **MarkerAnchor**. For the XRef text items, it is the **XRef** object. For the Var text items, it is the **Var** object.
- **TextOffset** — The offset from the beginning of the **InObject** object; in this case, the main text flow.
- **TextType** — The text item type that was used on the **Get TextList** command (**MarkerAnchor**, **XRefBegin**, **XRefEnd**, **VarBegin**, **VarEnd**).

See “Working With Text Lists and Text Items” on page 5-28 for more information.

An alternative method is to loop through the paragraphs in order and check each paragraph for the appropriate text items.

Code Listing 8-25

```
// Set a variable for the active document and the main flow.
Set vCurrentDoc = ActiveDoc;
Set vMainFlowInDoc = vCurrentDoc.MainFlowInDoc;

// Loop through the paragraphs in the main text flow.
Set vPgf = vMainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  // Get a list of cross-references in the paragraph.
  Get TextList InObject(vPgf) XRefBegin NewVar(vTextList);
  // Use a loop to process the cross-references in paragraph order.
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    // Get the cross-reference text item from the text list.
    Get Member Number(n) From(vTextList) NewVar(vXRefBegin);
    Set vXRef = vXRefBegin.TextData;
    // Do something with the cross-reference here.
    // ...
  EndLoop
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

The **Get TextList** command allows you to find more than one object type at a time. Here is an example that finds and processes markers, cross-references, and variables at the same time. In most cases, you will not do this, but you can if necessary.

Code Listing 8-26

```
// Set a variable for the active document and the main flow.
Set vCurrentDoc = ActiveDoc;
Set vMainFlowInDoc = vCurrentDoc.MainFlowInDoc;
```

```
// Loop through the paragraphs in the main text flow.
Set vPgf = vMainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  // Get a list of three text item types in the paragraph.
  Get TextList InObject(vPgf) MarkerAnchor XRefBegin VarBegin
    NewVar(vTextList);
  // Use a loop to process the items in paragraph order.
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    // Get the text item from the text list.
    Get Member Number(n) From(vTextList) NewVar(vTextItem);
    // Test to see what kind it is.
    If vTextItem.TextType = 'MarkerAnchor'
      Set vMarker = vTextItem.TextData;
      // Do something with the marker.
    EndIf
    If vTextItem.TextType = 'XRefBegin'
      Set vXRef = vTextItem.TextData;
      // Do something with the cross-reference.
    EndIf
    If vTextItem.TextType = 'VarBegin'
      Set vVar = vTextItem.TextData;
      // Do something with the variable.
    EndIf
  EndLoop
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

9 Script Interfaces

An important part of a script is its *interface*. The interface determines how a user interacts with the script. We are using the word interface in its broadest sense to include how the script is run and how information is passed back and forth between the user and the script.

Running Scripts

Standard Scripts

There are three simple ways to run a standard script. One, is to use **FrameScript** > **Run** and choose a script file that has been saved to disk. The second way is to run a script that is open in the Script Window by pressing the Run button, or by choosing **Run** > **Script** (Control+R). This is convenient when you are developing and testing small scripts.

The third method is to install the script so that it appears in the **FrameScript** > **Scripts** submenu. To install a script, you can use **FrameScript** > **Install**, or use an Initial Script to install the script automatically when FrameMaker starts. See "Installing Scripts Automatically" on page 1-15 for more information. Installed scripts can be run with a keyboard shortcut if one has been assigned.

Event Scripts

Event scripts must be installed before they are run. After they are installed, they are run in response to a particular "event." An event can be one of the following.

- The user chooses a menu command or presses a keyboard shortcut.
- The user clicks on a command in a FrameMaker command palette.
- A specific FrameMaker event occurs, such as when a file opens, closes, or is saved.

The following sections will discuss each method in detail. To help you understand the concepts, we will work through an example script. Our example script will allow you to lock one or more selected graphics in an anchored frame so that they can't be accidently selected. Another command will unlock all of the graphics in the selected anchored frame. To set up the example, follow these steps.

1. Make a script called **ResetGraphicScript.fsl** and add this code to it. Make sure you save the script after adding the code.

Code Listing 9-1

```
// Resets the example event script.

// Find the folder where this script is located.
Find String(DIRSEP) InString(ThisScript) Backward ReturnPos(vPos);
Get String FromString(ThisScript) EndPos(vPos) NewVar(vScriptPath);

// Uninstall the event script.
Uninstall Script Name('LockUnlockGraphics');
```

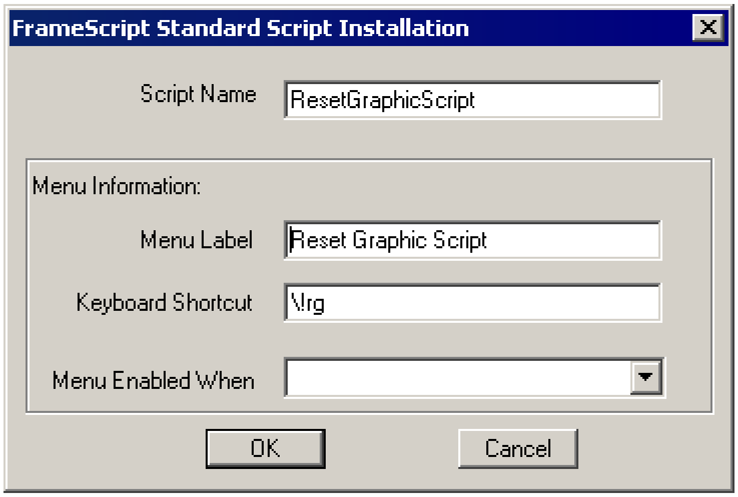
```
// Install (reinstall) the event script.
Install Script File(vScriptPath+'LockUnlockGraphics.fsl')
  Name('LockUnlockGraphics');
```

2. Make a script called **LockUnlockGraphics.fsl** and add this code to it. Save the script in the same folder as the ResetGraphicScript.fsl script.

Code Listing 9-2

```
Event Initialize
//
Write Console ThisScript + ' is installed.';
//
EndEvent

Event Terminate
//
Write Console ThisScript + ' is uninstalled.';
//
EndEvent
```

3. Choose **FrameScript** > **Install** to install the ResetGraphicScript.fsl script. Assign a shortcut key as indicated in the screenshot below. This will allow you to run the script by pressing Esc r g.



Script Name: ResetGraphicScript

Menu Label: Reset Graphic Script

Keyboard Shortcut: \!rg

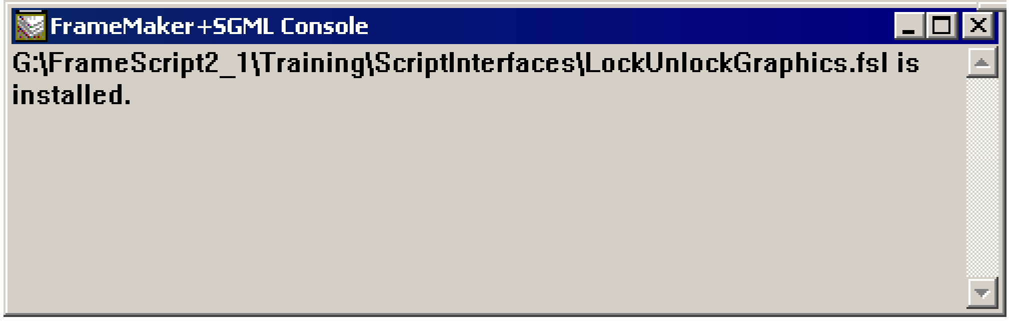
Menu Enabled When:

Using Commands with Event Scripts

Remember that an event script resides in memory after it is installed. It is removed from memory when the script is uninstalled or when you quit FrameMaker. When you develop and test an event script, it is helpful to have a “Reset” script that will quickly uninstall and reinstall it after you make changes to the code. That is the purpose of our ResetGraphicScript.fsl script. When you make changes to the other script—LockUnlockGraphics.fsl—you run the ResetGraphicScript.fsl script in order for the changes to be loaded into memory.

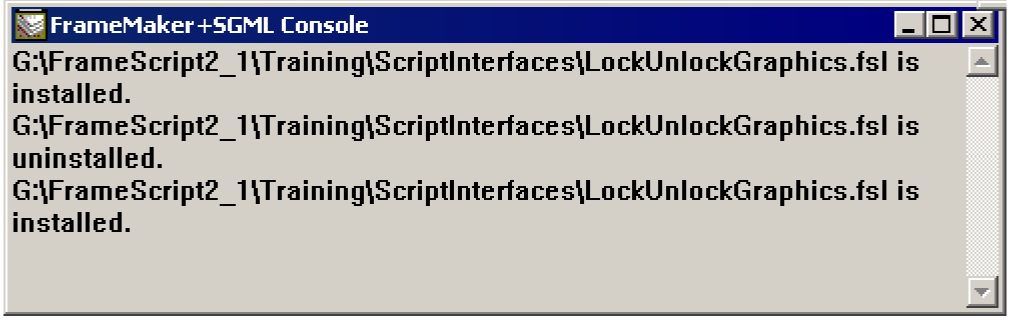
Initialize and Terminate Events

You should also remember that an event script does not have a body; all of its code is contained inside of one or more events and subroutines. There are two special kinds of events that can be present in an event script. One is called an **Initialize** event. An Initialize event runs automatically when the event script is installed. A **Terminate** event runs automatically when the event script is uninstalled. Run ResetGraphicsScript.fsl (Esc r g) and take a look at the Console window.



```
G:\FrameScript2_1\Training\ScriptInterfaces\LockUnlockGraphics.fsl is
installed.
```

You can see that the **Initialize** event ran and the **Write Console** command wrote a message to the Console. Run ResetGraphicScript.fsl and look at the Console window.



```
G:\FrameScript2_1\Training\ScriptInterfaces\LockUnlockGraphics.fsl is
installed.
G:\FrameScript2_1\Training\ScriptInterfaces\LockUnlockGraphics.fsl is
uninstalled.
G:\FrameScript2_1\Training\ScriptInterfaces\LockUnlockGraphics.fsl is
installed.
```

When the **Uninstall** command was run in ResetGraphicScript.fsl, it triggered the **Terminate** event in LockUnlockGraphics.fsl. When the script was reinstalled, the **Initialize** event was run.

Because the **Initialize** and **Terminate** events run automatically, you can use them to create and install custom menus and commands. If you uninstall your event script, the **Terminate** event can “cleanup” by uninstalling custom menus and commands. To illustrate this, edit the code in LockUnlockGraphics.fsl as shown below.

Code Listing 9-3

```
Event Initialize
//
// Add GraphicsSpecial menu.
New Menu Label('GraphicsSpecial') NewVar(vGraphicsSpecialMenu)
  AddTo('!MakerMainMenu');
//
EndEvent

Event Terminate
//
// Remove the GraphicsSpecial menu.
Remove MenuObject(vGraphicsSpecialMenu) From('!MakerMainMenu');
//
EndEvent
```

Run ResetGraphicScript.fsl and look at the FrameMaker menu. You see a GraphicsSpecial menu to the left of the Window menu. At this point, there are no commands on the menu, so it will not drop down when you click on it.

It is important that you use the **Terminate** event to remove whatever menus and commands that you add in the **Initialize** event. If you don't, new menus and commands will "stack up" everytime you install the event script. If you want to see this, comment out the **Terminate** event and run ResetGraphicScript.fsl a few times. Each time you run it, another GraphicSpecial menu will be added to the FrameMaker main menu. Note that you will have to exit FrameMaker in order to clear the GraphicSpecial menus.

Adding Custom Menus

As you saw in the previous section, you use the **New Menu** command to create a custom menu. You can use the **AddTo** parameter to add the new menu to an existing menubar or menu. There are four FrameMaker menubars that you can add your menus to: **!MakerMainMenu**, **!ViewOnlyMenu**, **!QuickMakerMainMenu**, and **!BookMainMenu**. When you can add a menu to an existing menu—for example, the Graphics menu—the new menu will be added as a submenu to the existing menu. Submenus are sometimes known as "pull-right" menus. For a list of FrameMaker menu names, see the Menus.cfg file in the fminit/maker (or fminit/fmsgml) folder inside your FrameMaker folder.

You can also create a menu and add it to a menu with two separate commands. This is useful if you want to add your new menu to more than one existing menu. Here is an example.

Code Listing 9-4

```
// Create a Callouts menu.
New Menu Label('Callouts') NewVar(vCalloutsMenu);

// Add the menu to two existing menus.
Add MenuObject(vCalloutsMenu) To('!MakerMainMenu');
Add MenuObject(vCalloutsMenu) To('!QuickMakerMainMenu');
```

Changing Menu Positions

When you add a new menu to an existing one, it is always added to the bottom of the existing menu. When adding a menu to a menubar, it appears to the left of the Window menu. You can move the new menu to a new location, by getting the menu or command object that is adjacent to the location where you want the menu. Then, you set the new menu's **NextMenuItemInMenu** or **PrevMenuItemInMenu** property to the adjacent menu or command. Here is an example that will move our GraphicSpecial menu directly to the right of the Graphics menu.

Code Listing 9-5

```
Event Initialize
//
// Add GraphicsSpecial menu.
New Menu Label('GraphicsSpecial') NewVar(vGraphicsSpecialMenu)
  AddTo('!MakerMainMenu');
```

```
// Get the Graphics menu object.
Get Object Type(Menu) Name('GraphicsMenu') NewVar(vGraphicsMenu);
// Get the object of the FrameMaker main menu.
Get Object Type(Menu) Name('!MakerMainMenu')
  NewVar(vMakerMainMenu);
Set vGraphicsMenu.Menu = vMakerMainMenu;

// Move the new menu to the right of the Graphics menu.
Set vGraphicsSpecialMenu.Menu = vMakerMainMenu;
Set vGraphicsSpecialMenu.PrevMenuItemInMenu = vGraphicsMenu;
//
EndEvent
```

Notice that in order to move the menus, you have to get the menubar object that currently contains the menus and assign it to the menu variable. That is the purpose of these commands.

```
Set vGraphicsMenu.Menu = vMakerMainMenu;
Set vGraphicsSpecial.Menu = vMakerMainMenu;
```

If you don't assign the **Menu** property to the menus, you cannot rearrange them. If you are adding a menu to an existing menu, you have to get an adjacent **Command** object in the existing menu in order to move the new menu. Temporarily modify the code in LockUnlockGraphics.fsl to add the GraphicsSpecial menu to the FrameMaker Graphics menu. The new menu will be added directly above the Gravity command.

Code Listing 9-6

```
Event Initialize
//
// Add GraphicsSpecial menu to the Graphics menu.
New Menu Label('GraphicsSpecial') NewVar(vGraphicsSpecialMenu)
  AddTo('GraphicsMenu');

// Get the Graphics menu object.
Get Object Type(Menu) Name('GraphicsMenu') NewVar(vGraphicsMenu);

// Get the Gravity command object.
Get Object Type(Command) Name('GraphicsGravity')
  NewVar(vGravityCmd);

// Set the menu object of the GraphicSpecial menu and Gravity
// command.
Set vGraphicsSpecialMenu.Menu = vGraphicsMenu;
Set vGravityCmd.Menu = vGraphicsMenu;

// Move the new menu above the Gravity command.
Set vGraphicsSpecialMenu.NextMenuItemInMenu = vGravityCmd;
//
EndEvent
```

IMPORTANT: If you try this code, quit FrameMaker, restore the original code, and restart FrameMaker. Reinstall the ResetGraphicScript.fsl script.

Adding Custom Commands

The syntax for adding custom commands is very similar to adding custom menus. You use the **New Command** command with the **AddTo** parameter to create a command and add it to a menu in one step. Or, you can add an existing command to a menu by using the **Add CommandObject** command. Here is the **Initialize** event of LockUnlockGraphics.fsl with two commands defined and added to the GraphicsSpecial menu.

Code Listing 9-7

```
Event Initialize
//
// Add GraphicsSpecial menu.
New Menu Label('GraphicsSpecial') NewVar(vGraphicsSpecialMenu)
  AddTo('!MakerMainMenu');

// Get the Graphics menu object.
Get Object Type(Menu) Name('GraphicsMenu') NewVar(vGraphicsMenu);
Get Object Type(Menu) Name('!MakerMainMenu')
NewVar(vMakerMainMenu);
Set vGraphicsMenu.Menu = vMakerMainMenu;

// Move the new menu to the right of the Graphics menu.
Set vGraphicsSpecialMenu.Menu = vMakerMainMenu;
Set vGraphicsSpecialMenu.PrevMenuItemInMenu = vGraphicsMenu;

// Make new commands and add them to the new menu.
New Command Label('Lock Selected Graphics') NewVar(vLockCmd)
  EventProc(LockGraphics) AddTo(vGraphicsSpecialMenu);
New Command Label('Unlock All Graphics in Frame') NewVar(vUnlockCmd)
  EventProc(UnlockGraphics) AddTo(vGraphicsSpecialMenu);
//
EndEvent
```

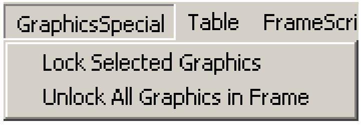
When you run ResetGraphicScript.fsl, you will see that the commands aren't added to the menu. The reason is the **EventProc** parameter. This parameter specifies an event procedure that will run when you choose the command. If the event procedure specified for a command doesn't exist in the script, the command won't be installed. To fix the code, add the two events below to the script. Although they don't do anything yet, they will allow the commands to be installed.

Code Listing 9-8

```
Event LockGraphics
//
//
EndEvent

Event UnlockGraphics
//
//
EndEvent
```

The events in an event script do not have to appear in any particular order in the script. I usually put the Initialize and Terminate events at the top of the script, followed by the events called by the commands. After that, I include any subroutines that are used in the script. You can use whatever order that is comfortable for you.

After you run ResetGraphicScript.fsl, you should see the commands in the menu.



To make the commands easier to use, you should add shortcut keys by using the **Shortcut** parameter. Change the **New Command** commands to add a shortcut to each of the commands.

Code Listing 9-9

```
// Make new commands and add them to the new menu.
New Command Label('Lock Selected Graphics') NewVar(vLockCmd)
  EventProc(LockGraphics) AddTo(vGraphicsSpecialMenu)
  Shortcut('\!lg');
New Command Label('Unlock All Graphics in Frame') NewVar(vUnlockCmd)
  EventProc(UnlockGraphics) AddTo(vGraphicsSpecialMenu)
  Shortcut('\!ug');
```

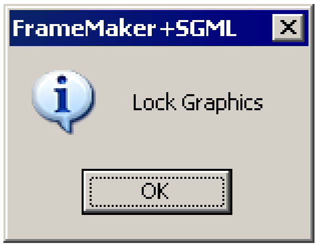
The \! represents the Escape key that is used in FrameMaker Escape key sequences. You use other modifier keys such as the Control key (represented by the ^ character), but be careful that you don't specify a shortcut that you normally use for a FrameMaker command. For example, if you specify Control+L ('^l') for Lock Selected Graphics, it will work for your command, but it will not refresh the screen like it normally does in the FrameMaker interface.

Add some temporary commands to the command events so you can try the shortcuts.

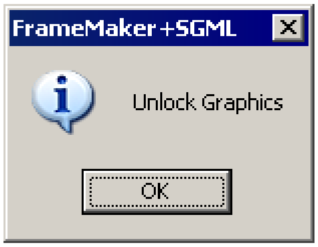
Code Listing 9-10

```
Event LockGraphics
//
Display 'Lock Graphics';
//
EndEvent

Event UnlockGraphics
//
Display 'Unlock Graphics';
//
EndEvent
```

When you press one of the shortcut keys or choose a menu item, you should see the appropriate message box.


FrameMaker+SGML

Lock Graphics

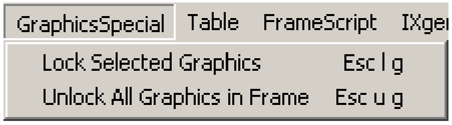

FrameMaker+SGML

Unlock Graphics

You can use the **Label** parameter to display the shortcut keys with the commands. This helps your users to see and learn the shortcut key each time they use the command. Use the **CHARTAB** variable to separate the command name from the shortcut key on the command.

Code Listing 9-11

```
// Make new commands and add them to the new menu.
New Command Label('Lock Selected Graphics'+CHARTAB+'Esc l g')
  NewVar(vLockCmd) EventProc(LockGraphics)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!lg');
New Command Label('Unlock All Graphics in Frame'+CHARTAB+'Esc u g')
  NewVar(vUnlockCmd) EventProc(UnlockGraphics)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!ug');
```

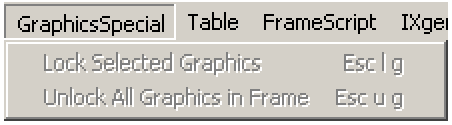

GraphicsSpecial
Table
FrameScript
IXgei
Lock Selected Graphics	Esc l g
Unlock All Graphics in Frame	Esc u g

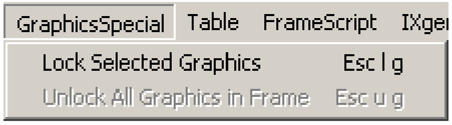
You can determine the "context" of your custom commands. The context of a command determines when the command is available in the menu. A command that is not available is "grayed out" on the menu. For this script, the Lock command should be available only when one or more graphics is selected inside of an anchored frame. We can accomplish this by setting the **Command** object's **EnableWhen** property to **EnableIsOrInFrame**. The Unlock command should only be available when an anchored frame is selected; the correct value is **EnableIsAFrame**. For a list of possible values for the **EnableWhen** property, see the *FrameScript Scriptwriter's Reference*. Here is the modified code.

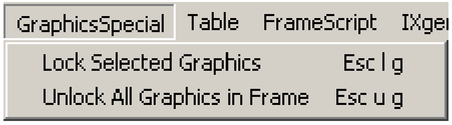
Code Listing 9-12

```
// Make new commands and add them to the new menu.
New Command Label('Lock Selected Graphics'+CHARTAB+'Esc l g')
  NewVar(vLockCmd) EventProc(LockGraphics)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!lg');
Set vLockCmd.EnabledWhen = EnableIsOrInFrame;
New Command Label('Unlock All Graphics in Frame'+CHARTAB+'Esc u g')
  NewVar(vUnlockCmd) EventProc(UnlockGraphics)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!ug');
Set vUnlockCmd.EnabledWhen = EnableIsAFrame;
```

Run ResetGraphicScript.fsl and you should see that both commands are grayed out. Create an anchored frame and add a graphic to the frame. Select the graphic and take a look at the GraphicsSpecial menu. You should see that the Lock Selected Graphics command is available, while the Unlock All Graphics in Frame command is still grayed out. Select the anchored frame and both commands will be available.


GraphicsSpecial
Table
FrameScript
IXgei
Lock Selected Graphics	Esc l g
Unlock All Graphics in Frame	Esc u g


GraphicsSpecial
Table
FrameScript
IXgei
Lock Selected Graphics	Esc l g
Unlock All Graphics in Frame	Esc u g



Adding Separators to Menus

In our example script, we have decided to add a command to proportionately scale the selected graphic to fit the anchored frame. We want to separate the new command from the others on the menu with a separator line. First off all, add the ScaleGraphic event to the script. This will ensure that the Scale Selected Graphic to Frame command will appear when we add it to the menu.

Code Listing 9-13

```
Event ScaleGraphic
//
Display 'Scale Graphic';
//
EndEvent
```

To add the separator, you use the **Add MenuSepObject** command. FrameMaker has five predefined separator objects, so you have to "get" one of them with the **Get Object** command before you add it a menu. Add this code to the end of the Initialize event.

Code Listing 9-14

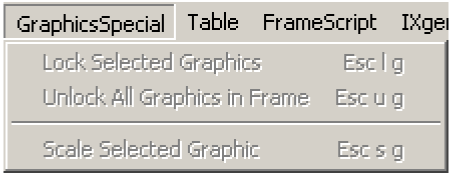
```
// Add a separator to the menu.
Get Object Type(MenuItemSeparator) Name('Separator1')
  NewVar(vMenuSep1);
Add MenuSepObject(vMenuSep1) To(vGraphicsSpecialMenu);
```

When you run ResetGraphicScript.fsl, you will not see the separator because it is at the bottom of the menu. When you add the new command, the separator will appear above it. Add the code for the new command at the end of the Initialize event.

Code Listing 9-15

```
// Add the Scale Graphics command.
New Command Label('Scale Selected Graphic'+CHARTAB+'Esc s g')
  NewVar(vScaleCmd) EventProc(ScaleGraphic)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!sg');
Set vScaleCmd.EnabledWhen = EnableIsOrInFrame;
```

Run ResetGraphicScript.fsl and take a look at the GraphicsSpecial menu. You should see the separator, followed by the new command.



You cannot add the same **Separator** object to a menu more than once. To add another separator to a menu, get the next separator object.

Code Listing 9-16

```
// Add a second separator to a menu.
Get Object Type(MenuItemSeparator) Name('Separator2')
  NewVar(vMenuSep2);
Add MenuSepObject(vMenuSep2) To(vMenu);
```

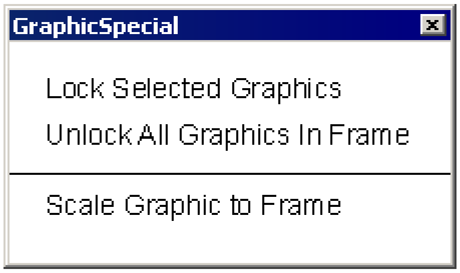
If you need more than five separators on a menu, you can make your own **Separator** objects using the **New MenuItemSeparator** command. This command has an optional **AddTo** parameter so you can create it and add it to a menu in one step.

Code Listing 9-17

```
// Make a new separator object and add it to a menu.
New MenuItemSeparator NewVar(vSep6) Name('Separator6')
  AddTo(vMenu);
```

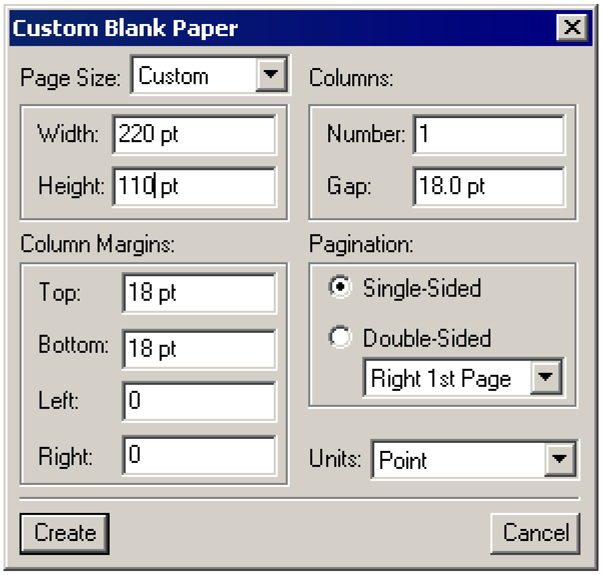
Controlling Scripts with FrameMaker Palettes

A FrameMaker palette is a special view-only document that floats above other document windows. FrameMaker's Equation Palette is an example of a palette. You can make your own palettes from ordinary FrameMaker documents and use them to control scripts. In this section, we will make a palette for executing the commands in our example script. Here is a screenshot of the finished palette.



Making the Palette Document

Choose **File** > **New** > **Document** and click the Custom button. Specify a page size and margins for your document, making it large enough to hold all of your command text or buttons. For our example palette, we are using the specifications below.


Custom Blank Paper

Page Size: Custom

Width: 220 pt

Height: 110 pt

Columns:

Number: 1

Gap: 18.0 pt

Column Margins:

Top: 18 pt

Bottom: 18 pt

Left: 0

Right: 0

Pagination:

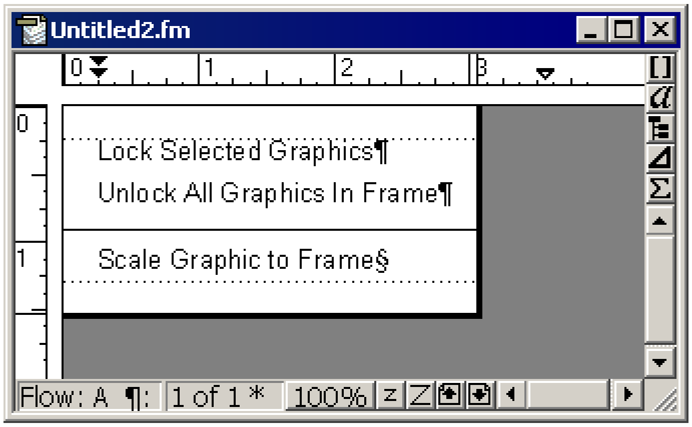
● Single-Sided

○ Double-Sided

Right 1st Page

Units: Point

If necessary, you can change the size of your document after you make it. Add the command text and any separator lines or graphics that you need. Because this is still a normal FrameMaker document, you can format it any way you want.


Untitled2.fm

Lock Selected Graphics¶

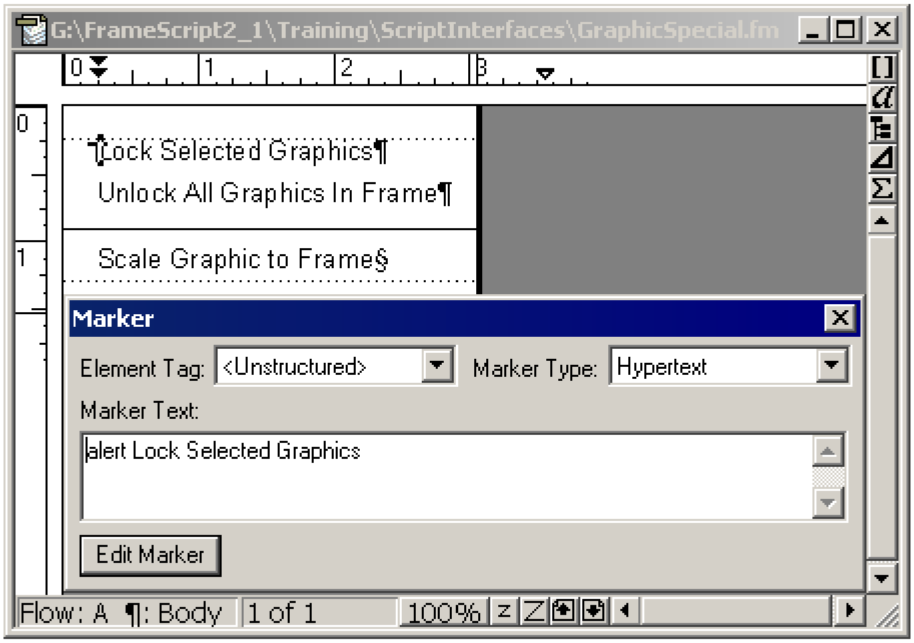
Unlock All Graphics In Frame¶

Scale Graphic to Frame§

Flow: A ¶: 1 of 1 * 100%

Save the document with a descriptive name; we are using GraphicsSpecial.fm so that it matches our menu name.

The next step is to add Hypertext markers to the command text. These markers will “trigger” the appropriate command in the script when the user clicks on it. For now, we will add simple “alert” commands to test the palette. Add a Hypertext marker at the beginning of each paragraph, using the paragraph text as the alert text.



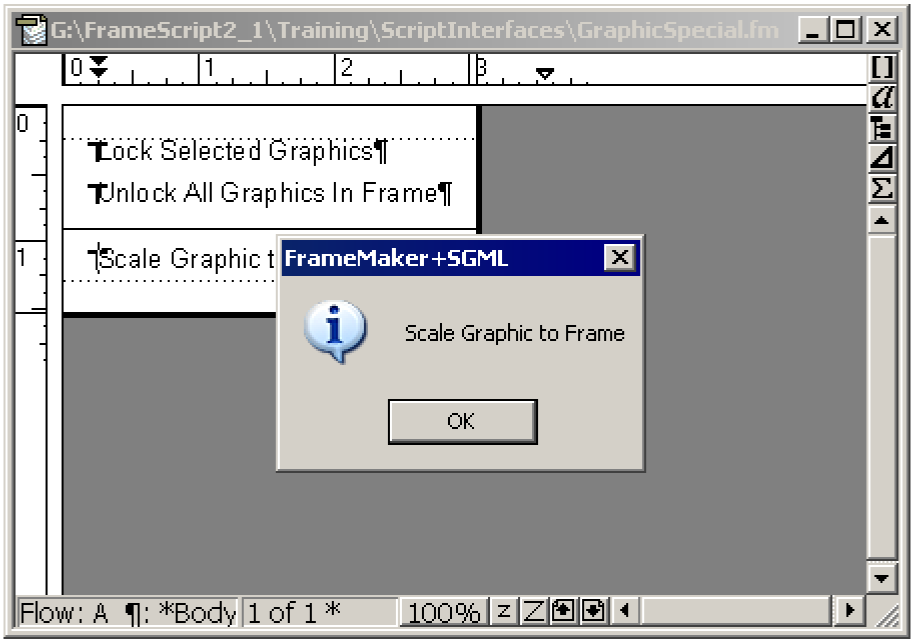
G:\FrameScript2_1\Training\ScriptInterfaces\GraphicSpecial.fm

Lock Selected Graphics¶
Unlock All Graphics In Frame¶
Scale Graphic to Frame§

Marker

Element Tag: <Unstructured>
Marker Type: Hypertext
Marker Text:
alert Lock Selected Graphics

To test the markers, Control+Alt+Click (Control+Option+Click on the Mac) on the text of each paragraph. You should see the correct alert text display.



G:\FrameScript2_1\Training\ScriptInterfaces\GraphicSpecial.fm

Lock Selected Graphics¶
Unlock All Graphics In Frame¶
Scale Graphic t

FrameMaker+SGML

Scale Graphic to Frame

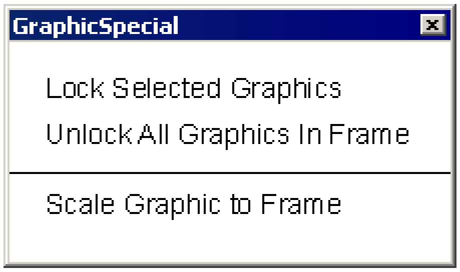
Converting the Document to a Palette

The next step is to make the document into a palette. Make sure that borders and text symbols are hidden and save the document. Choose **File** > **Save As** and save the document as MIF (Maker Interchange Format), using the .MIF extension.

Open the MIF file in a text editor and make the changes indicated in the table.

Find This	Replace With This
<DViewOnly No>	<DViewOnly Yes>
<DViewOnlyWinBorders Yes>	<DViewOnlyWinBorders No>
<DViewOnlyWinMenubar Yes>	<DViewOnlyWinMenubar No>
<DViewOnlyWinPopup Yes>	<DViewOnlyWinPopup No>
<DViewOnlyWinPalette No>	<DViewOnlyWinPalette Yes>

Save the document as text WITHOUT the file extension. Note that some text editors will automatically put an extension on the file if you don't supply one. To avoid this, put quotes around the file name in the Save As dialog box of your text editor ("GraphicsSpecial"). Now open the GraphicsSpecial text file with FrameMaker. It should open as a palette document.



Try clicking on each of the lines to display the alerts.

You can automate the process of converting the document to a palette by using the MakePalette.fsl script, which is included with the code listings for this chapter.

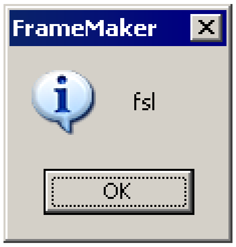
Passing Commands from the Palette to the Script

FrameMaker has a special Hypertext marker syntax that allows you to send commands from the palette to the script. The basic syntax is

```
message clientname scriptname parameter
```

The **clientname** argument is the registered name of the FrameMaker plugin, in this case FrameScript. FrameScript 2.1R3 for FrameMaker 7 has a **ClientName** variable that returns the registered name.

```
Display ClientName;
```



fsl

With Windows FrameMaker, you can also find out the registered name for FrameScript by looking in the maker.ini (or fmsgml.ini) file under the [API Clients] section.

```
[APIClients]
;------------------------ API Clients ------------------------
...
fsl=Standard,FrameScript,C:\FrameScript2_1\fsl60.dll
```

In the above case, the name is **fsl**. On Macintosh systems, the registered name is the name of the FrameScript plugin in the Modules folder, for example **fsl2_60mr**.

The **scriptname** argument is the name of the installed event script; in this case, **LockUnlockGraphics**. Finally, **parameter** is a unique string that we send for each of the commands on the palette. Here is the complete Hypertext marker text for each of the commands

```
message fsl LockUnlockGraphics LockGraphics

message fsl LockUnlockGraphics UnlockGraphics

message fsl LockUnlockGraphics ScaleGraphic
```

You may have noticed that we used the event name of each of the commands as the parameter value. This will make things easier later, as you will soon see. Close the palette and open the original palette document (GraphicSpecial.fm). Replace the current marker content with the appropriate line above. Save the document and run the MakePalette.fsl script to convert it to a palette.

Now we must prepare the LockUnlockGraphics.fsl script to receive the palette commands. The script must have a Message event to accept the Hypertext marker commands. Here is the shell for a Message Event; add this to your example script right after the Initialize event.

Code Listing 9-18

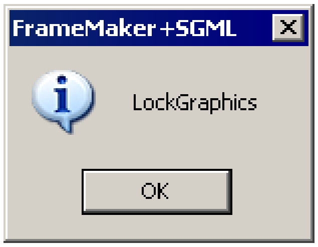
```
Event Message
//
//
EndEvent
```

The Message event will receive the parameter that is in the command's Hypertext marker. The parameter will be assigned to a special **Message** variable. To see this, add a line to the Message event.

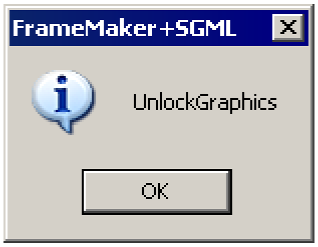
Code Listing 9-19

```
Event Message
//
Display Message;
//
EndEvent
```

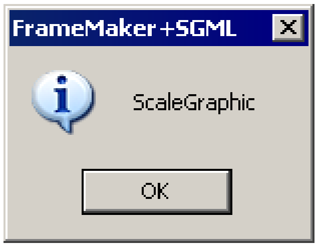
Save the LockUnlockGraphics.fsl script and run ResetGraphicScript.fsl to load the changes in memory. Click on the commands in the updated palette. You will see each of the parameters that are in the Hypertext markers in the palette.



LockGraphics



UnlockGraphics



ScaleGraphic

The easiest way to run the correct events is to assign a variable to **Message** and add the **Run** command in front of the variable. Edit the Message event in your script, save it, and reset it.

Code Listing 9-20

```
Event Message
//
Set vCommand = Message;
Run vCommand;
//
EndEvent
```

Now when you click on a command in the palette, it will run the correct event and display the same message that you get when you use the menu commands or shortcuts.

IMPORTANT: Do not attempt to use **Run Message;** directly. FrameMaker will crash. That is why we assign **Message** to a **vCommand** variable and use that with the Run command.

There is one problem with the Message event. If a Message is passed to the script doesn't have a corresponding event, you will get an error. You can add some error checking to the Message event to deal with incorrect parameters.

Code Listing 9-21

```
Event Message
//
Set vCommand = Message;
```

```
Loop While(1)
  If vCommand = 'LockGraphics'
    Run vCommand;
    LeaveLoop;
  EndIf
  If vCommand = 'UnlockGraphics'
    Run vCommand;
    LeaveLoop;
  EndIf
  If vCommand = 'ScaleGraphic'
    Run vCommand;
    LeaveLoop;
  EndIf
  MsgBox 'Command was not found.    ' Mode(Warn);
  LeaveLoop;
EndLoop
//
EndEvent
```

Adding a Way to Open the Palette

The palette document can be opened like any other FrameMaker document by using **File > Open**. To make it easier, we will add an Open Palette command to the GraphicsSpecial menu. We will add a separator to the menu before the Open Palette command. Before adding the code to the Initialize event, add the OpenPalette event to the script.

Code Listing 9-22

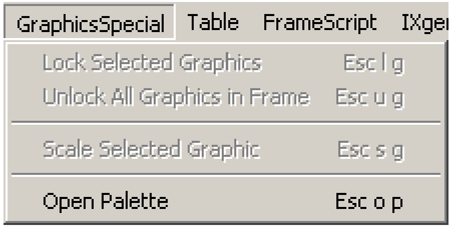
```
Event OpenPalette
//
//
EndEvent
```

Here is the code to add the Open Palette command. Put this code inside of the Initialize event at the end of the event.

Code Listing 9-23

```
// Add a separator to the menu.
Get Object Type(MenuItemSeparator) Name('Separator2')
NewVar(vMenuSep2);
Add MenuSepObject(vMenuSep2) To(vGraphicsSpecialMenu);
// Add the Open Palette command.
New Command Label('Open Palette'+CHARTAB+'Esc o p')
  NewVar(vOpenPaletteCmd) EventProc(OpenPalette)
  AddTo(vGraphicsSpecialMenu) Shortcut('\!op');
```

Run ResetGraphicScript.fsl and you should see the new separator and command on the menu.


GraphicsSpecial
Table
FrameScript
IXgei
Lock Selected Graphics Esc l g
Unlock All Graphics in Frame Esc u g
Scale Selected Graphic Esc s g
Open Palette Esc o p

Now we need the code for the OpenPalette event. We can use the **Open Document** command to open the palette. We use the **File** parameter to supply a path to the GraphicSpecial palette document. Rather than hard-code the path, we will assume that the palette is in the same folder as the LockUnlockCallouts.fsl script. You should instruct your users to install everything in the same folder.

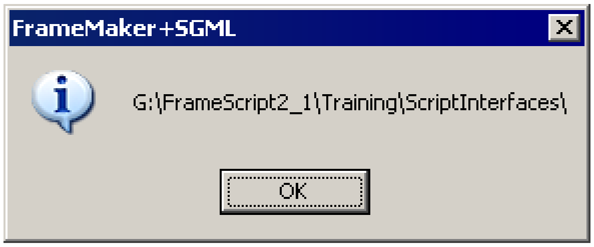
To determine what folder the script is in when it runs, you can use the special **ThisScript** variable. **ThisScript** returns the full path of the currently running script. You can use the **Find String** and **Get String** commands to drop the script name off of the full path.

Code Listing 9-24

```
Event OpenPalette
//
// Find the path of this script.
Find String(DIRSEP) InString(ThisScript) Backward ReturnPos(vPos);
Get String FromString(ThisScript) EndPos(vPos) NewVar(vPath);

// Display the path.
Display vPath;
//
EndEvent
```

We are finding the **DIRSEP** (directory separator) variable in the **ThisScript** variable, using the **Backward** option. **DIRSEP** represents a backslash (\) on the PC and a colon (:) on the Macintosh. The **Backward** option starts with the end of the string so it finds the last folder separator in the path.

The **Get String** command extracts everything in **ThisScript** up to and including the last folder separator, leaving us with the path we need to open the palette. Try the script and it will display the path.


FrameMaker+SGML

G:\FrameScript2_1\Training\ScriptInterfaces\

Now we can add the **Open Document** command to the OpenPalette event. Add this to the script and give it a try.

Code Listing 9-25

```
Event OpenPalette
//
// Find the path of this script.
Find String(DIRSEP) InString(ThisScript) Backward ReturnPos(vPos);
Get String FromString(ThisScript) EndPos(vPos) NewVar(vPath);

// Open the palette.
Open Document File(vPath+'GraphicSpecial') NewVar(vPaletteDoc);
//
EndEvent
```

If you uninstall the script, the palette will stay open and the palette commands will no longer work. Let's add some code to close the palette when the script is uninstalled. Add this code to the end of the Terminate event.

Code Listing 9-26

```
// Close the palette document.
Close Document DocObject(vPaletteDoc) IgnoreMods;
```

The only thing missing from our script is the code for the LockGraphics, UnlockGraphics, and ScaleGraphic events. This code is provided with the code listings for this chapter.

Responding to FrameMaker Events

FrameMaker has a series of events that you can use to trigger scripts. Here is a list of some of the events and when they occur.

Event Name	Event Description
`NoteBodyPageAdded`	Run after a body page is added to a document.
`NoteBodyPageDeleted`	Run after a body page is deleted from a document.
`NotePostFunction`	Run after a command is run or after text is typed.
`NotePreFunction`	Run before a command is run or before text is typed.
`NotePostGenerate`	Run after a document or book is generated.
`NotePreGenerate`	Run before a document or book is generated.
`NotePostPrint`	Run after a document or book is printed.
`NotePrePrint`	Run before a document or book is printed.
`NotePostQuitDoc`	Run after a document is closed.
`NotePreQuitDoc`	Run before a document is closed.
`NotePostSaveDoc`	Run after a document is saved.
`NotePreSaveDoc`	Run before a document is saved.

There are over 60 FrameMaker events; see the *FrameScript Scriptwriter's Reference* for a complete list.

Here is a simple example: This script opens the Paragraph catalog, Character catalog, and the Tool palette every time a document is opened.

Code Listing 9-27

```
Event PostOpenDoc
//
// Execute f-codes to open the catalogs and palette.
Execute Fc KbdPgfWin;       // Paragraph catalog
Execute Fc KbdFontWin;      // Character catalog
Execute Fc KbdSmallToolWin; // Tool palette
//
EndEvent
```

Save this script to a file and install it. Open a document to see what happens.

FrameMaker events return the following parameters: **FrameDoc** is a document or book object; **Filename** is the path of the document or book; and **IParm** can be an object or the f-code of a function that has occurred.

Here is a script that illustrates how to use the **FrameDoc** parameter to display the last page of each document that is opened. This script would be useful if you needed to check the last page of each document.

Code Listing 9-28

```
Event NotePostOpenDoc
//
// If the open document is the active document, display the last
// body page.
If FrameDoc = ActiveDoc
  Set FrameDoc.CurrentPage = FrameDoc.LastBodyPageInDoc;
EndIf
//
EndEvent
```

Here is a script that uses the **Filename** parameter. When you save a FrameMaker document, this script automatically saves a copy as MIF in the same folder.

Code Listing 9-29

```
Event NotePostSaveDoc
//
// Calculate the MIF filename.
Find String('.fm') InString(FileName) Backward NoCase
  ReturnStatus(vFound) ReturnPos(vPos);
If vFound
  Get String FromString(FileName) EndPos(vPos-1)
    NewVar(vNameNoExt);
  Set vMifName = vNameNoExt+'.mif';
Else
  Set vMifName = FileName+'.mif';
EndIf
// Save the document as MIF.
Save Document DocObject(FrameDoc) File(vMifName)
  FileType(SaveFmtInterchange);
//
EndEvent
```

You can use the **Return** command with the **Cancel** option to cancel an event. For example, suppose you have to open a series of documents that have missing fonts.

You can use a **NotePreOpenDoc** event to cancel the opening of a document, and then reopen it with an option to bypass the missing fonts message.

Code Listing 9-30

```
Event NotePreOpenDoc
//
// Cancel the opening of the document.
Return Cancel;
// Reopen the document, suppressing the missing fonts dialog.
Open Document File(Filename) FontNotFoundInDoc(OK);
//
EndEvent
```

You can use the **NotePreFunction** and **NotePostFunction** events to do something before or after a particular FrameMaker command executes. To do this, you must know the **Iparm** of the function. To determine the **Iparm**, you can temporarily install the script below, run the function, and record the **Iparm** number.

Code Listing 9-31

```
Event NotePreFunction
//
MsgBox 'Pre '+ IParm;
//
EndEvent

Event NotePostFunction
//
MsgBox 'Post '+ IParm;
//
EndEvent
```

Once you determine the **Iparm** number, you can use it in an event script. Here is a script that does not allow the user to open the paragraph or character designers.

Code Listing 9-32

```
Event NotePreFunction
//
// 817 = open paragraph designer; 816 = open character designer.
If (Iparm = 817) or (Iparm = 816)
  Return Cancel; // Don’t open the designers.
EndIf
//
EndEvent
```

Running Standard Scripts from Event Scripts

You can run a standard script from an event script. For example, you can run a standard script from a palette command or in response to a FrameMaker event. You use the **New ScriptVar** command to make a variable for the standard script. The example below shows a Message event that calls a standard script.

Code Listing 9-33

```
Event Message
//
Set vCommand = Message;
// Test for the command.
If vCommand = 'CountPagesInBook'
  // Get the path of this script.
  Find String(DIRSEP) InString(ThisScript) Backward ReturnPos(vPos);
  Get String FromString(ThisScript) EndPos(vPos)
    NewVar(vScriptPath);
  // Set a variable for the standard script.
  New ScriptVar File(vScriptPath+'CountPagesInBook.fsl')
    NewVar(vScriptVar);
  // Run the standard script.
  Run vScriptVar;
EndIf
//
EndEvent
```

Using Dialog Boxes

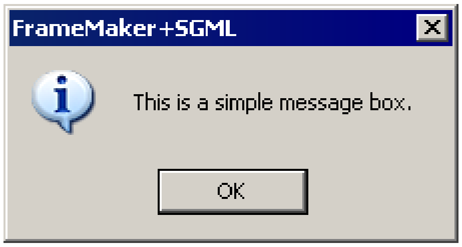
FrameScript has several dialog and message box commands to get and display information.

MsgBox

The **MsgBox** command displays a string message in a simple dialog. It has various parameters that add and change the appearance of buttons on the dialog. Here are some commented examples with corresponding screenshots.

Code Listing 9-34

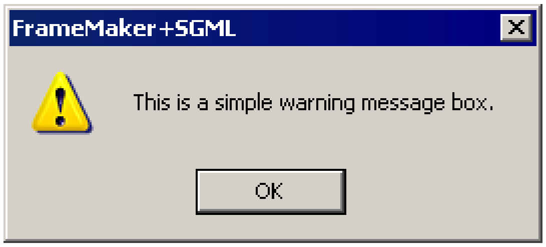
```
// Message only with Note icon.
MsgBox 'This is a simple message box.    ';
```



This is a simple message box.

Code Listing 9-35

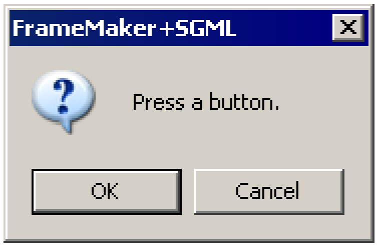
```
// Message only with a Warning icon.
MsgBox 'This is a simple warning message box.    ' Mode(Warn);
```



This is a simple warning message box.

Code Listing 9-36

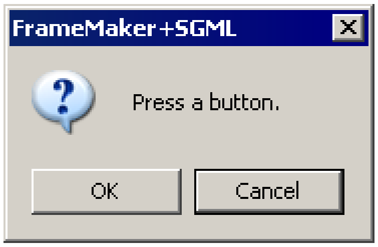
```
// Message with OK and Cancel buttons (OK button is default).
MsgBox Mode(OKCancel) 'Press a button.    ';
```



Press a button.

Code Listing 9-37

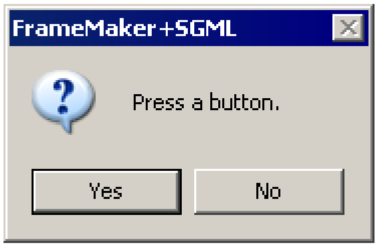
```
// Message with OK and Cancel buttons (Cancel button is default).
MsgBox Mode(CancelOK) 'Press a button.    ';
```



Press a button.

Code Listing 9-38

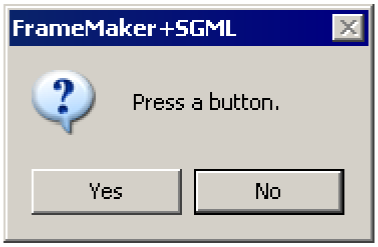
```
// Message with Yes and No buttons (Yes button is default).
MsgBox Mode(YesNo) 'Press a button.    ';
```



Press a button.

Code Listing 9-39

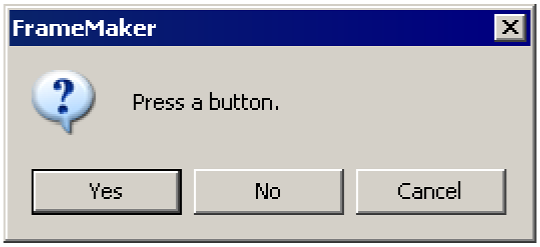
```
// Message with Yes and No buttons (No button is default).
MsgBox Mode(NoYes) 'Press a button.    ';
```



Press a button.

Code Listing 9-40

```
// Message with Yes, No, and Cancel buttons (Yes button is default).
// Works with FrameScript 2.1R3 for FrameMaker 7 only.
MsgBox Mode(YesNoCancel) 'Press a button.    ';
```



You can test which button the user pressed by including the **Button** parameter on the **MsgBox** command. You test the value of the variable that you assign to **Button**, by comparing it to the possible button constants.

Button Constant	How the Constant is Returned
`OKButton`	The OK button is clicked. The Enter key is pressed when the OK button is the default button.
`CancelButton`	The Cancel button is clicked. The Enter key is pressed when the Cancel button is the default button. The Escape key is pressed to dismiss the message box. The Close button in the message box title bar is clicked.
`YesButton`	The Yes button is clicked. The Enter key is pressed when the Yes button is the default button.
`NoButton`	The No button is clicked. The Enter key is pressed when the No button is the default button.

Here is an example that tests for the button constant. Try the different methods for OKing or cancelling the message box.

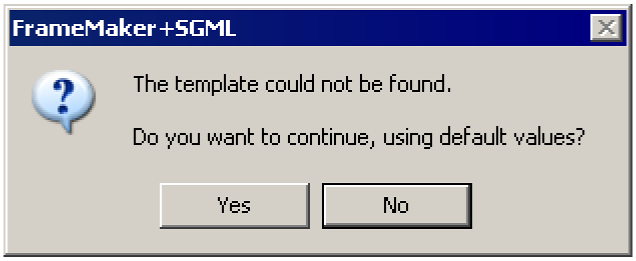
Code Listing 9-41

```
// Display a message box.
MsgBox 'Press OK to continue, Cancel to quit.    ' Mode(OKCancel)
  Button(vButton);
// Test which button was pressed and display a message.
If vButton = CancelButton
  MsgBox 'The user cancelled the message box.    ';
Else
  MsgBox 'The user OKed the message box.    ';
EndIf
```

You can display multiple lines in a message box by using the CHARLF (line feed) variable in your message box strings.

Code Listing 9-42

```
// Display a multiline message box.
MsgBox Mode(NoYes) Button(vButton)
  'The template could not be found.'+CHARLF+CHARLF+
  'Do you want to continue, using default values?    ';
```



The template could not be found.

Do you want to continue, using default values?

Display

The **Display** command is the same as a simple **MsgBox** command; it displays a string and an OK button with the note icon.

```
Display 'Hello world!    ';

MsgBox 'Hello world!    ';
```

You can use these two commands interchangeably, but a good practice is to reserve the **Display** command for adding debugging and troubleshooting messages to your scripts. That way, you can easily find and remove them after you solve the problems.

DialogBox

You can use the **DialogBox** command to retrieve information from the users of your scripts. There are three simple **DialogBox** types that prompt the user for particular kinds of data: strings, integers, and metrics (measurements).

Code Listing 9-43

```
// Prompt the user for a string.
DialogBox Type(String) Title('Please enter a string:')
  Button(vButton) NewVar(vString);
// If the user cancels the dialog, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf

// Display the string.
Display vString;
```

You can supply a default value for the dialog box by setting the value of the **NewVar** variable before the **DialogBox** command and using it with the **Init** parameter.

Code Listing 9-44

```
// Set a default value for vString.
Set vString = 'Default string';

// Prompt the user for a string.
DialogBox Type(String) Title('Please enter a string:')
  Button(vButton) Init(vString) NewVar(vString);
// If the user cancels the dialog, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf
```

In some cases, you may want to make sure that there is a value in the dialog box when the user clicks OK. You can do this with a simple loop.

Code Listing 9-45

```
// Set a default value for vString.
Set vString = 'Default string';

// Make an "endless" loop.
Loop While(1)
  // Prompt the user for a string.
  DialogBox Type(String) Title('Please enter a string:')
    Button(vButton) Init(vString) NewVar(vString);
  // If the user cancels the dialog, exit the script.
  If vButton = CancelButton
    LeaveSub;
  EndIf
  // Test for a value in the dialog box.
  If vString not= ''
    LeaveLoop; // If the value is not null, leave the loop.
  EndIf
EndLoop
```

If you delete the default string and click OK, the dialog box will reappear until you enter a value or click Cancel. If you don't need a default string, here is slightly shorter method.

Code Listing 9-46

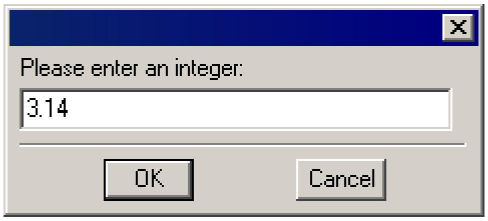
```
// Set a vString variable to a null string.
Set vString = '';

// Loop while the value is null.
Loop While(vString = '')
  // Prompt the user for a string.
  DialogBox Type(String) Title('Please enter a string:')
    Button(vButton) NewVar(vString);
  // If the user cancels the dialog, exit the script.
  If vButton = CancelButton
    LeaveSub;
  EndIf
EndLoop
```

Here is the **DialogBox** that prompts for an integer. Remember that an integer is a non-decimal value; if you enter a decimal number, only the whole number portion will be returned.

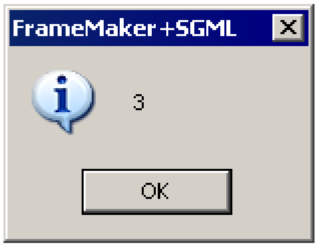
Code Listing 9-47

```
// Prompt the user for an integer.
DialogBox Type(Int) Title('Please enter an integer:')
  Button(vButton) NewVar(vInteger);
// If the user cancels the dialog, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf
```

```
// Display the integer.
Display vInteger;
```


Please enter an integer:

3.14

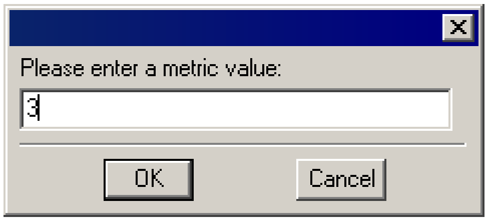

FrameMaker+SGML

3

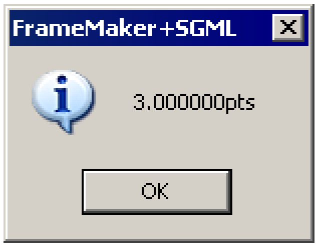
You can prompt for a metric value by using the **Metric** value on the **Type** parameter. A metric value is a FrameMaker measurement. You can specify the default units for the measurement by using the **Units** parameter, or you can override the units by using an abbreviation in the dialog box field. You can use the same abbreviations that can be used in FrameMaker dialog boxes. If you don’t supply a unit, the default unit is inches; however, when you display the measurement, it will be shown in points.

Code Listing 9-48

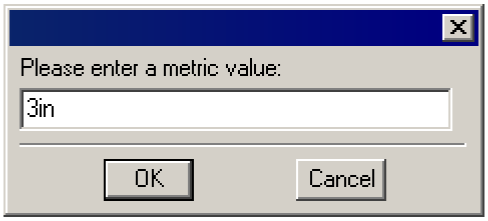
```
// Prompt the user for an measurement.
DialogBox Type(Metric) Title('Please enter a metric value:')
  Button(vButton) Units(Points) NewVar(vMetric);
// If the user cancels the dialog, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf

// Display the value.
Display vMetric;
```


Please enter a metric value:

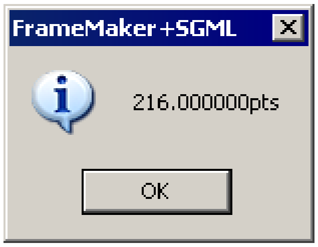
3


FrameMaker+SGML

3.000000pts


Please enter a metric value:

3in

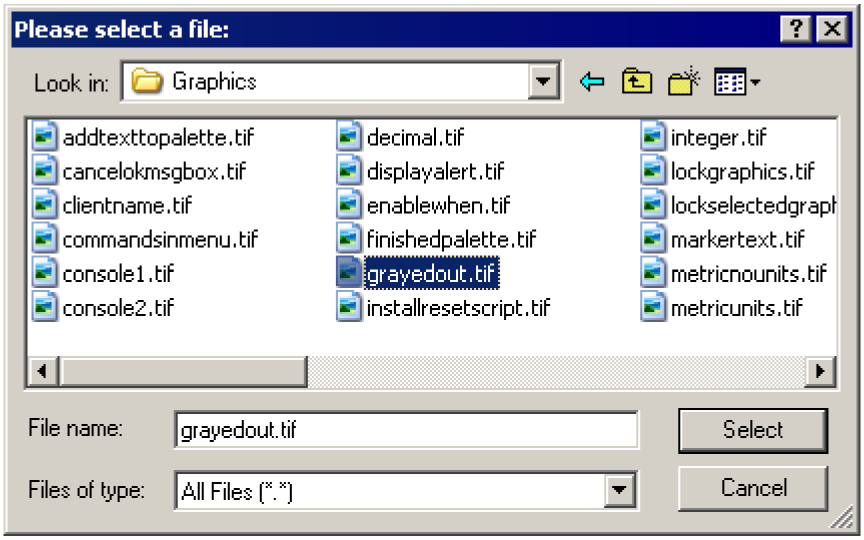

FrameMaker+SGML

216.000000pts

You can use dialog boxes to prompt your users for files or folders, using the **Type(ChooseFile)** option. The **Mode** parameter determines the appearance of the dialog box.

Code Listing 9-49

```
// Prompt the user to select a file.
DialogBox Type(ChooseFile) Mode(SelectFile) Button(vButton)
  Title('Please select a file:') NewVar(vFile);
Display vFile;
```



Please select a file:

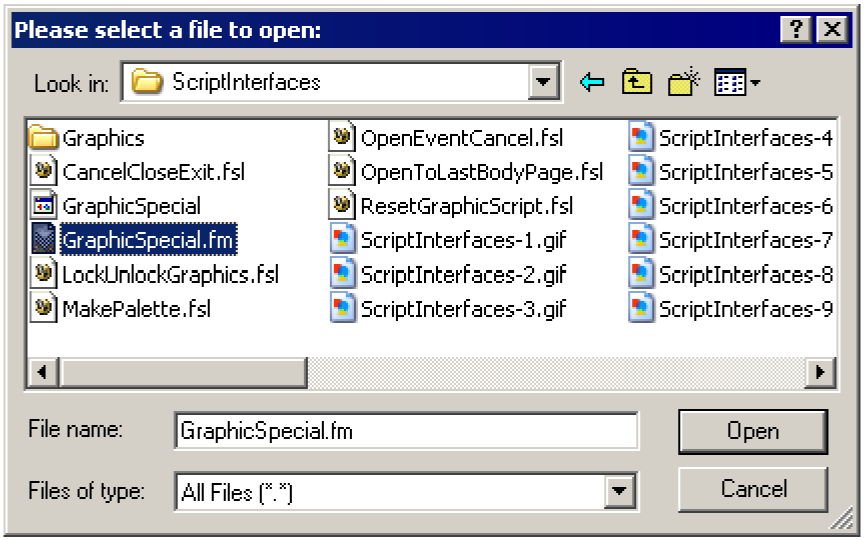
Look in: Graphics

File name: grayedout.tif

Files of type: All Files (*.*)

Code Listing 9-50

```
// Prompt the user to open a file.
DialogBox Type(ChooseFile) Mode(OpenFile) Button(vButton)
  Title('Please select a file to open:') NewVar(vFile);
Display vFile;
```



Please select a file to open:

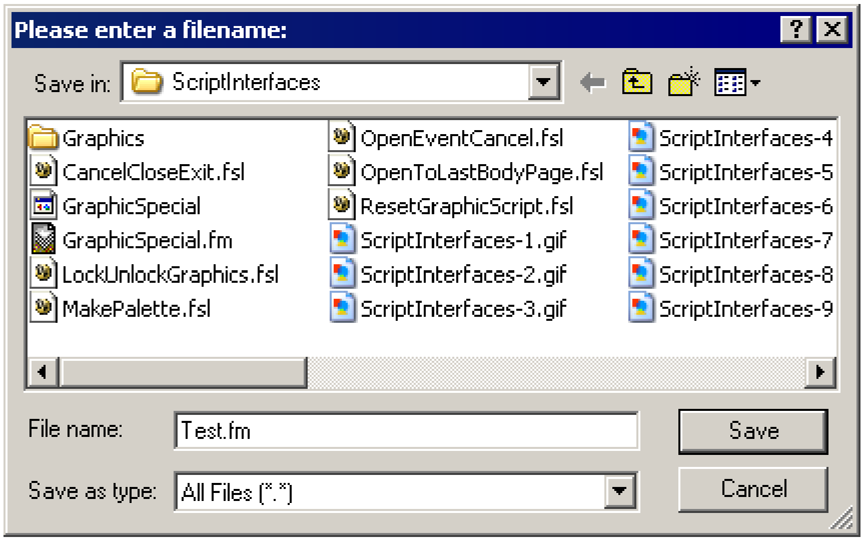
Look in: ScriptInterfaces

File name: GraphicSpecial.fm

Files of type: All Files (*.*)

Code Listing 9-51

```
// Prompt the user to save a file.
DialogBox Type(ChooseFile) Mode(SaveFile) Button(vButton)
  Title('Please enter a filename:') NewVar(vFile);
Display vFile;
```



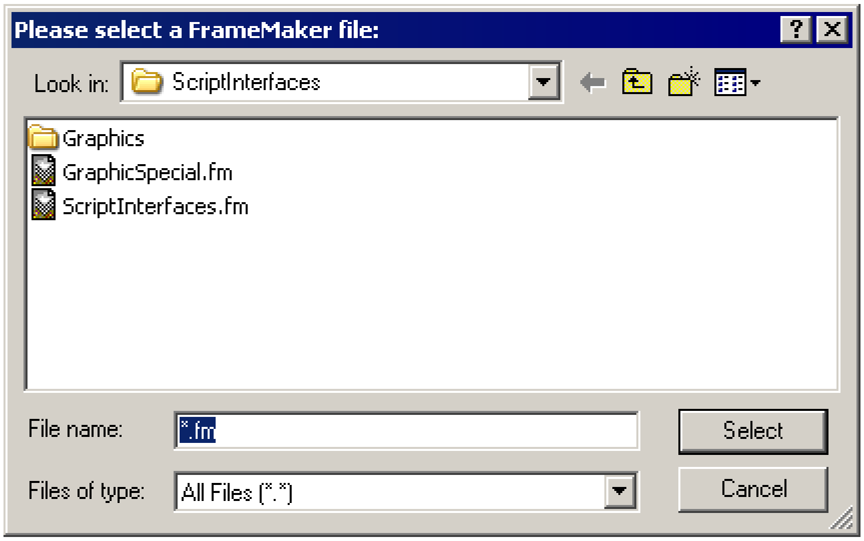
File name: Test.fm

Save as type: All Files (*.*)

The difference between the three modes is the button text. In addition, you cannot dismiss the **SaveFile** dialog box without entering a file name (unless you cancel the dialog box). You use the **Directory** parameter to specify a starting folder for the dialog box. The **Init** parameter will limit the types of files shown in the dialog box. Here is an example that only shows .fm files.

Code Listing 9-52

```
// Prompt the user for a file.
DialogBox Type(ChooseFile) Mode(SelectFile) Button(vButton)
  Title('Please select a FrameMaker file:') Init('*.fm')
  NewVar(vFile);
Display vFile;
```



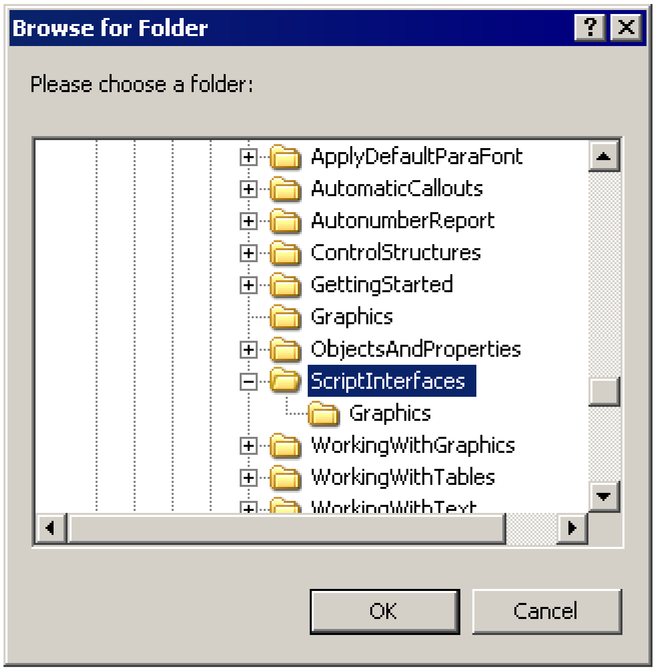
File name: *.fm

Files of type: All Files (*.*)

The **OpenDirectory** mode displays a folder browser dialog box.

Code Listing 9-53

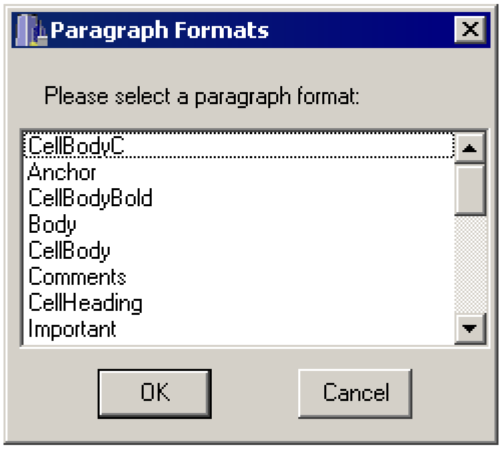
```
// Prompt the user for a folder.
DialogBox Type(ChooseFile) Mode(OpenDirectory) Button(vButton)
  Title('Please choose a folder:') NewVar(vFolder);
Display vFolder;
```



The **ScrollBox** dialog box is used to present a list of choices for the user to pick. You supply the list to display in the scroll box. In this example, we prompt the user with a list of the paragraph formats in the document.

Code Listing 9-54

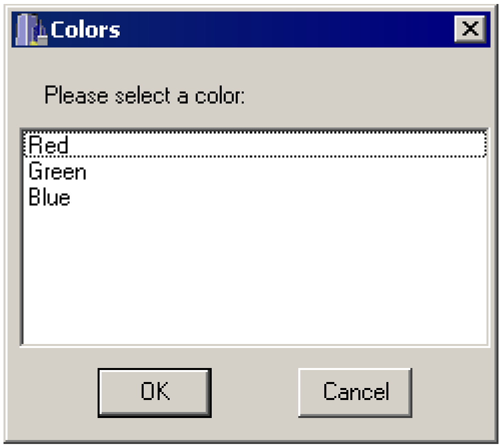
```
// Display a ScrollBox dialog of the paragraph formats in the doc.
DialogBox Type(ScrollBox) Title('Please select a paragraph format:')
  Caption('Paragraph Formats') List(DocPgfFmtNameList)
  Button(vButton) NewVar(vPgfFmtName);
Display vPgfFmtName;
```



You can use a predefined list like we did in the previous example (**DocPgfFmtNameList**), or you can make your own stringlist to use in the scroll list.

Code Listing 9-55

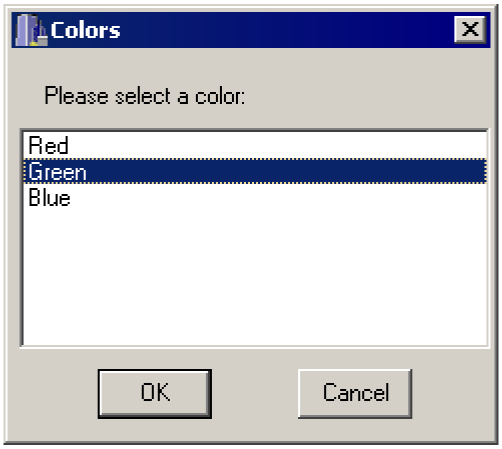
```
// Make a string list of colors.
New StringList NewVar(vColors);
Add Member('Red') To(vColors);
Add Member('Green') To(vColors);
Add Member('Blue') To(vColors);

// Display a ScrollBox dialog of the colors.
DialogBox Type(ScrollBox) Title('Please select a color:')
  Caption('Colors') List(vColors) Button(vButton) NewVar(vColor);
Display vColor;
```



You can supply a default selection by specifying its position in the **Init** parameter.

Code Listing 9-56

```
// Make a string list of colors.
New StringList NewVar(vColors);
Add Member('Red') To(vColors);
Add Member('Green') To(vColors);
Add Member('Blue') To(vColors);

// Display a ScrollBox dialog of the colors.
DialogBox Type(ScrollBox) Title('Please select a color:')
  Caption('Colors') List(vColors) Init(2) Button(vButton)
  NewVar(vColor);
Display vColor;
```



Please select a color:

The **NewVar** parameter returns the string that was chosen. If you use the **ReturnIndex** parameter, it will return the position number of the selected string. For instance, if the user chooses Green in the list above, the **ReturnIndex** variable will return **2**.

You can use the **ReturnIndex** parameter to make sure that the user makes a choice before the dialog box is dismissed.

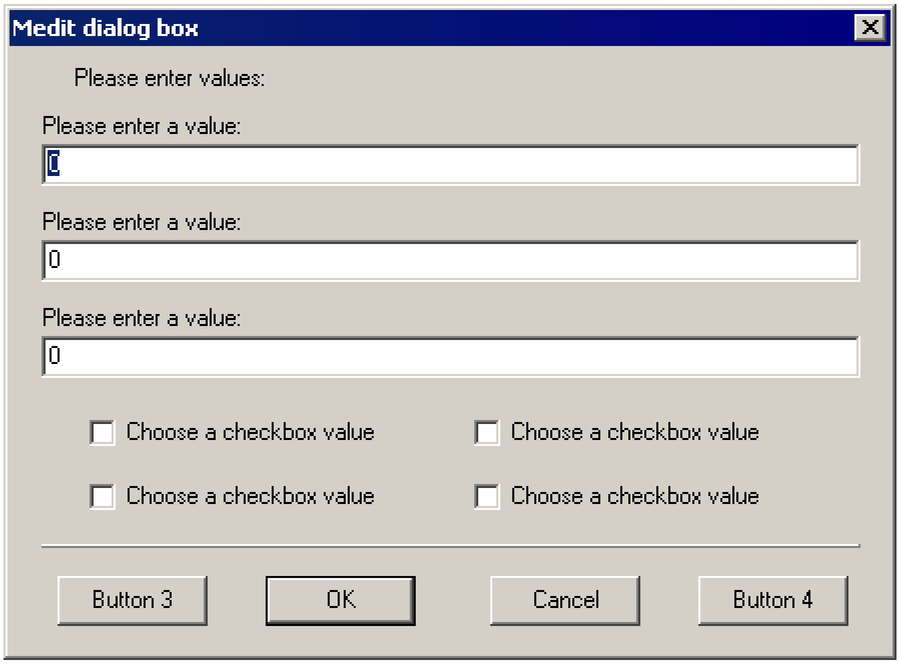
Code Listing 9-57

```
// Make a string list of colors.
New StringList NewVar(vColors);
Add Member('Red') To(vColors);
Add Member('Green') To(vColors);
Add Member('Blue') To(vColors);

// Initialize a return index variable.
Set vIndex = 0;

// Display the dialog box until vIndex is not equal to zero.
Loop While(vIndex = 0)
  // Display a ScrollBox dialog of the colors.
  DialogBox Type(ScrollBox) Title('Please select a color:')
    Caption('Colors') List(vColors) ReturnIndex(vIndex)
    Button(vButton) NewVar(vColor);
  // If the dialog box is cancelled, exit the script.
  If vButton = CancelButton
    LeaveSub;
  EndIf
EndLoop

Display vColor;
```

The **Medit** dialog box can be customized to display up to three text fields, four checkboxes with corresponding labels. You also add one or two optional buttons, in addition to the default OK and Cancel buttons. Here is an **Medit** dialog box with all of the options in use.

Code Listing 9-58

```
// Medit dialog box.
DialogBox Type(Medit) Title('Medit dialog box')
  Title2('Please enter values:')
  Label1('Please enter a value:') Edit1(vEdit1)
  Label2('Please enter a value:') Edit2(vEdit2)
  Label3('Please enter a value:') Edit3(vEdit3)
  CheckBox1Label('Choose a checkbox value') CheckBox1(vCheckBox1)
  CheckBox2Label('Choose a checkbox value') CheckBox2(vCheckBox2)
  CheckBox3Label('Choose a checkbox value') CheckBox3(vCheckBox3)
  CheckBox4Label('Choose a checkbox value') CheckBox4(vCheckBox4)
  Button3Label('Button 3') Button4Label('Button 4')
  Button(vButton);
```



Medit dialog box

Please enter values:

Please enter a value: 0

Please enter a value: 0

Please enter a value: 0

□ Choose a checkbox value   □ Choose a checkbox value

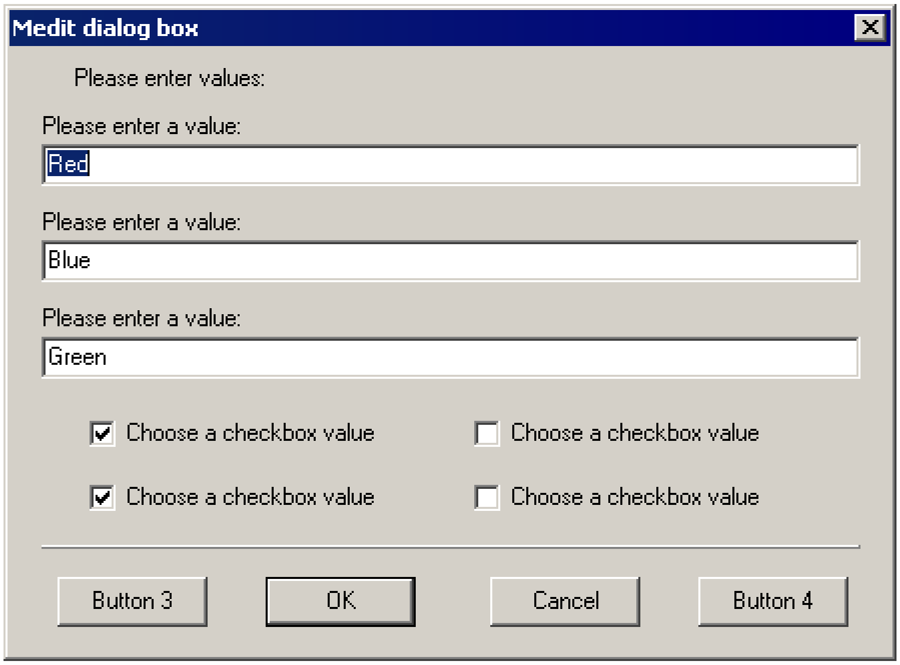
□ Choose a checkbox value   □ Choose a checkbox value

You can set defaults for the **Medit** dialog by assigning values to the **Edit** and **CheckBox** variables before displaying the dialog box. For **CheckBox** variables, use a **0** (zero) for unchecked and a **1** (one) for checked.

Code Listing 9-59

```
// Set some defaults for the dialog box.
Set vEdit1 = 'Red';
Set vEdit2 = 'Blue';
Set vEdit3 = 'Green';
Set vCheckBox1 = 1;
Set vCheckBox3 = 1;
```

```
// Medit dialog box.
DialogBox Type(Medit) Title('Medit dialog box')
  Title2('Please enter values:')
  Label1('Please enter a value:') Edit1(vEdit1)
  Label2('Please enter a value:') Edit2(vEdit2)
  Label3('Please enter a value:') Edit3(vEdit3)
  CheckBox1Label('Choose a checkbox value') CheckBox1(vCheckBox1)
  CheckBox2Label('Choose a checkbox value') CheckBox2(vCheckBox2)
  CheckBox3Label('Choose a checkbox value') CheckBox3(vCheckBox3)
  CheckBox4Label('Choose a checkbox value') CheckBox4(vCheckBox4)
  Button3Label('Button 3') Button4Label('Button 4')
  Button(vButton);
```



Medit dialog box

Please enter values:

Please enter a value: Red

Please enter a value: Blue

Please enter a value: Green

☑ Choose a checkbox value □ Choose a checkbox value

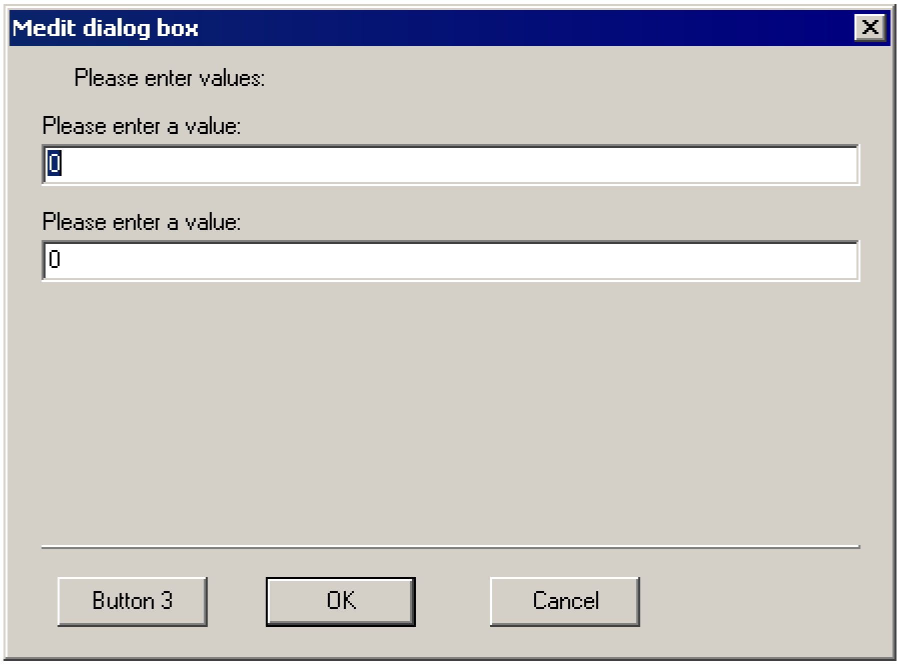
☑ Choose a checkbox value □ Choose a checkbox value

You can determine the values that the user entered by testing the values of the variables. The **Button** parameter will return an integer value depending on which button the user pressed. **Button3** returns 3, **OK** returns **0** (**OKButton**), **Cancel** returns **1** (**CancelButton**), and **Button4** returns 4.

If you leave off some of the parameters, the items will not appear on the dialog box, but the overall size of the dialog box does not change.

Code Listing 9-60

```
// Medit dialog box.
DialogBox Type(Medit) Title('Medit dialog box')
  Title2('Please enter values:')
  Label1('Please enter a value:') Edit1(vEdit1)
  Label2('Please enter a value:') Edit2(vEdit2)
  Button3Label('Button 3') Button(vButton);
```



Showing a Script’s Progress

Some scripts will take a fair amount of time to run, for example, scripts that run on all of the files in a book or folder. It is nice for your user to be able to “see” a script’s progress as it runs, or at least to know that the script is still running. FrameScript does not have any built in progress dialogs, but there are some creative methods that you can use to show a script’s progress.

Document or Book Status Bar

The lower left hand corner of a book or document window is called the Status Bar. You can write messages to a book or document’s Status Bar by using its **StatusLine** property. When you are finished writing messages to it, you must set it to an empty string so that FrameMaker can use it for its own messages.

Here is an example of a script that processes all of the paragraphs in a document. First it counts the paragraphs so that it can display the total in the Status Bar. Then it displays the current paragraph that it is processing. Finally, it resets the Status Bar to an empty string, so that FrameMaker can use it for its messages when the script is done. Note that the script does not do anything useful, it simply illustrates the concept of showing a script’s progress.

Code Listing 9-61

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Write a message to the status bar.
Set vCurrentDoc.StatusLine = 'Counting paragraphs...';

// Count the paragraphs in the document.
Set vTotal = 0;
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  Set vTotal = vTotal + 1; // Increment the counter.
EndLoop
```

```
// Process the paragraphs in the document.
Set n = 0;
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  Set n = n + 1; // Increment the counter.
  // Write the paragraph text to the Console.
  Write Console vPgf.Text;
  // Write a progress message to the status bar.
  Set vCurrentDoc.StatusLine =
    'Processing paragraph '+n+' of '+vTotal;
EndLoop

// IMPORTANT: Reset the status bar to a NULL string.
Set vCurrentDoc.StatusLine = '';
```

The **Displaying** property must be set to **1** (True) in order to set the **StatusLine** property. If you have **Displaying** set to **0**, you must set it to **1** before you write to the Status Bar. Here is how this would work with the previous script.

Code Listing 9-62

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Write a message to the status bar.
Set vCurrentDoc.StatusLine = 'Counting paragraphs...';

// Turn off the document display to prevent flicker and speed
// the script.
Set Displaying  = 0;

// Count the paragraphs in the document.
New Real NewVar(vTotal) Value(0);
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  Set vTotal = vTotal + 1; // Increment the counter.
EndLoop

// Process the paragraphs in the document.
New Real NewVar(n) Value(0);
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  Set n = n + 1; // Increment the counter.
  // Write the paragraph text to the Console.
  Write Console vPgf.Text;
  // Write a progress message to the status bar.
  Set Displaying = 1; // Necessary for the next command.
  Set vCurrentDoc.StatusLine =
    'Processing paragraph '+n+' of '+vTotal;
  Set Displaying = 0;
EndLoop

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;

// IMPORTANT: Reset the status bar to a NULL string.
Set vCurrentDoc.StatusLine = '';
```

You can use math to express the progress as a percentage. Replace the Set **vCurrentDoc.StatusLine** line with the following lines.

Code Listing 9-63

```
New Integer NewVar(vPercent) Value(n/vTotal*100);
Set vCurrentDoc.StatusLine =
  'Processing paragraphs: '+vPercent+'% complete';
```

Before this will work, you must use real numbers instead of integers for the **n** and **vTotal** variables. To do this, replace these lines

```
Set vTotal = 0;
Set n = 0;
```

to

Code Listing 9-64

```
New Real NewVar(vTotal) Value(0);
New Real NewVar(n) Value(0);
```

In some cases, you may not want to use processing time to count the total number of objects ahead of time. Here is the original script modified so it doesn't show the total paragraph count.

Code Listing 9-65

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Write a message to the status bar.
Set vCurrentDoc.StatusLine = 'Counting paragraphs...';

// Turn off the document display to prevent flicker and speed
// the script.
Set Displaying  = 0;

// Process the paragraphs in the document.
Set n = 0;
Loop ForEach(Pgf) In(vCurrentDoc) LoopVar(vPgf)
  Set n = n + 1; // Increment the counter.
  // Write the paragraph text to the Console.
  Write Console vPgf.Text;
  // Write a progress message to the status bar.
  Set Displaying = 1; // Necessary for the next command.
  Set vCurrentDoc.StatusLine =
    'Processing paragraph '+n;
  Set Displaying = 0;
EndLoop

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;

// IMPORTANT: Reset the status bar to a NULL string.
Set vCurrentDoc.StatusLine = '';
```

Using a book's status bar is basically the same as using a document's—you simply set the book's **StatusLine** property to the desired message. And, you must set it back to an empty string when you are finished. In most cases, you can display the book component's name that you are currently processing.

Code Listing 9-66

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Loop through all of the components in the book.
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  // Do something to the component here.
  Write Console vBookComp.Name;
  // Write to the book's status bar.
  Set vCurrentBook.StatusLine = 'Processing '+vBookComp.Name;
EndLoop

// IMPORTANT: Restore the book's status bar.
Set vCurrentBook.StatusLine = '';
```

The book component's **Name** property displays the full path to the book component, which may not fit in the book's status bar. It is usually better to display the base file name instead of the whole path. Here is the modified code.

Code Listing 9-67

```
// Set a variable for the active book.
Set vCurrentBook = ActiveBook;

// Loop through all of the components in the book.
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  // Do something to the component here.
  Write Console vBookComp.Name;
  // Get the base file name from the book component path.
  Find String(DIRSEP) InString(vBookComp.Name) Backward
    ReturnPos(vPos);
  Get String FromString(vBookComp.Name) StartPos(vPos+1)
    NewVar(vBasename);
  // Write to the book's status bar.
  Set vCurrentBook.StatusLine = 'Processing '+vBasename;
EndLoop

// IMPORTANT: Restore the book's status bar.
Set vCurrentBook.StatusLine = '';
```

10 Autonumber Report – Part 1

This script will give us a report of autonumber usage in a document or book. This will be useful for troubleshooting autonumbering problems that can crop up in documents. The script will produce a report file with a 5-column table. The table will show the paragraph format name, autonumber format, paragraph number, and paragraph text for each autonumber paragraph in the document or book. A fifth column will show if the paragraph has overrides from its paragraph format definitions. Each row of the table will contain a hypertext link to the autonumber paragraph to assist in correcting autonumber problems.

AutoNumberReport.fsl will illustrate the following FrameScript concepts:

- Making a log or report file.
- Making a table and writing data to it.
- Adding and deleting table rows.
- Making hypertext links.

Testing for the Right Conditions

We will begin the script by making it work on the active document before expanding it to work with a book. We need to first test for an active document. Because the report will contain hypertext links back to the document, we also want to make sure that the document has been saved. An untitled document must be saved before you can link to it from another document.

Code Listing 10-1

```
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  Set vCurrentDoc = ActiveDoc;
  If vCurrentDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub;
  EndIf
EndIf
```

Try running the script with no document open and with an untitled document so you can see the messages.

Making the Report File

Before we start looking for autonumber paragraphs in the active document, we need to make a report file. We will use the **New Document** command to make a new document. Try the following code.

Code Listing 10-2

```
New Document Portrait NewVar(vReportDoc);
```

This command simply makes a new document based on FrameMaker's built-in Portrait template. This will work, but since the report table will contain 5 columns, it will be better to make a landscape document with smaller margins. Close the document and use this code instead.

Code Listing 10-3

```
New Document NewVar(vReportDoc) Width(11in) Height(8.5in)
  TopMargin(.75in) BottomMargin(.75in) LeftInsideMargin(.5in)
  RightOutsideMargin(.5in);
```

This form of the **New Document** command allows you to supply specifications for the new document. This is similar to choosing New > Document and clicking the Custom button. (See "Creating New Documents and Books" on page 4-5.) In the final script, we will use the **Invisible** option to hide the document while we write the data to it. For now, we will keep it visible so we can see that the table is added correctly.

Adding the Report Table

We will add the report table to the first paragraph in the document. As we work through this section, close the previously created report file before running the code again.

Code Listing 10-4

```
Set vReportPgf = vReportDoc.MainFlowInDoc.FirstPgfInFlow;
New Table NewVar(vReportTbl) TextLoc(vReportPgf) Format('Format A')
  NumCols(5) HeaderRows(1) BodyRows(1);
```

After you run the script, you can see that the table is inserted, but will need some formatting, particularly with the column widths. Before resizing the table column widths, let's add the table column headings.

Code Listing 10-5

```
Set vReportRow = vReportTbl.FirstRowInTbl;
Set vReportCell = vReportRow.FirstCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Format';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Autonumber Format';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Number';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Text';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Override?';
```

We need to resize the table columns by hand, then we can transfer the values to the script. To get started, follow these steps.

1. Run the script.
2. Choose View > Options, and change the Display Units to Points.

3. Check the Grid Spacing check box, and set the Grid Spacing to 1pt, and click Set.
4. Select the entire table and choose Table > Resize Columns.
5. Click By Scaling to Widths Totalling, and click Set.

Now you can resize the columns by hand, but keep the total table width the same as the text column (720 points). The Paragraph Text columns should be the widest and the Overrides column the narrowest. We can make final adjustments after we test the script with data.

My initial values for the table column widths are 120, 120, 120, 290, 70. To set our table columns with these widths, we need to put the values in a **MetricList** and apply that to the table.

Code Listing 10-6

```
New MetricList NewVar(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(290) To(vTblColWidths);
Add Member(70) To(vTblColWidths);
Set vReportTbl.TblColWidths = vTblColWidths;
```

Close any open report files and try the code. A new report should open with the table scaled to the correct widths. One last thing needs to be done before the report table is ready to write to. We need to make a variable for the first body row in the table. We already have the vReportRow variable for the first row in the table, so we can use the following code.

Code Listing 10-7

```
Set vReportRow = vReportRow.NextRowInTbl;
Set vFirstRow = vReportRow;
```

We have also created an extra variable (**vFirstRow**) for the first body row in the table; you will see why later in the script.

Using a Subroutine to Make the Report

Our script already contains about 35 lines. When the body of a script gets too long, it is usually best to break things up into subroutines. By doing this, we make the script more manageable and readable. In addition, we can often reuse subroutines in other scripts. The code that makes the report and report table is a good candidate for a subroutine. Here is the new MakeReport subroutine.

Code Listing 10-8

```
Sub MakeReport
//
New Document NewVar(vReportDoc) Width(11in) Height(8.5in)
  TopMargin(.75in) BottomMargin(.75in) LeftInsideMargin(.5in)
  RightOutsideMargin(.5in) Invisible;

Set vReportPgf = vReportDoc.MainFlowInDoc.FirstPgfInFlow;
New Table NewVar(vReportTbl) TextLoc(vReportPgf) Format('Format A')
  NumCols(5) HeaderRows(1) BodyRows(1);
```

```
Set vReportRow = vReportTbl.FirstRowInTbl;
Set vReportCell = vReportRow.FirstCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Format';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Autonumber Format';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Number';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Paragraph Text';
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) 'Override?';

New MetricList NewVar(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(120) To(vTblColWidths);
Add Member(290) To(vTblColWidths);
Add Member(70) To(vTblColWidths);
Set vReportTbl.TblColWidths = vTblColWidths;

Set vReportRow = vReportRow.NextRowInTbl;
Set vFirstRow = vReportRow;
//
EndSub
```

The **New Document** command now has the **Invisible** option added to it; make sure it is added to your code. Below is the body of the script. The body of the script is anything that is not inside of a subroutine. Make sure the body of the script comes before the subroutine.

Code Listing 10-9

```
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  Set vCurrentDoc = ActiveDoc;
  If vCurrentDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub;
  EndIf
EndIf

Run MakeReport;

Set vReportDoc.IsOnScreen = 1;
```

The second last line calls the **MakeReport** subroutine to make the report. Since the report is initially made invisible, we need the last line to make it visible at the end of the script. If you try the script, you will see that it works just like it did before we made the subroutine. As we continue the script, the new code will go between the last two lines; in other words,

```
Set vReportDoc.IsOnScreen = 1;
```

will be the last line in the body of the script. We make the script invisible so that you don't see it until the script is finished.

Looping Through the Main Document

Now it is time to loop through the document and find autonumber paragraphs so we can write them to the report. Make sure your test document has plenty of autonumber paragraphs. We want to list the autonumber paragraphs in document order so we will use a standard method to loop through the paragraphs. Here is the shell of the loop with the test to see if the paragraph is an autonumber paragraph.

Code Listing 10-10

```
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.PgfIsAutoNum
    // ... Do something here ...
  EndIf
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

There a couple of things to be aware of when using this loop. One, this code will not work on documents that don't have a Main Flow. If you have a Main Flow but there are some disconnected pages in your document, the disconnect pages will be skipped. Most documents will have a Main Flow, so we won't worry about this here.

Second, this loop will skip paragraphs in tables. We will need to include table paragraphs, so here is the revised loop with a subroutine to process table paragraphs. Remember to put the subroutine somewhere after the body of the script.

Code Listing 10-11

```
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.PgfIsAutoNum
    // ... Do something here ...
  EndIf
  Get TextList InObject(vPgf) TblAnchor NewVar(vTextList);
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
    Run ProcessTable vTbl(vTblAnchor.TextData);
  EndLoop
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop

Sub ProcessTable
//
If vTbl.FirstPgf
  Set vTblPgf = vTbl.FirstPgf;
  Loop While(vTblPgf)
    If vTblPgf.PgfIsAutoNum
      // ... Do something here ...
    EndIf
    Set vTblPgf = vTblPgf.NextPgfInFlow;
  EndLoop
EndIf
```

```
Set vCell = vTbl.FirstRowInTbl.FirstCellInRow;
Loop While(vCell)
  Set vTblPgf = vCell.FirstPgf;
  Loop While(vTblPgf);
    If vTblPgf.PgfIsAutoNum
      // ... Do something here ...
    EndIf
    Set vTblPgf = vTblPgf.NextPgfInFlow;
  EndLoop
  Set vCell = vCell.NextCellInTbl;
EndLoop
//
EndSub
```

Let's examine what we have added to the revised loop. For each paragraph, we want to test to see if it contains any tables. The **Get TextList** command with the **TblAnchor** option finds any table anchors in the paragraph and puts them in a **vTextList** variable. (See "Working With Text Lists and Text Items" on page 5-28.) If there are any tables, the loop is entered and each table is processed in the **ProcessTable** subroutine.

The **ProcessTable** subroutine is a series of loops that processes each paragraph in the table. Let's look at this briefly, so you can see what is going on.

```
If vTbl.FirstPgf
```

The first line tests to see if the table has a **FirstPgf** object. The **FirstPgf** object of a table is its table title; not all tables have titles, so we use an **If/EndIf** statement to test for it. If the table has a title, we set **vTblPgf** variable for the first paragraph in the title.

```
Set vTblPgf = vTbl.FirstPgf;
```

Table titles and cells are like text flows, so we use the same type of loop as we do in the body of the script.

```
Loop While(vTblPgf)
  If vTblPgf.PgfIsAutoNum
    // ... Do something here ...
  EndIf
  Set vTblPgf = vTblPgf.NextPgfInFlow;
EndLoop
```

The second part of the subroutine loops through each cell in the table and processes each paragraph in each cell.

Writing the Data to the Report

Now we get to the meat of the script, writing the autonumber data to the report. We will do this in a subroutine to keep things neat and organized. The subroutine will be called whenever an autonumber paragraph is encountered; this is where are existing codes says

```
// ... Do something here ...
```

When writing scripts, it's best to break things down into individual tasks and get each one working before tying the pieces together. We will do that here by leaving the main script behind and working on the code to write the data to the report table. To set this up, save the existing script, and follow these steps.

1. Make a new FrameMaker document and save it.
2. Add some text to the first paragraph and apply the Numbered paragraph format to the text.
3. Add a second paragraph to the document.
4. Add a 5-column table to the second paragraph with 1 heading row and 1 body row.
5. Start a new script with the following code.

Code Listing 10-12

```
Set vCurrentDoc = ActiveDoc;

Set vReportTbl = SelectedTbl;
Set vReportRow = vReportTbl.FirstRowInTbl.NextRowInTbl;
Set vFirstRow = vReportRow;

Set vAutoNumPgf = ActiveDoc.MainFlowInDoc.FirstPgfInFlow;
```

What we are doing with these first few lines is staging the script so that it is set up like the main script will be when it processes an autonumber paragraph. The lines are necessary in this temporary script because we are using an existing table rather than creating one with the script, but we will not use them in the final script. We will write the autonumber data from the first paragraph into the table so that we can see it happen. The report table in the finished script will be invisible, so this temporary file is a way of testing the code without “flying blind.”

First, we want to add a new table row. We already have a blank table row, but it is easier to add a new row every time we add data to the table. That way, we don’t have to test for a blank row, and we can easily delete the first body row at the end of the script. (See “Adding Rows and Columns” on page 7-7.)

Code Listing 10-13

```
New TableRows TableObject(vReportTbl) RowObject(vReportRow)
  Direction(Below) BodyRows(1);
Set vReportRow = vReportRow.NextRowInTbl;
```

To try this, click in the table, and run the code. You should see a new row added to the end of the table. Delete this row before proceeding.

Now we can write the data to the new table row.

Code Listing 10-14

```
Set vReportCell = vReportRow.FirstCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.Name;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.AutoNumString;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.PgfNumber;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.Text;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.FormatOverride;
```

Click in the table and run the code. A new row will be added to the table and the data from the first paragraph will be added to the cells.

Now we need the code to add the Hypertext marker to the autonumber paragraph. We will use the same syntax that is used in FrameMaker generated files, such as tables of contents, and indexes. To examine the syntax, follow these steps.

1. Open an existing FrameMaker document.
2. Generate a Table of Contents (TOC). Make sure you have the Create Hypertext Links checkbox checked.
3. Go to the TOC document and choose View > Text Symbols. This will allow you to see the markers at the beginning of each paragraph.
4. Select one of the markers and choose Special > Markers to see the marker text. The Hypertext marker will look something like this:

```
openObjectId AutoNumberReport.fm:2 999383
```

This kind of Hypertext marker provides a link to a particular paragraph in the target document. The **openObjectId** keyword identifies the link, followed by the filename of the target document. The number **2** after the colon means that the target of the link is a paragraph, and the six-digit number at the end is the Unique Id of the target paragraph. Every object in FrameMaker contains a Unique Id that is like a unique serial number for the object.

We will insert the Hypertext marker in the Paragraph Format column of each table row. We can use the temporary document to develop and test the code, which we will put in a subroutine.

Code Listing 10-15

```
Sub AddHypertextMarker
//
New Marker MarkerName('Hypertext') TextLoc(vMarkerPgf)
  NewVar(vMarker);
Set vMarkerText = 'openObjectId '+vCurrentDoc.Name+':2 '+vUnique;
Set vMarker.MarkerText = vMarkerText;
//
EndSub
```

Add this subroutine to the bottom of the temporary script and add the following line to the end of the body of the script (before the subroutine).

```
Run AddHypertextMarker
vMarkerPgf(vReportRow.FirstCellInRow.FirstPgf)
  vUnique(vAutoNumPgf.Unique);
```

Make sure the temporary FrameMaker document has been saved, click in the table and run the temporary script. A new body row with data should be added to the table. Choose View > Text Symbols and you should see the marker anchor at the beginning of the first cell in the new row. Hold down the Control+Alt (Control+Option on the Macintosh) keys and click on the link to try it out. It should highlight the first paragraph in the document.

Before leaving our temporary document, let's add the code to delete the blank table row. We will do this in the final script just before the report is displayed. You will now see why we stored the first body row in the **vFirstRow** variable in the last line of the **MakeReport** subroutine.

Code Listing 10-16

```
Delete TableRows TableObject(vReportTbl) RowObject(vFirstRow)
NumRows(1);
```

Delete all body rows except the first, click in the table, and run the script. The results will be the same as last time, except that we no longer have the extra body row.

We can now add our temporary code to the main script. First, add the **Delete TableRows** line to the body of the script, just before

```
Set vReportDoc.IsOnScreen = 1;
```

The rest of the temporary code body will go into a **WriteData** subroutine after the body of the main script. Remember, we don't need the first five lines from the temporary script, so begin at the command to create a new table row. Make sure your script matches what is shown below.

Code Listing 10-17

```
Sub WriteData
//
New TableRows TableObject(vReportTbl) RowObject(vReportRow)
  Direction(Below) BodyRows(1);
Set vReportRow = vReportRow.NextRowInTbl;

Set vReportCell = vReportRow.FirstCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.Name;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.AutoNumString;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.PgfNumber;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.Text;
Set vReportCell = vReportCell.NextCellInRow;
New Text Object(vReportCell.FirstPgf) vAutoNumPgf.FormatOverride;
```

```
Run AddHypertextMarker
vMarkerPgf(vReportRow.FirstCellInRow.FirstPgf)
  vUnique(vAutoNumPgf.Unique);
//
EndSub
```

Copy the **WriteData** and **AddHypertextMarker** subroutines into the main script and replace the lines

```
// ... Do something here ...
```

with one of the following calls to the **WriteData** subroutine (the different calls to the same subroutine are required so that the correct parameter is passed in each case):

In the main loop through the document paragraphs, use this:

Code Listing 10-18

```
Run WriteData vAutoNumPgf(vPgf);
```

In the **ProcessTable** loop, use the following line (in two places):

Code Listing 10-19

```
Run WriteData vAutoNumPgf(vTblPgf);
```

Give the main script a try on your test document.

Making the Script Work for a Book

Autonumber problems generally surface with FrameMaker books. In this section, we will modify the script to work with a book, as well as with a single document. The techniques that we use to do this can be used in any script that you want to expand for use with a book.

Since the script we have so far works fine on a single document, it is best to make a copy of it before we modify it to work with a book. Starting at the beginning of the script, we must modify the test for an active document to include a test for an active book.

Code Listing 10-20

```
If (ActiveDoc = 0) and (ActiveBook = 0)
  MsgBox 'There is no active document or book.    ';
  LeaveSub;
Else
  If ActiveBook
    Set vCurrentBook = ActiveBook;
  Else
    Set vCurrentDoc = ActiveDoc;
    If vCurrentDoc.Name = ''
      MsgBox 'Please save the document and rerun the script.    ';
      LeaveSub;
    EndIf
  EndIf
EndIf
```

Since the script will work on either a document or book, we modify the first line to check for either an active document or an active book. The line after the **Else** statement checks to see what is active. If it is a book, a variable is set for the book name; if it is a document, we use the same test we used before to make sure the document is not untitled.

The next line is

Code Listing 10-21

```
Run MakeReport;
```

which is fine because we will want a single report whether we are processing a book, or a single document.

Now, we have to determine whether the script is running on a book or document and call an appropriate subroutine.

Code Listing 10-22

```
If vCurrentBook
  Run ProcessBook;
Else
  Run ProcessDoc;
EndIf
```

You can probably imagine which each of the subroutines has to do. The **ProcessBook** subroutine will have to open each file in the book, gather up the autonumber information, and write it to the report. The **ProcessDoc** subroutine will simply do what the original script did for the active document. Because of this, we already have the **ProcessDoc** code in our script.

Code Listing 10-23

```
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.PgfIsAutoNum
    Run WriteData vAutoNumPgf(vPgf);
  EndIf
  Get TextList InObject(vPgf) TblAnchor NewVar(vTextList);
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
    Run ProcessTable vTbl(vTblAnchor.TextData);
  EndLoop
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

What we need to do is to make this code a subroutine and move it after the body of the script. Here is the code made into a subroutine and placed in the correct place in the script.

Code Listing 10-24

```
Run MakeReport;
```

```
If vCurrentBook
  Run ProcessBook;
Else
  Run ProcessDoc;
EndIf

Delete TableRows TableObject(vReportTbl) RowObject(vFirstRow)
  NumRows(1);

Set vReportDoc.IsOnScreen = 1;

Sub ProcessDoc
//
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
Loop While(vPgf)
  If vPgf.PgfIsAutoNum
    Run WriteData vAutoNumPgf(vPgf);
  EndIf
  Get TextList InObject(vPgf) TblAnchor NewVar(vTextList);
  Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
    Get Member Number(n) From(vTextList) NewVar(vTblAnchor);
    Run ProcessTable vTbl(vTblAnchor.TextData);
  EndLoop
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
//
EndSub
```

If you run the script now on a single document, it will work exactly like it did before. If you run it on a book, you will get an error because the **ProcessBook** subroutine does not exist in the script. Our next step will be to write the **ProcessBook** subroutine.

ProcessBook is a fairly simple loop. Working through each book component in order, it attempts to open each one, and then calls the **ProcessDoc** subroutine for that component. Here is the shell of the subroutine.

Code Listing 10-25

```
Sub ProcessBook
//
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  Open Document File(vBookComp.Name) NewVar(vCurrentDoc);
  Run ProcessDoc;
  Close Document DocObject(vCurrentDoc) IgnoreMods;
EndLoop
//
EndSub
```

There are several problems that we have to address with this loop.

- What if the document is not opened successfully in the first command in the loop? The script will attempt to run the next two lines on a document that is not opened.
- The **Close Document** command closes the document without saving changes. That is because the script is only reading data from the document. But what if the document was already opened when the script starts?

- If a document has missing fonts or graphics, or unresolved cross-references, the script will stop while prompting you with messages. In addition, there is no reason to make the document visible as it opens.

Let's address these problems in order. For the first one, we can use the built-in **ErrorCode** variable to see if the document was opened successfully.

Code Listing 10-26

```
Set ErrorCode = 0;
Open Document File(vBookComp.Name) NewVar(vCurrentDoc);
If ErrorCode not= 0
  Write Console vBookComp.Name+' could not be opened.';
Else
  Run ProcessDoc;
  Close Document DocObject(vCurrentDoc) IgnoreMods;
EndIf
```

To address the second problem, we will make use of a subroutine that checks to see if the book component is already opened. If the book component is already open, the script will leave it open when it is finished. Here is the subroutine that you can put at the bottom of the script. The code is commented so you can follow it.

Code Listing 10-27

```
Sub IsDocOpen Using vFileName vDocObj vDocIsOpen
//
// Initialize a variable for the document object.
Set vDocObj = 0;
// Initialize a variable that indicates if the document is open.
Set vDocIsOpen = 0;
// Uppercase the book component name that is passed to the sub.
Get String FromString(vFileName) Uppercase NewVar(vUFileName);
// Loop through all of the open documents.
Loop ForEach(Doc) In(Session) LoopVar(vTestDocObj)
  // Uppercase the document name.
  Get String FromString(vTestDocObj.Name) Uppercase
    NewVar(vTestName);
  // See if the document name matches the book component name.
  If vTestName = vUFileName
    // If it does, return the document object variable.
    Set vDocObj = vTestDocObj;
    // Return the variable indicating that the document is open.
    Set vDocIsOpen = 1;
    // Leave the loop (no need to check any more documents).
    LeaveLoop;
  EndIf
EndLoop
//
EndSub
```

Here is how the **IsDocOpen** subroutine is called by the **ProcessBook** subroutine.

Code Listing 10-28

```
Sub ProcessBook
//
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  Run IsDocOpen vFileName(vBookComp.Name)
    Returns vDocObj(vCurrentDoc) Returns vDocIsOpen;
  If vCurrentDoc = 0
    Set ErrorCode = 0;
    Open Document File(vBookComp.Name) NewVar(vCurrentDoc);
    If ErrorCode not= 0
      Write Console vBookComp.Name+' could not be opened.';
    EndIf
  EndIf
  If vCurrentDoc not= 0
    Run ProcessDoc;
    If vDocIsOpen = 0
      Close Document DocObject(vCurrentDoc) IgnoreMods;
    EndIf
  EndIf
EndLoop
//
EndSub
```

When the **IsDocOpen** subroutine is called, it will return a **vCurrentDoc** object if the book component is already open. If the document is not open, **vCurrentDoc** will be zero, and the script will attempt to open the book component. If the book component cannot be opened, a message will be written to the console, and **vCurrentDoc** will still be zero. This will cause the rest of the commands inside the loop to be skipped.

If **vCurrentDoc** is an object (and not zero), the **ProcessDoc** subroutine will be called. After that, the **vDocIsOpen** variable is tested. If it is zero, that means that the document wasn't open before the script began, and it will be closed. If the document was already open, it will be left open.

Now we can solve the third problem by adding some parameters to the **Open Document** command.

Code Listing 10-29

```
Open Document File(vBookComp.Name) NewVar(vCurrentDoc)
  AlertUserAboutFailure(False) MakeVisible(False)
  FontNotFoundInDoc(OK) UpdateXRefs(No)
  RefFileNotFound(AllowAllRefFilesUnFindable);
```

Here is an explanation of the parameters.

- **AlertUserAboutFailure(False)**. This prevents the script from displaying error messages to you as it opens the files. This suppresses the normal dialog boxes that report missing fonts, graphics, etc.
- **MakeVisible(False)**. This prevents the document from being visible after it is opened. This speeds the script and keeps the screen "clean" as it processes the document.

- **FontNotFoundInDoc(OK)**. This allows the document to be opened, even if there are missing fonts on your system. Missing fonts will not cause a problem with this script, so it should not prevent it from running.
- **UpdateXRefs(No)**. This allows the document to be opened without updating cross-references. This will prevent unresolved cross-references from stopping the script; they should not cause a problem with this script.
- **RefFileNotFound(AllowAllRefFilesUnFindable)**. This allows the document to be opened, even if imported graphics cannot be found. Missing graphics should not cause a problem with this script.

You will have to decide which of these and other available parameters are appropriate in your own scripts. In a script that deals with cross-references, for example, it might not be appropriate to open a document with unresolved cross-references.

Here is the finished ProcessBook subroutine. Put it in your script and give it a try on a book that you know has autonumbers its documents.

Code Listing 10-30

```
Sub ProcessBook
//
Loop ForEach(BookComponent) In(vCurrentBook) LoopVar(vBookComp)
  Run IsDocOpen vFileName(vBookComp.Name)
    Returns vDocObj(vCurrentDoc) Returns vDocIsOpen;
  If vCurrentDoc = 0
    Set ErrorCode = 0
    Open Document File(vBookComp.Name) NewVar(vCurrentDoc)
      AlertUserAboutFailure(False) MakeVisible(False)
      FontNotFoundInDoc(OK) UpdateXRefs(No)
      RefFileNotFound(AllowAllRefFilesUnFindable);
    If ErrorCode not= 0
      Write Console vBookComp.Name+' could not be opened.';
    EndIf
  EndIf
  If vCurrentDoc not= 0
    Run ProcessDoc;
    If vDocIsOpen = 0
      Close Document DocObject(vCurrentDoc) IgnoreMods;
    EndIf
  EndIf
EndLoop
//
EndSub
```

11 Autonumber Report – Part 2

With many scripts you write, you will find that they can be improved and enhanced, especially after you use them for awhile. There are several enhancements we can make to the AutonumberReport.fsl script. The first three will improve the performance of the script and the usability of the report, while the last one will improve the script's interface.

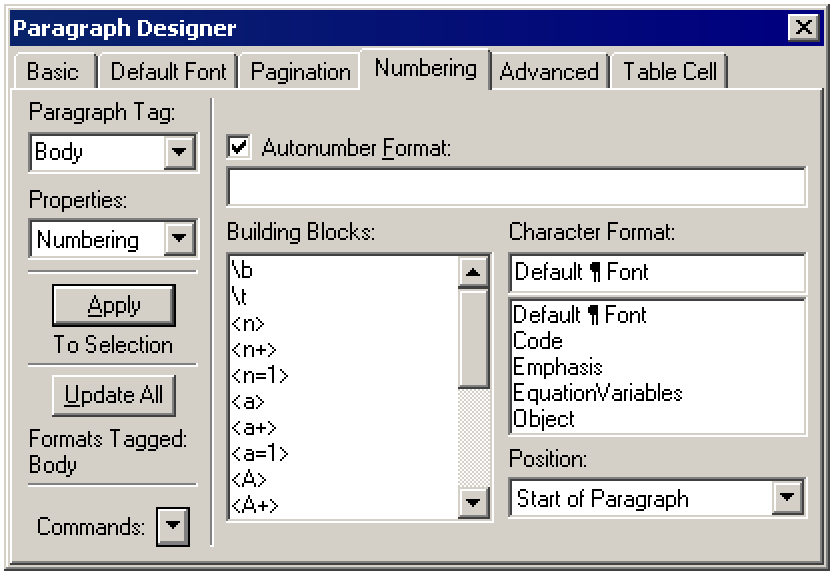
- Make sure blank autonumber fields are not reported. The Autonumber Format checkbox in the Paragraph Designer can be checked, but the field may be blank. There is no reason to include these paragraphs in the report.
- Report counter-type autonumbers only. Other autonumbers, such as bullets, dashes, etc., do not normally cause problems, so we can eliminate them from the report.
- Improve the appearance of the report.
- Report the progress of the script. The script may take some time to run, especially on a book, so it is helpful to show the user which document is being processed.

Refining the Autonumber Test

This is the existing code that tests if a paragraph is an autonumber paragraph.

```
If vPgf.PgfIsAutoNum
  Run WriteData vAutoNumPgf(vPgf);
EndIf
...
If vTblPgf.PgfIsAutoNum
  Run WriteData vAutoNumPgf(vTblPgf);
EndIf
```

The **PgfIsAutoNum** property is equivalent to the Autonumber Format checkbox in the Numbering tab of the Paragraph Designer.



Paragraph Tag: Body

Properties: Numbering

☑ Autonumber Format:

Character Format: Default ¶ Font

Position: Start of Paragraph

Formats Tagged: Body

As you can see from the above screenshot, it is possible for the Autonumber Format checkbox to be checked, but the field itself to be blank. In our existing script, paragraphs like this would be included in the autonumber report. Instead of testing for the **PgfIsAutoNum** setting, it would be better to test for the contents of the Autonumber Format field. To do this, we test the **AutoNumString** property. Here is the modified code.

Code Listing 11-1

```
If vPgf.AutoNumString not= ''
  Run WriteData vAutoNumPgf(vPgf);
EndIf
...
If vTblPgf.AutoNumString not= ''
  Run WriteData vAutoNumPgf(vTblPgf);
EndIf
```

We also want to eliminate autonumbers that are not “counters.” Counter autonumbers use the numeric, alphabetic, or Roman building blocks, in contrast to bullets or strings such as “Note” or “Important.” Counter autonumbers will contain at least one angled bracket (<) character, so we can use a test for this character in the autonumber string.

```
Find String('<') InString(vPgf.AutoNumString) ReturnStatus(vFound);
```

If the angled bracket exists in the autonumber string, **vFound** will return True (1). Here is the test in context.

Code Listing 11-2

```
If vPgf.AutoNumString not= ''
  Find String('<') InString(vPgf.AutoNumString)
    ReturnStatus(vFound);
  If vFound
    Run WriteData vAutoNumPgf(vPgf);
  EndIf
EndIf
...
If vTblPgf.AutoNumString not= ''
  Find String('<') InString(vTblPgf.AutoNumString)
    ReturnStatus(vFound);
  If vFound
    Run WriteData vAutoNumPgf(vTblPgf);
  EndIf
EndIf
```

Replace the code in your script with the modified code above and give it a try. In most cases, the script will run faster because it has less paragraphs to write to the report. More importantly, it should make the report more usable for troubleshooting incorrect autonumbers.

Improving the Appearance of the Report

There are several things that can be done to improve the appearance of the report.

- Use Arial or another san-serif font to make the report more readable on the screen.
- Turn off hyphenation.
- Remove the table title from the table.
- Add an explanatory line above the table so that the user knows that they can click to go to the paragraph.
- Display the report document without borders, rulers, and text symbols.
- Make the document view-only so that you can click on the hypertext links without using the Control-Alt modifier.
- Add an “Autonumber Report” label to the report’s title bar.
- Suppress the report if there are no autonumber paragraphs.

Changing the Report Font

There is a session property called **FontFamilyNames** that is a list of the fonts installed on your system. You can display this list by running this code.

Code Listing 11-3

```
Display FontFamilyNames;
```

The list may be too long to display on your screen, but you can use the following code to see if a particular font is available on your system. We will need to do this before we attempt to change the report font to Arial.

Code Listing 11-4

```
Find Member('Arial') InList(FontFamilyNames) ReturnPos(vPos);
Display vPos;
```

If **vPos** is greater than zero (0), then you know that the font is installed on your system. **vPos-1** also returns the *index* of the font in the list of the **FontFamilyNames**; this is an integer that is used to change the font in a paragraph or paragraph format. To see this, set a paragraph in a document to Arial, and put your cursor in the paragraph. Run the following code.

Code Listing 11-5

```
Find Member('Arial') InList(FontFamilyNames) ReturnPos(vPos);
Display vPos-1;

Set vPgf = TextSelection.Begin.Object;
Display vPgf.FontFamily;
```

Both numbers should be the same. So, to set a paragraph's font to Arial, you can use this code.

Code Listing 11-6

```
Find Member('Arial') InList(FontFamilyNames) ReturnPos(vPos);
Set vPgf = TextSelection.Begin.Object;
If vPos > 0
  Set vPgf.FontFamily = vPos-1;
EndIf
```

With this information at hand, let's add it to the **MakeReport** subroutine, right after the report is created.

```
Sub MakeReport
//
New Document NewVar(vReportDoc) Width(11in) Height(8.5in)
  TopMargin(.75in) BottomMargin(.75in) LeftInsideMargin(.5in)
  RightOutsideMargin(.5in) Invisible;

Find Member('Arial') InList(FontFamilyNames) ReturnPos(vPos);
If vPos > 0
  Loop ForEach(PgfFmt) In(vReportDoc) LoopVar(vPgfFmt)
    Set vPgfFmt.FontFamily = vPos-1;
  EndLoop
EndIf
```

To make sure the new paragraph font and hyphenation settings (which we will add next) are applied, you have to add this line just before the report document is displayed.

Code Listing 11-7

```
Import Formats DocObject(vReportDoc) FromDocObject(vReportDoc) Pgf
  RemoveExceptions;
```

The line above will should go just above the following line, which already appears in the script.

```
Set vReportDoc.IsOnScreen = 1;
```

Turning Off Hyphenation

To turn off hyphenation, you can add this loop next.

Code Listing 11-8

```
Loop ForEach(PgfFmt) In(vReportDoc) LoopVar(vPgfFmt)
  Set vPgfFmt.Hyphenate = 0;
EndLoop
```

Removing the Table Title

The code to remove the table title can go right after the above code in the **MakeReport** subroutine. This code will also set the space above the table to zero.

Code Listing 11-9

```
Get Object Type(TblFmt) Name('Format A') DocObject(vReportDoc)
  NewVar(vTblFmt);
Set vTblFmt.TblTitlePosition = TblNoTitle;
Set vTblFmt.TblSpaceAbove = 0;
```

Add Explanatory Text Above the Table

The following line already exists as the next line in the **MakeReport** subroutine.

```
Set vReportPgf =   vReportDoc.MainFlowInDoc.FirstPgfInFlow;
```

Use the following code to add explanatory text to the first paragraph and add another paragraph for the table.

Code Listing 11-10

```
New Text Object(vReportPgf)
  'Click in the first column of the row to go to the paragraph.';
New Pgf NewVar(vReportPgf) PrevObject(vReportPgf);
```

The rest of the MakeReport subroutine will be the same as in the original script.

If you try the script now, you will see that the report font is Arial, hyphenation is off, the explanatory text appears above the table, and that the table does not have a title.

Hiding Borders, Rulers, and Text Symbols

The following code further cleans up the display of the report document. It goes at the end of the script body, before the `Set vReport.IsOnScreen = 1;` statement.

Code Listing 11-11

```
Set vReportDoc.ViewBorders = 0;
Set vReportDoc.ViewRulers = 0;
Set vReportDoc.ViewTextSymbols = 0;
```

Making the Report View-Only

Add this code below the previous code.

```
Set vReportDoc.DocIsViewOnly = 1;
```

Adding the “Autonumber Report” Label

Finally, we want to add a label to the report’s title bar. FrameScript will not change the label on an invisible document, so you have to display the document first before changing the label. The script already has this line to display the report.

```
Set vReportDoc.IsOnScreen = 1;
```

So, simply add this after it.

Code Listing 11-12

```
Set vReportDoc.Label = 'Autonumber Report';
```

Run the script now and take a look at the report, including the title bar. You can also try the hypertext links in the left column of the table.

Suppressing an Empty Report

You will usually run this script on a document or book that you know contains autonumber paragraphs. But if you run it on a document or book that has none, the report will display with an empty table. While this is not a big problem, it would be cleaner and more professional if a message displayed instead. In addition, it is not a difficult enhancement to add.

The easiest way to do it is to add a variable near the beginning of the script, right after the **MakeReport** subroutine is called, but before the **ProcessBook** or **ProcessDoc** subroutines are called. We will initialize the variable to zero.

Code Listing 11-13

```
Set vReportWritten = 0;
```

At the end of the **WriteData** subroutine, add the following line.

Code Listing 11-14

```
Set vReportWritten = 1;
```

This ensures that this variable will be changed only if data is written to the table. Now, add this after the code in the body that calls **ProcessBook** and **ProcessDoc**.

Code Listing 11-15

```
If vReportWritten = 0
  MsgBox 'There are no autonumber paragraphs to report.    ';
  Close Document DocObject(vReportDoc) IgnoreMods;
  LeaveSub;
EndIf
```

Here is the code in context in the script.

Code Listing 11-16

```
...

Run MakeReport;

Set vReportWritten = 0;
```

```
If vCurrentBook
  Run ProcessBook;
Else
  Run ProcessDoc;
EndIf

If vReportWritten = 0
  MsgBox 'There are no autonumber paragraphs to report.    ';
  Close Document DocObject(vReportDoc) IgnoreMods;
  LeaveSub;
EndIf

...
```

Providing Progress Feedback to the User

This script can take a long time to run, particularly when it is run on a book. It would be nice to show the user which book component is being processed when the script is running on a book. We will do this by simply writing the book component name to the book's status area in its lower, left-hand corner.

Because we are only interested in showing progress when the script is running on a book, we can put our code in the **ProcessBook** subroutine. We will insert the code just before the following line.

```
Run ProcessDoc;
```

To write to the book status area, you simply use

```
Set vCurrentBook.StatusLine = 'Message to display.';
```

IMPORTANT: When you are done using the status area, you must reset it to a null string so that FrameMaker can use it for its own messages.

```
Set vCurrentBook.StatusLine = '';
```

We could simply use this for our progress message

```
Set vCurrentBook.StatusLine = 'Processing '+vBookComp.Name;
```

but **vBookComp.Name** will display the entire path of the book component name, which will not match the name in the book window, and will probably be truncated because if its length. Instead, we will parse the path off the name, leaving us with the file name only. Here is the code.

Code Listing 11-17

```
Find String(DIRSEP) InString(vBookComp.Name) Backward
  ReturnPos(vPos);
Get String FromString(vBookComp.Name) StartPos(vPos+1)
  NewVar(vFilename);
Set vCurrentBook.StatusLine = 'Processing '+vFilename;
```

Add this before the **Run ProcessDoc;** line in the **ProcessBook** subroutine. Be sure to add the following line at the end of the **ProcessBook** subroutine.

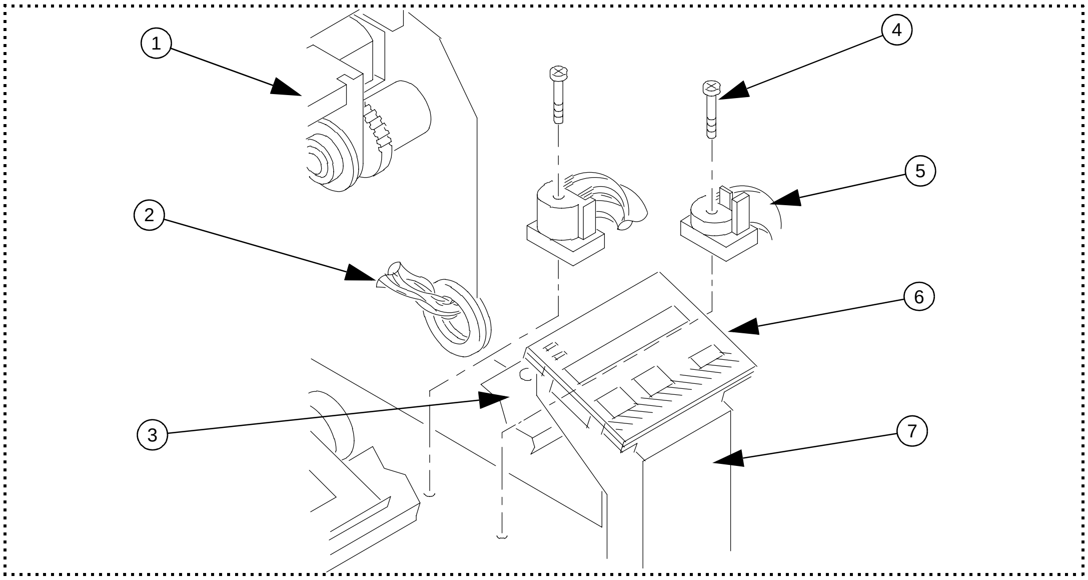
Code Listing 11-18

```
Set vCurrentBook.StatusLine = '';
```

Run the script on a book and take a look at the book status area as the script runs.

12 Automatic Callouts

This script will show how to automate the creation of complex, repeating FrameMaker objects, such as the callouts on the following illustration.

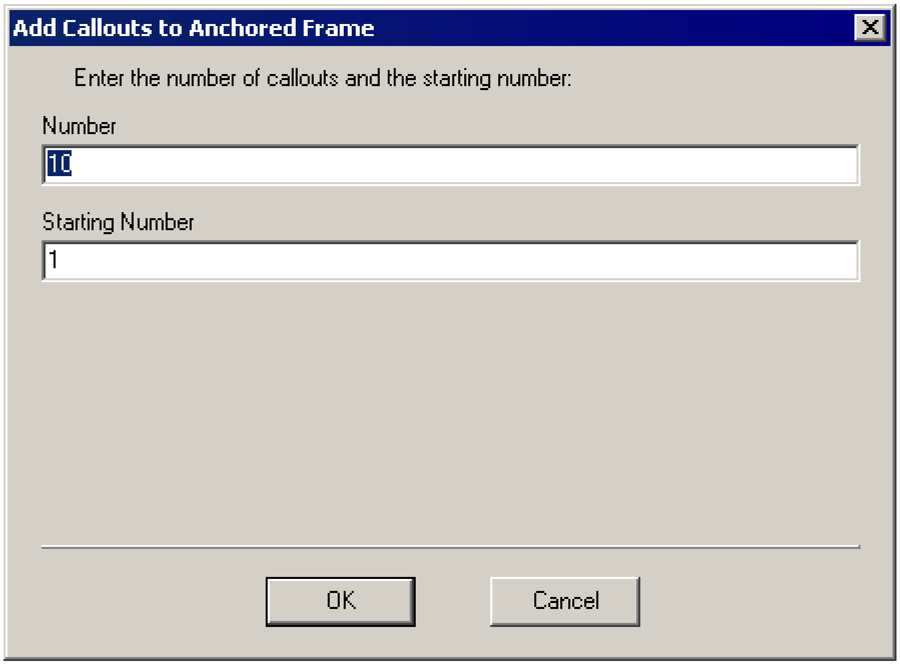


Each callout consists of four objects: two circles for the callout bubble, one text frame for the number, and a line with an arrowhead. There are two circles for each bubble, so you can specify different background and border colors. For example, you could have a bubble with a red border and a yellow background.

One end of the callout line is centered under the bubble. This allows you to drag the arrowhead end of the line to any position and still have the line positioned correctly on the edge of the bubble.

Although these callouts look simple, you can imagine how difficult it would be to make each one on an illustration. Even if you made one, and then copied and pasted it to make the others, you would still have to edit each one to have the correct number.

Our script will simplify the process. Here is the interface to the script.



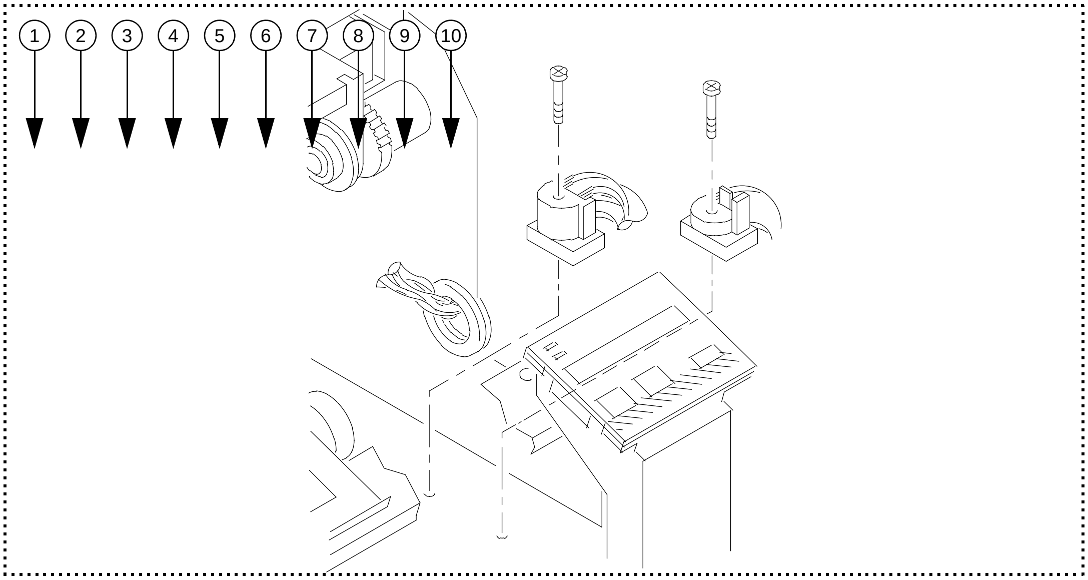
Add Callouts to Anchored Frame

Enter the number of callouts and the starting number:

Number: 10

Starting Number: 1

You will simply enter the number of callouts that you want and the starting number of the first callout. Here is the result.



The callouts are individually grouped so you can drag each one into position on the illustration. After you get the bubble in position, you choose Graphics > Ungroup, and then drag the line so it points to the correct object. After positioning, you can choose Graphics > Group to regroup the bubble with the line.

Defining the Tasks

When developing a script like this, it can be difficult to decide where to start. Generally, it is best to break the script down into small steps and work on each one individually. Once you can determine the individual steps, you can code each one. When each piece of code is working correctly, you can put them together to solve the larger problem. Here is a rough outline of the things that we will have to do in this script.

- Test for an active document.
- Test for a selected anchored frame. We will only want to make callouts inside of an anchored frame.
- Get information from a dialog box.
- Make various graphic objects, such as ellipses (circles), text frames, lines, and groups.
- Make and use a paragraph format.
- Add and space multiple objects with a loop.

The outline may not be complete, but it gives us something to work with. We will start with the first point, but you don't necessarily have to work in order.

Test for an Active Document

It may seem evident that you need an active document in order to run the script, but it may not be so evident to your users. A big part of programming is preparing for unexpected circumstances when the script is run. You might think that someone would never try to run the script without a document open, but sooner or later (probably sooner), someone will.

Here is a way to test for an active document. You will use this (or a similar) test in many scripts, so you can often copy and paste it from another script.

Code Listing 12-1

```
// Test for an active document.
If ActiveDoc = 0
  // Display a warning and exit the script.
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  // Set a variable for the active document.
  Set vCurrentDoc = ActiveDoc;
EndIf
```

We set a variable, **vCurrentDoc**, for the active document. This is not absolutely necessary in this script because it always works on the active document; but it is good practice because it makes the code more *portable*. Code that is portable is more likely to be reused in other scripts. In this case, **vCurrentDoc** will represent the active document, but in other scripts, it may represent a document that is *not* active; for example, an invisible document.

Test for a Selected Anchored Frame

Since we want to put the callouts into an anchored frame, we need to make sure one is selected. A document has a **FirstSelectedGraphicInDoc** property that returns the first selected object in the document. We can use this to test for a selected anchored frame.

Code Listing 12-2

```
// Test for a selected anchored frame.
If vCurrentDoc.FirstSelectedGraphicInDoc.ObjectName not= 'AFrame'
  MsgBox 'Please select an anchored frame.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the anchored frame.
  Set vFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
EndIf
```

We don’t want to test for any selected graphic; we want to make sure that the graphic is an anchored frame. That is why we test the **ObjectName** of the graphic. If an **AFrame** is selected, then we set a variable for it (**vAFrame**) for later use in the script.

Making Graphics

The next task we will tackle is making graphic objects inside of the anchored frame. We are skipping the dialog box step because we have to know how to create objects before we prompt the user for details about the objects.

In this section, we will be working with a selected anchored frame in the active document. Our script up to this point will help us prevent errors because we have the tests for an active document and selected anchored frame in place. Make sure you have an anchored frame selected in a document, and add this to the script.

Code Listing 12-3

```
// Make an ellipse inside the anchored frame.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
```

All new objects that you create will have this same basic syntax: you use the **New** command with the appropriate object name, with the **ParentObject** parameter indicating the selected anchored frame. You can then use the **NewVar** variable to change properties on the new object. Here is code revised to change some of the properties on the **Ellipse**.

Code Listing 12-4

```
// Make an ellipse inside the anchored frame.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = 6pt;
Set vEllipse.LocX = 6pt;
Set vEllipse.Height = 12pt;
Set vEllipse.Width = 12pt;
Set vEllipse.BorderWidth = .5pt;
```

Our specification actually calls for two circles; one to supply the background, the other to supply the border. Let's add both circles, using a yellow background and a red border.

Code Listing 12-5

```
// Get the color object for Yellow.
Get Object Type(Color) Name('Yellow') DocObject(vCurrentDoc)
  NewVar(vColor);

// Make an ellipse for the bubble background.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = 6pt;
Set vEllipse.LocX = 6pt;
Set vEllipse.Height = 12pt;
Set vEllipse.Width = 12pt;
Set vEllipse.Fill = FillBlack; // Solid fill
Set vEllipse.Pen = FillClear;  // Clear border
Set vEllipse.Color = vColor;

// Get the color object for Red.
Get Object Type(Color) Name('Red') DocObject(vCurrentDoc)
  NewVar(vColor);

// Make an ellipse for the bubble border.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = 6pt;
Set vEllipse.LocX = 6pt;
Set vEllipse.Height = 12pt;
Set vEllipse.Width = 12pt;
Set vEllipse.Fill = FillClear; // Clear fill
Set vEllipse.Pen = FillBlack;  // Solid border
Set vEllipse.BorderWidth = .5pt;
Set vEllipse.Color = vColor;
```

Now we need a text frame to hold the callout text. Initially, we will use the same dimensions as the bubbles.

Code Listing 12-6

```
// Make a text frame for the callout text.
New TextFrame NewVar(vTextFrame) ParentObject(vFrame);
Set vTextFrame.LocY = 6pt;
Set vTextFrame.LocX = 6pt;
Set vTextFrame.Height = 12pt;
Set vTextFrame.Width = 12pt;
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) '1';
```

Run the code to see all of the objects so far. Refresh the screen (Ctrl+L), if necessary. It is evident that the text will have to be positioned differently, but we can do that later. Right now, let's add the callout line. When creating a line, you position it by making a **PointList**. A **PointList** is a set of X and Y coordinates for each of the endpoints on the line. Let's assume that we want our line to be 42 points long. We want one endpoint to be centered on the callout bubble. To get the center of the bubble, you first divide the bubble dimensions by 2 (12 / 2 = 6). You then add this number to the LocY and LocX values of the bubble (6 + 6 = 12), which means that the X and Y coordinates for the first point of the line is 12, 12. Because the line is vertical, the X value will not change for the second point, but we want to add the line length to the first Y value (42 + 12 = 54). This makes the second point of the line at 12, 54. Here is the code to add the line.

Code Listing 12-7

```
// Make the point list for the line.
New PointList NewVar(vPointList) X(12pt) Y(12pt) X(12pt) Y(54pt);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointList);
Set vLine.BorderWidth = .5pt;
Set vLine.HeadArrow = 1;
```

Run the script and you will see the line. In the finished script, we will add the line before the bubble, so that it will be behind it; we have added it here on top so you can see how the end of the line centers on the bubble. We will also group all of the bubble components together and then group the line with the bubble group.

Before going further, let's examine what we have so far. For each object, we are basically setting its position and the appropriate appearance properties. Up to now, we have hardcoded the **LocX** and **LocY** values, but this will be a problem when we expand the script to add more than one callout. As the callouts are added across the frame, the value of **LocX** will get bigger. Because this value will change, it should be put in a variable. It is also possible that **LocY** will have to change. If there are more objects that can fit in one row, we will want to start a second row at the left of the frame. So we should put **LocY** in a variable, also.

You should also notice that we are setting quite a few appearance properties for each object as we add it to the frame. It is usually easier to apply all of the properties to an object by using a **PropertyList**. This is an advantage when you want to modify or add properties for an object, because you do not have to alter the code that adds the object. You can isolate your property settings in a separate part of the script. You will see that this makes your script more organized and easier to troubleshoot.

Working With Property Lists

You may have noticed that we didn't have this section as part of our original task list. This illustrates what usually happens when you develop scripts: you will discover challenges that may alter or add to your task list. As your script grows, you will find ways to organize the code to make it easier to organize and follow. You should not only think about writing the script, but also *maintaining* it. When you have to make changes to the script, especially weeks or months after it has been written, you will appreciate a well-organized script.

We will make a subroutine to set up appearance properties for each item. Before doing this, we will briefly discuss **PropertyLists**. To create a **PropertyList**, you use the **New PropertyList** command and assign a variable name to the list.

```
New PropertyList NewVar(vBubbleFillProps);
```

You can add properties as you create the list.

```
New PropertyList NewVar(vBubbleFillProps) BorderWidth(.5pt)
  Fill(FillBlack) Pen(FillClear);
```

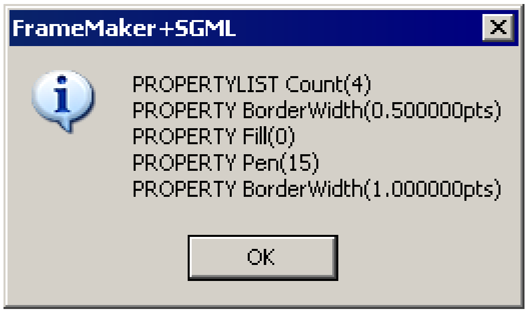
You can also add properties to a PropertyList after you create it.

```
New PropertyList NewVar(vBubbleFillProps);
Add Property To(vBubbleFillProps) BorderWidth(.5pt)
  Fill(FillBlack) Pen(FillClear);
```

It is possible to add the same property to list more than once; for example, in the code below the **BorderWidth** property is added twice.

Code Listing 12-8

```
New PropertyList NewVar(vBubbleFillProps);
Add Property To(vBubbleFillProps) BorderWidth(.5pt)
  Fill(FillBlack) Pen(FillClear) BorderWidth(1pt);
Display vBubbleFillProps;
```



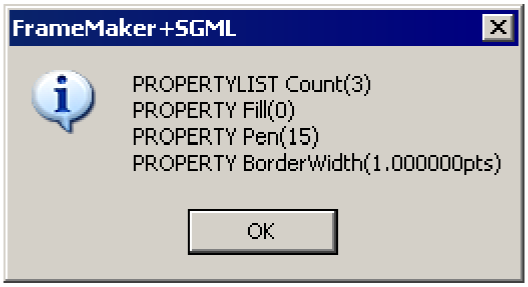
```
PROPERTYLIST Count(4)
PROPERTY BorderWidth(0.500000pts)
PROPERTY Fill(0)
PROPERTY Pen(15)
PROPERTY BorderWidth(1.000000pts)
```

In this case, the first instance of the property will be used when it is applied to an object.

You can also remove properties from a **PropertyList** by using the **Remove Property** command. If the property you are removing appears more than once in the list, the first occurrence will be removed.

Code Listing 12-9

```
New PropertyList NewVar(vBubbleFillProps);
Add Property To(vBubbleFillProps) BorderWidth(.5pt)
  Fill(FillBlack) Pen(FillClear) BorderWidth(1pt);
Remove Property(BorderWidth) From(vBubbleFillProps);
Display vBubbleFillProps;
```



```
PROPERTYLIST Count(3)
PROPERTY Fill(0)
PROPERTY Pen(15)
PROPERTY BorderWidth(1.000000pts)
```

To apply a **PropertyList** to an object, you use the **Set** command.

```
Set vEllipse.Properties = vBubbleFillProps;
```

For our script, we will make a subroutine called **SetProperties** to set up **PropertyLists** for the callout objects. When you make a subroutine in a script, it has to come after any commands that are not inside a subroutine. Any code that is not inside a subroutine as part of the *body* of the script. Subroutines can be in any order in a script, but they have to appear after the body of the script. Here is the revised script with the **SetProperties** subroutine.

Code Listing 12-10

```
// Test for an active document.
If ActiveDoc = 0
  // Display a warning and exit the script.
  MsgBox 'There is no active document.    ';
  LeaveSub;
Else
  // Set a variable for the active document.
  Set vCurrentDoc = ActiveDoc;
EndIf

// Test for a selected anchored frame.
If vCurrentDoc.FirstSelectedGraphicInDoc.ObjectName not= 'AFrame'
  MsgBox 'Please select an anchored frame.    ';
  LeaveSub; // Exit the script.
Else
  // Set a variable for the anchored frame.
  Set vFrame = vCurrentDoc.FirstSelectedGraphicInDoc;
EndIf

// Run the subroutine to set up property lists.
Run SetupProperties;

// Set the location for the first callout.
Set vLocY = 6pt;
Set vLocX = 6pt;

// Make the point list for the line.
New PointList NewVar(vPointList) X(12pt) Y(12pt) X(12pt) Y(54pt);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointList);
// Apply the property list.
Set vLine.Properties = vLineProps;

// Make an ellipse for the bubble background.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleFillProps;
```

```
// Make an ellipse for the bubble border.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleBorderProps;

// Make a text frame for the callout text.
New TextFrame NewVar(vTextFrame) ParentObject(vFrame);
Set vTextFrame.LocY = vLocY;
Set vTextFrame.LocX = vLocX;
// Apply the property list.
Set vTextFrame.Properties = vTextFrameProps;
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) '1';

Sub SetupProperties
//
// Make a property list for the callout line.
New PropertyList NewVar(vLineProps) BorderWidth(.5pt)
  HeadArrow(1) ArrowTipAngle(16) ArrowBaseAngle(90) ArrowLength(12)
  ArrowType(3) Runaround(1);

// Get the color object for Yellow.
Get Object Type(Color) Name('Yellow') DocObject(vCurrentDoc)
  NewVar(vColor);
// Make a property list for the bubble background.
New PropertyList NewVar(vBubbleFillProps)
  Fill(FillBlack) Pen(FillClear) Color(vColor)
  Width(12pt) Height(12pt);

// Get the color object for Red.
Get Object Type(Color) Name('Red') DocObject(vCurrentDoc)
  NewVar(vColor);
// Make a property list for the bubble border.
New PropertyList NewVar(vBubbleBorderProps) BorderWidth(.5pt)
  Fill(FillClear) Pen(FillBlack) Color(vColor)
  Width(12pt) Height(12pt);

// Text frame properties for numbers.
New PropertyList NewVar(vTextFrameProps) Fill(FillClear)
  Width(12pt) Height(12pt);
//
EndSub
```

This script works the same as the original one, but it is more organized. You may notice that the revised script is longer—it has more lines—than the original, but this will usually be the case when you use subroutines.

There are still some issues with this script that will make maintainence difficult. What if the user decides to change the size of the callout bubbles? What if the length of the callout line needs to be changed? In our current script, we have a lot of *hardcoded* values. When something is hardcoded, it means that it uses real values instead of variables. The value of the height and width (**12pt**) is used in at least four places in the script. If we want to change the size of the callout bubbles, we will have to hunt each value down and change it. It would be better to use a variable for each of these values. Then we can set the value of the variable in one place.

We will solve this problem by making another subroutine in the script. The subroutine will set variables for each of the property values used in the SetupProperties subroutine. This will allow the properties for the callout objects to be easily modified in one place in the script. We will use comments for each of the properties to indicate what each variable represents. Here is the new **DefineProperties** subroutine, followed by the revised **SetupProperties** subroutine.

Code Listing 12-11

```
Sub DefineProperties
//
// ***** Set callout properties. *****
//
// Diameter of callout bubble.
Set vCircleSize = 12pt;
// Line width of callout line and bubble border.
Set vLineWidth = .5pt;
// Arrow head? 1 or yes, 0 for no.
Set vHeadArrow = 1;
// Arrow tip angle for the line.
Set vArrowTipAngle = 16;
// Arrow base angle for the line.
Set vArrowBaseAngle = 90;
// Arrow length.
Set vArrowLength = 12pt;
// Initial callout line length.
Set vLineLength = 42pt;

// Circle background color.
Set vCircleColor = 'Yellow';
// Circle border color.
Set vCircleBorderColor = 'Red';
// Callout line color.
Set vLineColor = 'Red';
//
EndSub

Sub SetupProperties
//
// Get the color object for the bubble background.
Get Object Type(Color) Name(vLineColor) DocObject(vCurrentDoc)
  NewVar(vColor);
// If the color does not exist, use white.
If vColor = 0
  Get Object Type(Color) Name('Black') DocObject(vCurrentDoc)
    NewVar(vColor);
EndIf
// Make a property list for the callout line.
New PropertyList NewVar(vLineProps) BorderWidth(vBorderWidth)
  HeadArrow(vHeadArrow) ArrowTipAngle(vArrowTipAngle)
  ArrowBaseAngle(vArrowBaseAngle) ArrowLength(vArrowLength)
  Color(vColor) ArrowType(3) Runaround(1);
```

```
// Get the color object for the bubble background.
Get Object Type(Color) Name(vCircleColor) DocObject(vCurrentDoc)
  NewVar(vColor);
// If the color does not exist, use white.
If vColor = 0
  Get Object Type(Color) Name('White') DocObject(vCurrentDoc)
    NewVar(vColor);
EndIf
// Make a property list for the bubble background.
New PropertyList NewVar(vBubbleFillProps)
  Fill(FillBlack) Pen(FillClear) Color(vColor)
  Width(vCircleSize) Height(vCircleSize);

// Get the color object for the bubble border.
Get Object Type(Color) Name(vCircleBorderColor)
DocObject(vCurrentDoc)
  NewVar(vColor);
// If the color does not exist, use black.
If vColor = 0
  Get Object Type(Color) Name('Black') DocObject(vCurrentDoc)
    NewVar(vColor);
EndIf
// Make a property list for the bubble border.
New PropertyList NewVar(vBubbleBorderProps)
BorderWidth(vBorderWidth)
  Fill(FillClear) Pen(FillBlack) Color(vColor)
  Width(vCircleSize) Height(vCircleSize);

// Text frame properties for numbers.
New PropertyList NewVar(vTextFrameProps) Fill(FillClear)
  Width(vCircleSize) Height(vCircleSize);
//
EndSub
```

You should notice a couple of things about the revised **SetupProperties** subroutine. First, we still have a few hardcoded values on the **PropertyLists**. There are some properties whose values are not likely to change; for example, the background on the top bubble will always be clear (**FillClear**). It is best not to crowd the **DefineProperties** subroutine with values that will rarely, if ever, change.

Second, because we allow the user to specify whatever colors they want, we need to make sure that the colors exist in the document to prevent errors. If any of the colors don't exist, then we use default colors that we know exist in every document.

We will also make use of some of these variables in the body of the script. Here is the **New PointList** command, using math formulas and variables to calculate the points for the callout line.

Code Listing 12-12

```
// Set the location for the first callout.
Set vLocY = 6pt;
Set vLocX = 6pt;
```

```
// Make the point list for the line.
// Make a variable for half the bubble size (radius).
Set vRadius = vCircleSize / 2;
New PointList NewVar(vPointList)
  X(vLocX+vRadius) Y(vLocY+vRadius)
  X(vLocX+vRadius) Y(vLocY+vRadius+vLineLength);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointList);
// Apply the property list.
Set vLine.Properties = vLineProps;
```

It should be evident that we are trying to generalize the code as much as possible. Try changing the value on the **vCircleSize** variable in the **DefineProperties** subroutine and run the script.

Formatting Callout Text

It is important that the callout text be formatted correctly so that it looks right inside the bubble. We want to use a paragraph format for the callouts so that the user will be able to globally alter the appearance after the callouts are added. We will add the paragraph format name to the **DefineProperties** subroutine so it can be changed, if necessary. Here is the code.

Code Listing 12-13

```
// Paragraph format name for callouts.
Set vCalloutPgfFmt = 'Callout';
```

It is possible that this paragraph format will not exist when the script is run. This will cause the Body paragraph format to be used by default, and make it difficult to change the bubble text. Because of this, we will add a subroutine that will make the paragraph format, using a couple of default values. Here are the two default values to add to the **DefineProperties** subroutine.

Code Listing 12-14

```
// Font for callout bubble.
Set vFont = 'Arial';
// Font size for callout bubble.
Set vFontSize = 7pt;
```

It is important to note that if the **vCalloutPgfFmt** does exist in the document, the **vFont** and **vFontSize** values will not be used by the script. Here is the **MakeCalloutPgfFormat** subroutine.

Code Listing 12-15

```
Sub MakeCalloutPgfFormat
//
// See if the paragraph format object exists in the document.
Get Object Type(PgfFmt) Name(vCalloutPgfFmt) DocObject(vCurrentDoc)
  NewVar(vCalloutFmt);
If vCalloutFmt.ObjectName not= 'PgfFmt'
  New PgfFmt DocObject(vCurrentDoc) Name(vCalloutPgfFmt)
    NewVar(vCalloutFmt);
  // Make a property list.
  New PropertyList NewVar(vCalloutProps) FontSize(vFontSize)
    PgfAlignment(PgfCenter) Placement(0);
  // Add the font to the property list if it exists.
  Find Member(vFont) InList(FontFamilyNames) ReturnPos(vPos);
  If vPos > 0
    Add Property To(vCalloutProps) FontFamily(vPos-1);
  EndIf
  Set vCalloutFmt.Properties = vCalloutProps;
EndIf
//
EndSub
```

Now, we must alter the body of the script to call the **MakeCalloutPgfFormat** subroutine, and then use it when adding the callout text. Here are the revised sections of the body of the script. Copy the new sections into your script and give it a try.

Code Listing 12-16

```
...
// Run the subroutine to define the properties.
Run DefineProperties;
// Run the subroutine to set up property lists.
Run SetupProperties;
// Run the subroutine to make the callout paragraph format.
Run MakeCalloutPgfFormat;
...
// Make a text frame for the callout text.
New TextFrame NewVar(vTextFrame) ParentObject(vFrame);
Set vTextFrame.LocY = vLocY;
Set vTextFrame.LocX = vLocX;
// Apply the property list.
Set vTextFrame.Properties = vTextFrameProps;
// Apply the callout paragraph format to the text frame.
Set vTextFrame.FirstPgf.Properties = vCalloutFmt.Properties;
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) '1';
...
```

You should see the Callout paragraph format created (if it doesn't already exist) and used for the callout text. The text will be too high in the bubble, so we will adjust this by setting up another variable in the **DefineProperties** subroutine. We will call this variable **vTextFrameTopOffset**. You will have to experiment to get the value just right, but let's start with **3pt**.

Code Listing 12-17

```
// Top offset of text frame from bubble.
Set vTextFrameTopOffset = 3pt;
```

Now, add this to the body of the script, in the section where the text frame is added.

Code Listing 12-18

```
Set vTextFrame.LocY = vLocY + vTextFrameTopOffset;
```

Run the script and you will see that the top alignment of the text is better. However, the bottom of the text frame hangs below the bottom of the bubble. To correct his, we need to subtract **vTextFrameOffset** from the height of the text frame. This change will be made in the **SetupProperties** subroutine.

Code Listing 12-19

```
// Text frame properties for numbers.
New PropertyList NewVar(vTextFrameProps) Fill(FillClear)
  Width(vCircleSize) Height(vCircleSize-vTextFrameTopOffset);
```

Make this change, and try the script.

Grouping the Objects

We have one more thing to do to the callout before figuring out how to add more than one. We need to make two groups, one for the bubble objects, and the second to group the line with the bubble group. A **Group** is an invisible graphic object that contains other objects. Using **Groups** will keep related objects together and make it easier to position them on the illustration. To make a **Group**, you use the **New Group** command and specify its **ParentObject** property. To add an object to a **Group**, you set its **GroupParent** property to the **Group** object. As you will see, you can add one **Group** to another **Group**. Here is the group code in context.

Code Listing 12-20

```
// Set the location for the first callout.
Set vLocY = 6pt;
Set vLocX = 6pt;

// Make groups for the objects.
New Group ParentObject(vFrame) NewVar(vBubbleGroup);
New Group ParentObject(vFrame) NewVar(vCalloutGroup);

// Make the point list for the line.
// Make a variable for half the bubble size (radius).
Set vRadius = vCircleSize / 2;
New PointList NewVar(vPointList)
  X(vLocX+vRadius) Y(vLocY+vRadius)
  X(vLocX+vRadius) Y(vLocY+vRadius+vLineLength);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointList);
// Apply the property list.
Set vLine.Properties = vLineProps;
// Add the line to the overall group.
Set vLine.GroupParent = vCalloutGroup;
```

```
// Make an ellipse for the bubble background.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleFillProps;
// Add the object to the bubble group.
Set vEllipse.GroupParent = vBubbleGroup;

// Make an ellipse for the bubble border.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleBorderProps;
// Add the object to the bubble group.
Set vEllipse.GroupParent = vBubbleGroup;

// Make a text frame for the callout text.
New TextFrame NewVar(vTextFrame) ParentObject(vFrame);
Set vTextFrame.LocY = vLocY + vTextFrameTopOffset;
Set vTextFrame.LocX = vLocX;
// Apply the property list.
Set vTextFrame.Properties = vTextFrameProps;
// Add the object to the bubble group.
Set vTextFrame.GroupParent = vBubbleGroup;
// Apply the callout paragraph format to the text frame.
Set vTextFrame.FirstPgf.Properties = vCalloutFmt.Properties;
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) '1';

// Add the bubble group to the overall callout group.
Set vBubbleGroup.GroupParent = vCalloutGroup;
```

Try the script and select the resulting callout. You can see that it selects as one unit because it is grouped. Right-click on it and choose Ungroup. Now you can drag the end of the line into position. When you are done, you can right-click and choose Group to regroup the line with the bubble.

Adding Multiple Callouts

Now that we can add a single callout, we can turn our attention to adding more than one. We will use a loop to add multiple callouts to the anchored frame. To make the loop less complicated, let's make a subroutine for the code that adds a single callout. Then we can call this subroutine as many times as necessary. Here is the **MakeCallout** subroutine.

Code Listing 12-21

```
Sub MakeCallout
//
// Make groups for the objects.
New Group ParentObject(vFrame) NewVar(vBubbleGroup);
New Group ParentObject(vFrame) NewVar(vCalloutGroup);
```

```
// Make the point list for the line.
// Make a variable for half the bubble size (radius).
Set vRadius = vCircleSize / 2;
New PointList NewVar(vPointList)
  X(vLocX+vRadius) Y(vLocY+vRadius)
  X(vLocX+vRadius) Y(vLocY+vRadius+vLineLength);
New Line NewVar(vLine) ParentObject(vFrame) Points(vPointList);
// Apply the property list.
Set vLine.Properties = vLineProps;
// Add the line to the overall group.
Set vLine.GroupParent = vCalloutGroup;

// Make an ellipse for the bubble background.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleFillProps;
// Add the object to the bubble group.
Set vEllipse.GroupParent = vBubbleGroup;

// Make an ellipse for the bubble border.
New Ellipse NewVar(vEllipse) ParentObject(vFrame);
Set vEllipse.LocY = vLocY;
Set vEllipse.LocX = vLocX;
// Apply the property list.
Set vEllipse.Properties = vBubbleBorderProps;
// Add the object to the bubble group.
Set vEllipse.GroupParent = vBubbleGroup;

// Make a text frame for the callout text.
New TextFrame NewVar(vTextFrame) ParentObject(vFrame);
Set vTextFrame.LocY = vLocY + vTextFrameTopOffset;
Set vTextFrame.LocX = vLocX;
// Apply the property list.
Set vTextFrame.Properties = vTextFrameProps;
// Add the object to the bubble group.
Set vTextFrame.GroupParent = vBubbleGroup;
// Apply the callout paragraph format to the text frame.
Set vTextFrame.FirstPgf.Properties = vCalloutFmt.Properties;
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) '1';

// Add the bubble group to the callout group.
Set vBubbleGroup.GroupParent = vCalloutGroup;
//
EndSub
```

We need to add a variable to the **New Text** command so that the correct number is added to the callout. Right now, there is a hardcoded "1" that will be replaced with the **vText** variable.

Code Listing 12-22

```
// Add text to the text frame.
New Text Object(vTextFrame.FirstPgf) vText;
```

Now we need to construct the call to the subroutine. Here is a simple loop that can go right at the end of the body of the script. Initially, we are setting a **vNumber** variable to 6 to indicate the number of callouts that will be added.

Code Listing 12-23

```
// Set the number of callouts to add.
Set vNumber = 6;
Loop While(n < vNumber) LoopVar(n) Init(0) Incr(1)
  // Run the subroutine that adds the callout.
  Run MakeCallout vText(n+1);
EndLoop
```

If you run the script now, it creates six callouts in the anchored frame, but it puts them on top of each other. We need a mechanism to move each new callout to the right of the previous one. Let's begin by adding a **vBubbleGap** variable that determines how much space will go between each bubble. Add this code to the **DefineProperties** subroutine.

Code Listing 12-24

```
// Gap between callouts.
Set vBubbleGap = 6pt;
```

We can use this variable to initialize the **vLocX** and **vLocY** variables. Replace these lines

```
// Set the location for the callout.
Set vLocY = 6pt;
Set vLocX = 6pt;
```

with

Code Listing 12-25

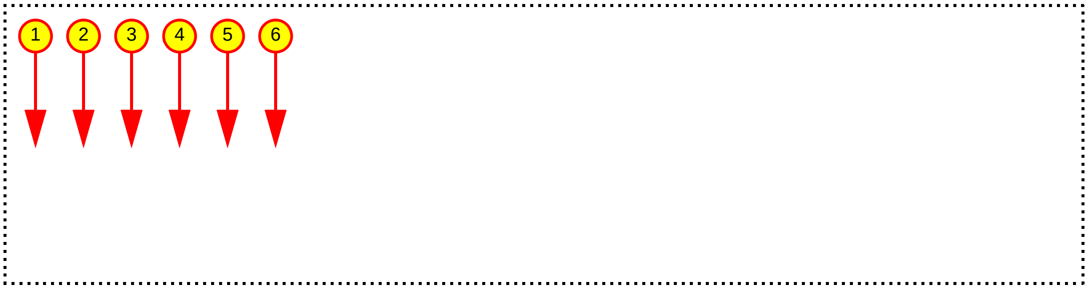
```
// Initialize the X and Y location of the first callout.
New Real NewVar(vLocX) Value(vBubbleGap);
New Real NewVar(vLocY) Value(vBubbleGap);
```

Inside the loop, we will use the **vCircleSize** and **vBubbleGap** variables to determine the **vLocX** value for the next callout.

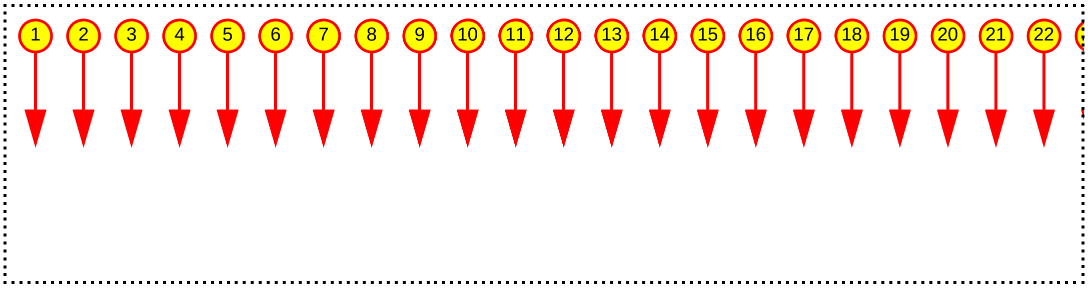
Code Listing 12-26

```
// Set the number of callouts to add.
Set vNumber = 6;
Loop While(n < vNumber) LoopVar(n) Init(0) Incr(1)
  // Run the subroutine that adds the callout.
  Run MakeCallout vText(n+1);
  // Move the x location for the next callout.
  Set vLocX = vLocX + vCircleSize + vBubbleGap;
EndLoop
```

Update your script and give it a try. You should see the callouts move across the anchored frame as they are created.



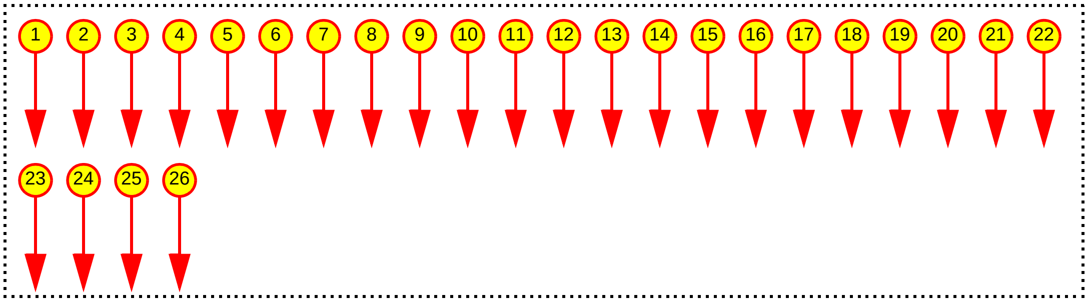
Six callouts should fit in the frame, but try a large number like 26 and see what happens.



You should notice two things. One, all of the callouts are not visible in the frame; and two, it takes a while for the script to draw all of the callouts. Let's start with the first issue. We can test for the location of each new callout to see if it will end up wider than the anchored frame. If it will, we can move it down in the frame, and start back at the left. Let's try it in our loop.

Code Listing 12-27

```
// Set the number of callouts to add.
Set vNumber = 26;
Loop While(n < vNumber) LoopVar(n) Init(0) Incr(1)
  // Run the subroutine that adds the callout.
  Run MakeCallout vText(n+1);
  // Move the x location for the next callout.
  Set vLocX = vLocX + vCircleSize + vBubbleGap;
  // Test the LocX location to see if it is wider than the frame.
  If (vLocX + vCircleSize + vBubbleGap) >= vFrame.Width
    // Move LocY down and LocX to the left to start another row.
    Set vLocY = vLocY + vRadius + vBubbleGap + vLineLength;
    Set vLocX = vBubbleGap;
  EndIf
EndLoop
```



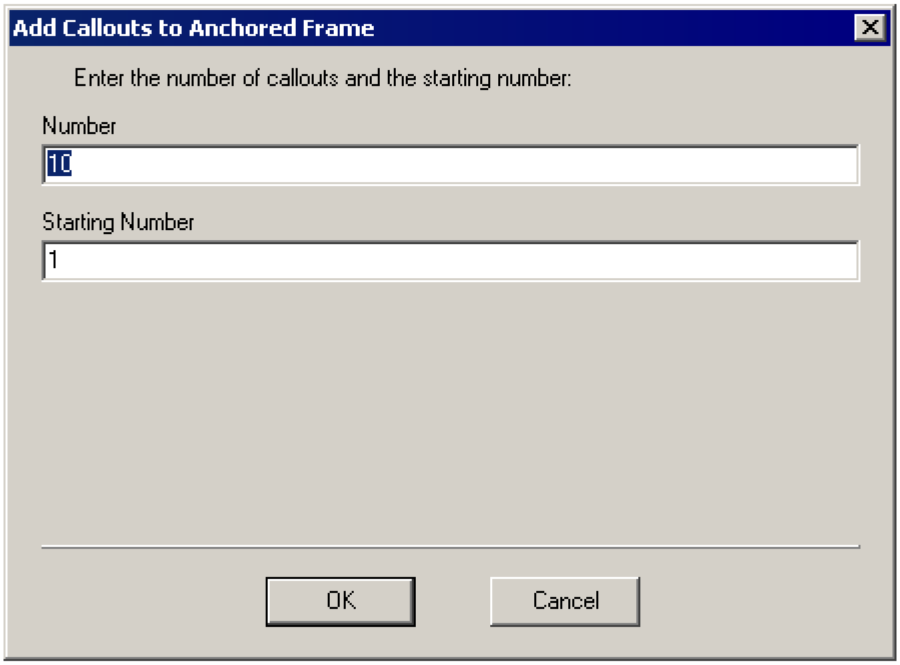
The second problem—the slow speed of the script—is caused by FrameMaker redrawing the screen as each object is drawn. Fortunately, the solution is simple. We need to turn off the **Displaying** property before the loop begins and restore it at the end. Add the following code to your script and give it a try.

Code Listing 12-28

```
// Turn off the document display to speed the script.
Set Displaying = 0;

// Set the number of callouts to add.
Set vNumber = 26;
Loop While(n < vNumber) LoopVar(n) Init(0) Incr(1)
  // Run the subroutine that adds the callout.
  Run MakeCallout vText(n+1);
  // Move the x location for the next callout.
  Set vLocX = vLocX + vCircleSize + vBubbleGap;
  // Test the LocX location to see if it is wider than the frame.
  If (vLocX + vCircleSize + vBubbleGap) >= vFrame.Width
    // Move LocY down and LocX to the left to start another row.
    Set vLocY = vLocY + vRadius + vBubbleGap + vLineLength;
    Set vLocX = vBubbleGap;
  EndIf
EndLoop

// Restore the document display and refresh the screen.
Set Displaying = 1;
Update DocObject(vCurrentDoc) Redisplay;
```

Prompting the User for Information

The mechanics of the script are finished, but we need to add the interface. The purpose of the interface is to gather information from the user. We want to know how many callouts they want and what callout number to start with. Here is the screenshot of the Medit dialogbox that we will use, followed by the code behind it. To test the dialog box, let’s put the code in a separate script for now.



Add Callouts to Anchored Frame

Enter the number of callouts and the starting number:

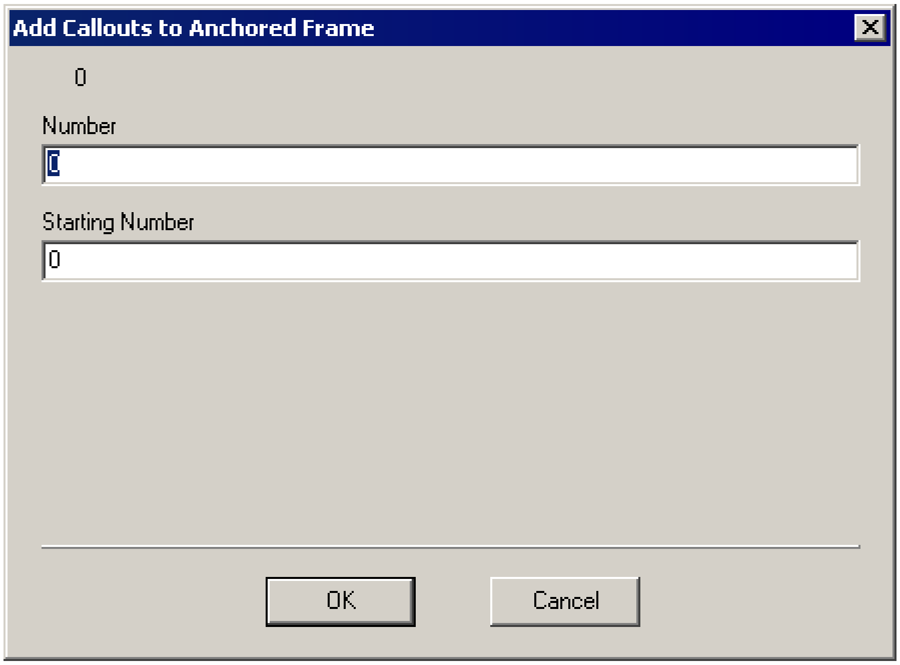
Number: 10

Starting Number: 1

Code Listing 12-29

```
DialogBox Type(MEdit) Title('Add Callouts to Anchored Frame')
  Title2(vLabel)
  Label1('Number') Edit1(vNumber)
  Label2('Starting Number') Edit2(vStartNo)
  Button(vButton);
// If the user cancels the dialog box, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf
```

If you run the above code, you will get the following result.



Add Callouts to Anchored Frame

0

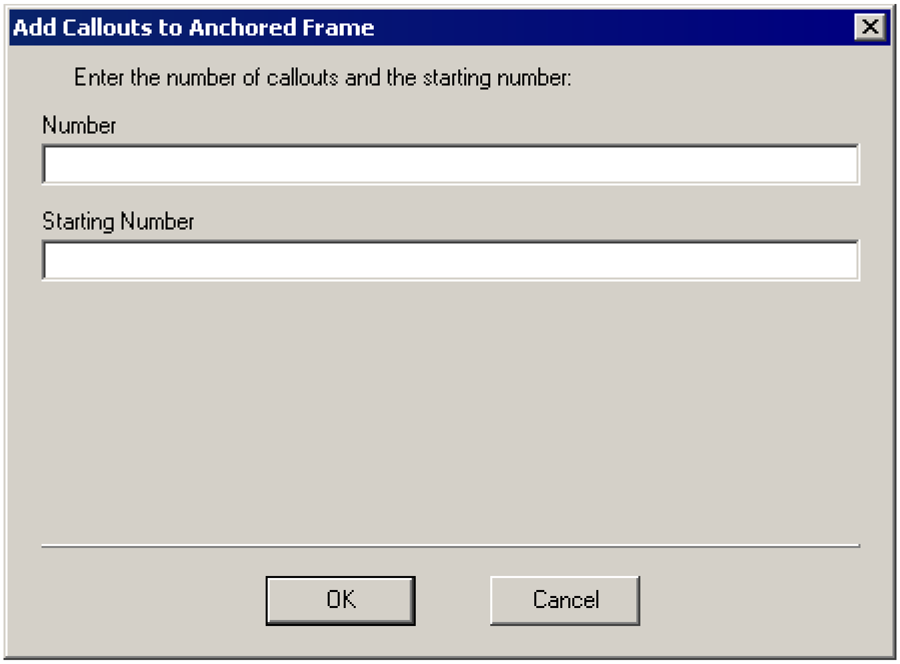
Number: 0

Starting Number: 0

The zeros come from variables in the **DialogBox** command that need to be initialized to default values. Here is the revised code.

Code Listing 12-30

```
// Initialize dialog box variables.
Set vLabel = 'Enter the number of callouts and the starting '+
  'number:';
Set vNumber = '';
Set vStartNo = '';
DialogBox Type(MEdit) Title('Add Callouts to Anchored Frame')
  Title2(vLabel)
  Label1('Number') Edit1(vNumber)
  Label2('Starting Number') Edit2(vStartNo)
  Button(vButton);
// If the user cancels the dialog box, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf
```


Add Callouts to Anchored Frame
Enter the number of callouts and the starting number:
Number
Starting Number
OK
Cancel

You can see that this is better, but it would be more useful to give the user some default values to start with, as in the revised code below.

Code Listing 12-31

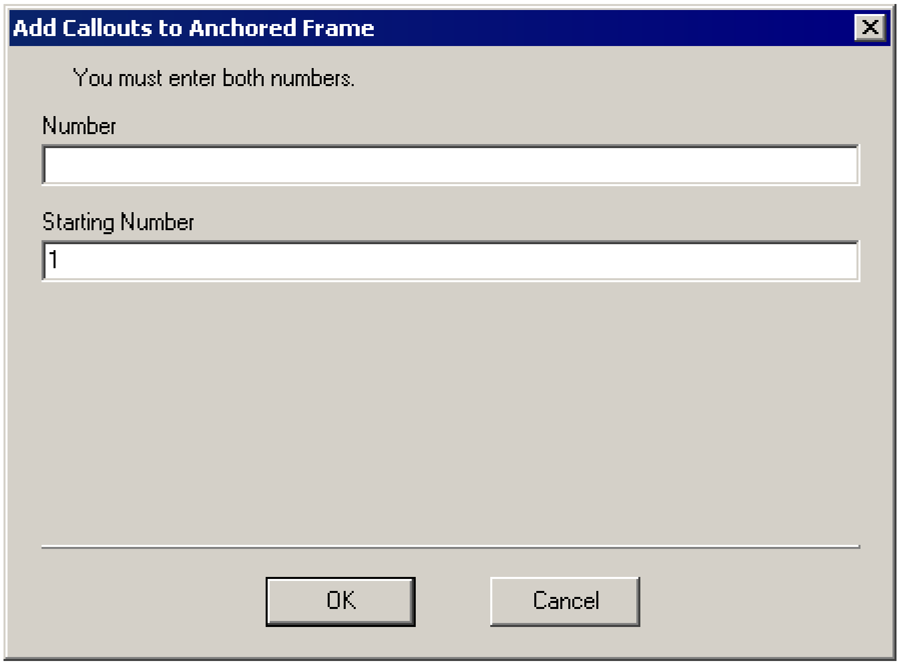
```
// Initialize dialog box variables.
Set vLabel = 'Enter the number of callouts and the starting '+
  'number:';
Set vNumber = '10';
Set vStartNo = '1';
DialogBox Type(MEdit) Title('Add Callouts to Anchored Frame')
  Title2(vLabel)
  Label1('Number') Edit1(vNumber)
  Label2('Starting Number') Edit2(vStartNo)
  Button(vButton);
// If the user cancels the dialog box, exit the script.
If vButton = CancelButton
  LeaveSub;
EndIf
```

If the user clicks OK or hits Enter, we can use the values that they entered. But what if either or both of the text fields are empty? We should give the user a warning and require them to enter data. This can be accomplished with an "endless loop" that will not dismiss the dialog box unless there is data in both fields. This kind of loop is always True and will only exit with the **LeaveLoop** or **LeaveSub** commands.

Code Listing 12-32

```
// Initialize dialog box variables.
Set vLabel = 'Enter the number of callouts and the starting '+
  'number:';
Set vNumber = '10';
Set vStartNo = '1';
Loop While(1)
  DialogBox Type(MEdit) Title('Add Callouts to Anchored Frame')
    Title2(vLabel)
    Label1('Number') Edit1(vNumber)
    Label2('Starting Number') Edit2(vStartNo)
    Button(vButton);
  // If the user cancels the dialog box, exit the script.
  If vButton = CancelButton
    LeaveSub;
  EndIf
  If (vNumber = '') or (vStartNo = '')
    // If either field is blank, put a warning on the dialog box.
    Set vLabel = 'You must enter both numbers.';
  Else
    LeaveLoop;
  EndIf
EndLoop
```

If the user cancels the dialog box, the loop (and the script) will exit with the **LeaveSub** command. If either text fields are blank, the label on the dialog box will change to a warning that the user must enter both numbers. If both numbers have data, then the **LeaveLoop** command will cause the dialog box to exit. Try the script and erase one of the data fields to see the message on the dialog box change.



Where should we put the **DialogBox** command in the script? Ordinarily, you would put it after the tests for an active document and a selected anchored frame. That way, if the dialog box is cancelled, the script exits quickly before doing any work. In this script, however, we will put the **DialogBox** command right after the call to the **DefineProperties** subroutine. We will do this because we want to move the two lines that initialize the values for the dialog box fields to the **DefineProperties** subroutine. This allows us to easily change the default values for the **vNumber** and **vStartNumber** in the same place as the other script settings. Here are the lines to move, with descriptive comments added.

Code Listing 12-33

```
// Dialog box settings.
// The default number of callouts.
Set vNumber = '10';
// The default starting number.
Set vStartNo = '1';
```

The **DialogBox** code now looks like this. Remember to put it after the **Run DefineProperties** command.

Code Listing 12-34

```
// Initialize dialog box variables.
Set vLabel = 'Enter the number of callouts and the starting '+
  'number:';
Loop While(1)
  DialogBox Type(MEdit) Title('Add Callouts to Anchored Frame')
    Title2(vLabel)
    Label1('Number') Edit1(vNumber)
    Label2('Starting Number') Edit2(vStartNo)
    Button(vButton);
  // If the user cancels the dialog box, exit the script.
  If vButton = CancelButton
    LeaveSub;
  EndIf
  If (vNumber = '') or (vStartNo = '')
    // If either field is blank, put a warning on the dialog box.
    Set vLabel = 'You must enter both numbers.';
  Else
    LeaveLoop;
  EndIf
EndLoop
```

There are a couple of loose ends to attend to before the script is finished. The first is to convert the **DialogBox** values from strings to integers. You can do this right after the EndLoop statement that contains the **DialogBox** command.

Code Listing 12-35

```
// Convert dialog box strings to integers.
New Integer NewVar(vNumber) Value(vNumber);
New Integer NewVar(vStartNo) Value(vStartNo);
```

Don’t forget to remove any hardcoded values that were in the loop that calls the **MakeCallout** subroutine, such as the following lines.

```
// Set the number of callouts to add.
Set vNumber = 26;
```

You should remove this because **vNumber** is now supplied by the dialog box. In addition, the first callout should begin with the starting number that is supplied by the **vStartNo** variable. This accomplished by adding **vStartNo** to the value of **n** in the call to the **MakeCallout** subroutine. Here is the complete loop.

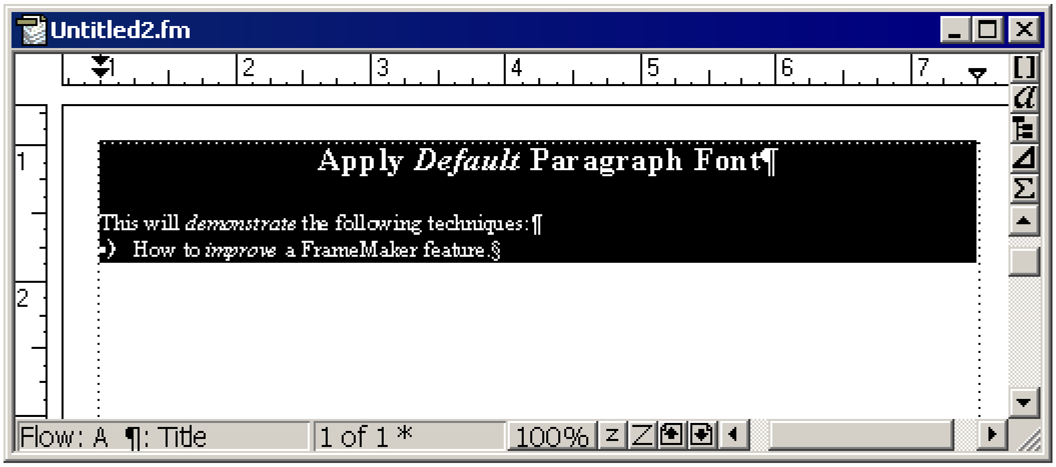
Code Listing 12-36

```
Loop While(n < vNumber) LoopVar(n) Init(0) Incr(1)
  // Run the subroutine that adds the callout.
  Run MakeCallout vText(n+vStartNo);
  // Move the x location for the next callout.
  Set vLocX = vLocX + vCircleSize + vBubbleGap;
  // Test the LocX location to see if it is wider than the frame.
  If (vLocX + vCircleSize + vBubbleGap) >= vFrame.Width
    // Move LocY down and LocX to the left to start another row.
    Set vLocY = vLocY + vRadius + vBubbleGap + vLineLength;
    Set vLocX = vBubbleGap;
  EndIf
EndLoop
```

13 Applying the Default Paragraph Font

This tutorial will demonstrate how to apply the Default Paragraph Font to selected text. To demonstrate why this will be useful, follow these steps.

1. Make a new FrameMaker document.
2. Enter three short paragraphs of text.
3. Apply the Title paragraph format to the first paragraph, Body format to the second paragraph, and Bulleted format to the third.
4. Apply the Emphasis character format to at least one word in each paragraph.
5. Select all three paragraphs at once and apply the Default Paragraph Font.

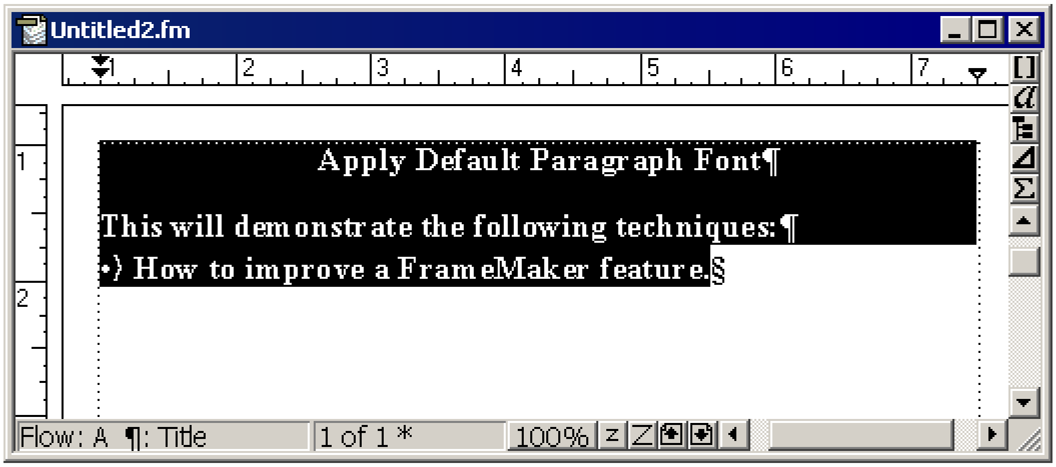


Apply *Default* Paragraph Font¶

This will *demonstrate* the following techniques:¶

•) How to *improve* a FrameMaker feature.§

You can see that FrameMaker applies the Default Paragraph Font of the first paragraph to all of the selected paragraphs! Our script will apply the correct Default Paragraph Font from each paragraph to the selected text.



Apply Default Paragraph Font¶

This will demonstrate the following techniques:¶

•) How to improve a FrameMaker feature.§

ApplyDefaultParaFont.fsl will illustrate the following FrameScript concepts:

- Understanding the **TextSelection** property of a document.
- Applying properties to text.
- Understanding **TextRange** and **TextLoc** objects.
- Using pseudocode to develop script strategies.
- Finding the distinct paragraphs in a text selection.
- Passing parameters to a subroutine.

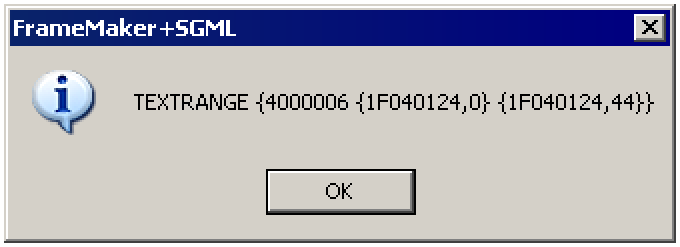
Working with a Single Paragraph

Let's start by applying the Default Paragraph Font to a text selection in a single paragraph. Make a new FrameMaker document and enter some text in the first paragraph. Apply the Emphasis character format to at least one word in the paragraph, and leave the text selected.

Since we want to work with the selected text, we use the document's **TextSelection** property. This property indicates the current insertion point or the text selection of the document. Run the following code to see the **TextSelection**.

Code Listing 13-1

```
Display TextSelection;
```

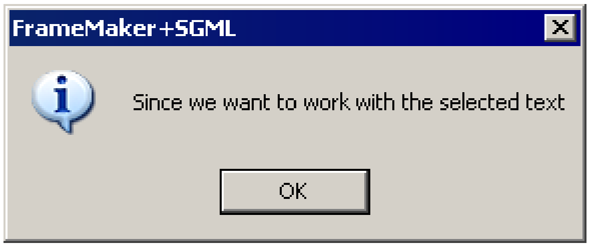


TEXTRANGE {4000006 {1F040124,0} {1F040124,44}}

As you can see from the dialog box, the **TextSelection** is simply a **TextRange** or a range of text. You can see the text of the **TextRange** by running this code.

Code Listing 13-2

```
Display TextSelection.Text;
```

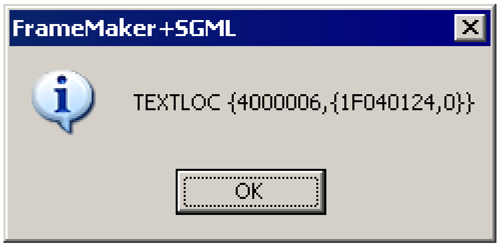


Since we want to work with the selected text

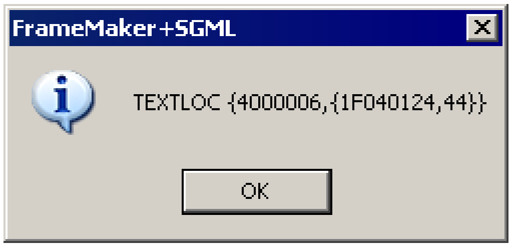
A **TextRange** is bordered on each end by a **TextLoc** (text location) object. To see the **TextLoc** portions of the TextSelection, use this code.

Code Listing 13-3

```
Display TextSelection.Begin;
Display TextSelection.End;
```



TEXTLOC {4000006,{1F040124,0}}



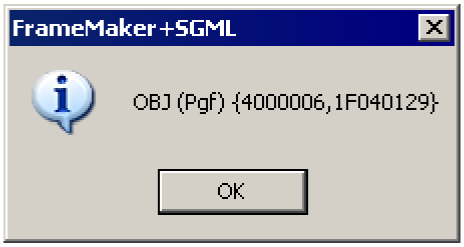
TEXTLOC {4000006,{1F040124,44}}

When you have an insertion point in a document, but no text is selected, both **TextLoc** objects are the same. Click an insertion point in the document and rerun the previous code. If you look carefully at the numbers, you will see that both **TextLoc** objects are the same.

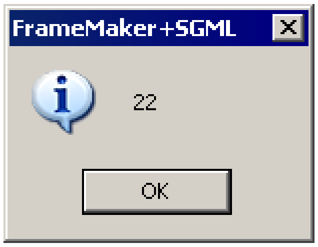
Here is a breakdown of the seemingly obscure numbers that you see when you display a **TextLoc**. The first number is the **Unique Id** of the document containing the **TextLoc**. The second number is the **Unique Id** of the text object containing the **TextLoc**. This will either be a **Pgf** (paragraph) or **TextLine** (text line) object. The third number is the **Offset** of the **TextLoc** from the beginning of the **Pgf** or **TextLine**. Click in the paragraph and run the following code.

Code Listing 13-4

```
Display TextSelection.Begin.Object;
Display TextSelection.Begin.Offset;
```



OBJ (Pgf) {4000006,1F040129}



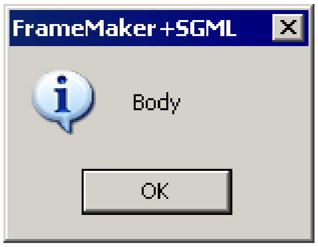
22

You will see the **Pgf** object that contains the insertion point and the offset of the cursor from the beginning of the paragraph. Getting the **Pgf** object in the first line is the key to applying the Default Paragraph Font to a selected text.

What is the Default Paragraph Font? There is no special property called “Default Paragraph Font”; we use initial caps because that is how it appears in the FrameMaker interface. It is simply the default font properties of the paragraph format that is applied to the paragraph. To get the default paragraph font properties of a **Pgf**, we need to the **PgfFmt** (paragraph format) object that is applied to the paragraph. Here is the syntax for getting the name of the **PgfFmt**.

Code Listing 13-5

```
Display TextSelection.Begin.Object.Name;
```



Body

Once we have the name, we can use the **Get Object** command to get the **PgfFmt** and its properties.

Code Listing 13-6

```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
```

Now that we have the **PgfFmt** object, we can apply its properties to the TextSelection. Select the emphasized text in the paragraph and run this code.

Code Listing 13-7

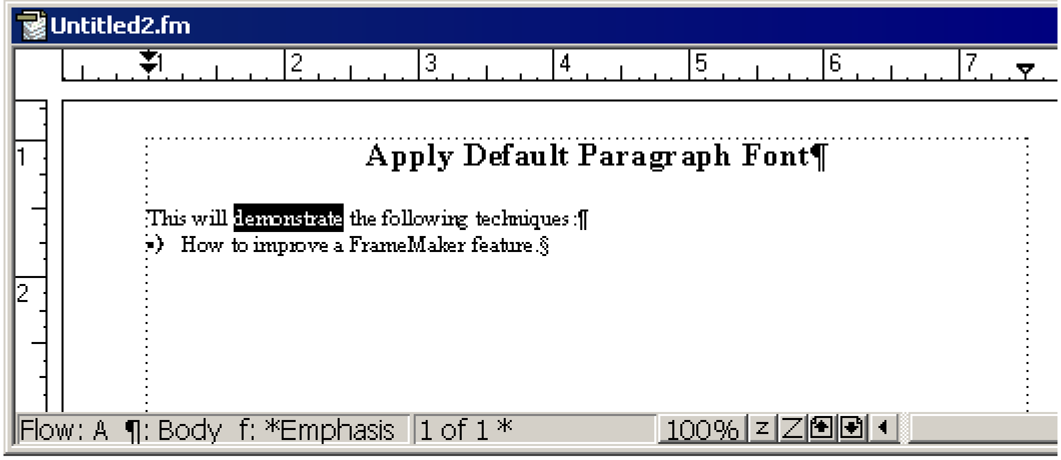
```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
Apply TextProperties TextRange(TextSelection)
  Properties(vPgfFmt.Properties);
```

You can see that the character formatting is removed from the text. However, there is a chance that this code will fail. If the paragraph format does not exist in the document's paragraph catalog, the **Get Object** command will return zero and the **Apply TextProperties** command will not do anything. Here is a way to accommodate this possibility.

Code Listing 13-8

```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgf.Properties);
EndIf
```

One small problem remains. Click in the text and look at the status bar at the lower left of the document window. You will see that the Emphasis character format is actually still applied to the text!


Untitled2.fm

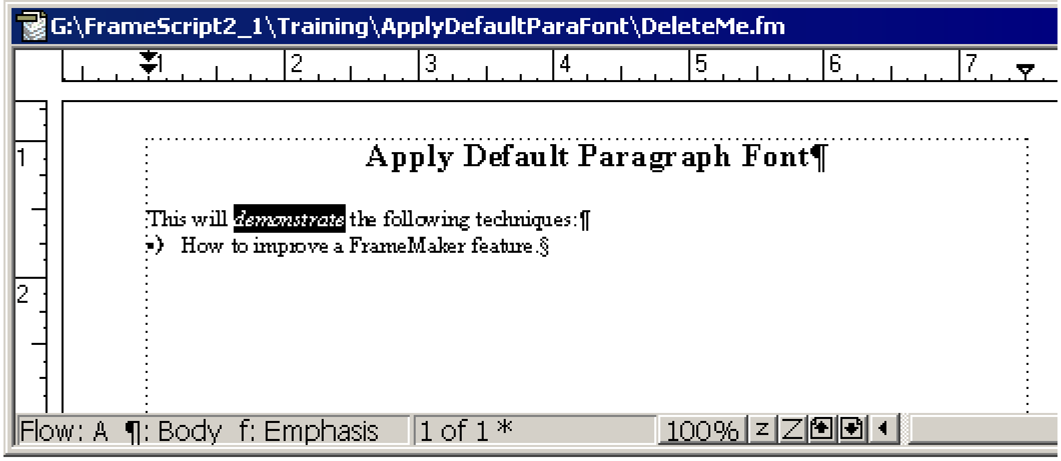
Apply Default Paragraph Font¶

This will demonstrate the following techniques:¶
•) How to improve a FrameMaker feature.§

Flow: A ¶: Body f: *Emphasis 1 of 1 * 100%

To illustrate this further, follow these steps.

1. Save the document.
2. Choose File > Import > Formats.
3. Make sure the Character Formats and Remove Other Format/Layout Overrides checkboxes are checked.
4. Click the Import button.


G:\FrameScript2_1\Training\ApplyDefaultParaFont\DeleteMe.fm

Apply Default Paragraph Font¶

This will *demonstrate* the following techniques:¶
•) How to improve a FrameMaker feature.§

Flow: A ¶: Body f: Emphasis 1 of 1 * 100%

Look at the text and you will see that the Emphasis character format properties are again applied to the text. The explanation for this is simple. The **CharTag** property of the text must be removed and applying the **PgfFmt** properties does not do this. Fortunately, the solution is also simple; try this revised code on the emphasized text.

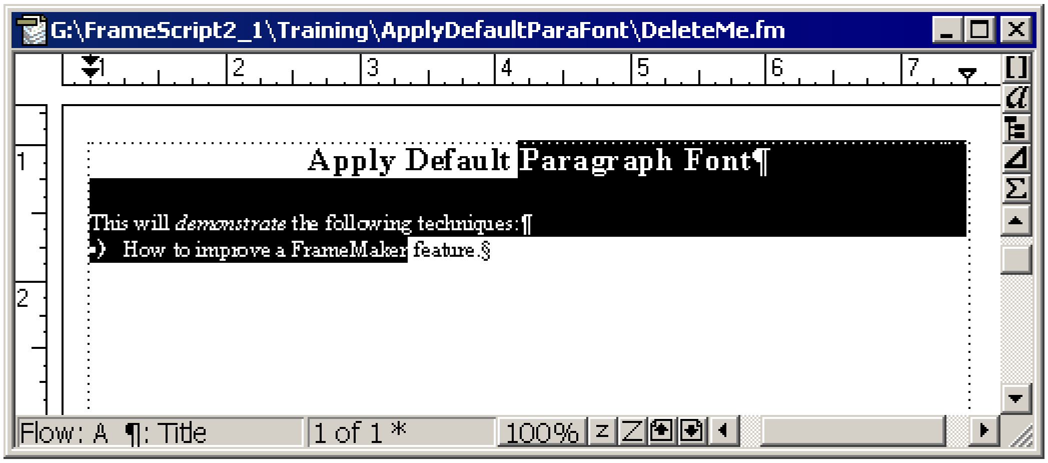
Code Listing 13-9

```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgf.Properties);
EndIf
```

```
New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(TextSelection)
  Properties(vPropertyList);
```

Working with Multiple Paragraphs

To understand what is involved with multiple paragraphs in a text selection, let's briefly return to our discussion of **TextRange** and **TextLoc** objects. Set up a document with three paragraphs like we did at the beginning of this tutorial. Start about half way through the first paragraph and select right through to about half way through the last paragraph.


G:\FrameScript2_1\Training\ApplyDefaultParaFont\DeleteMe.fm

Apply Default Paragraph Font¶

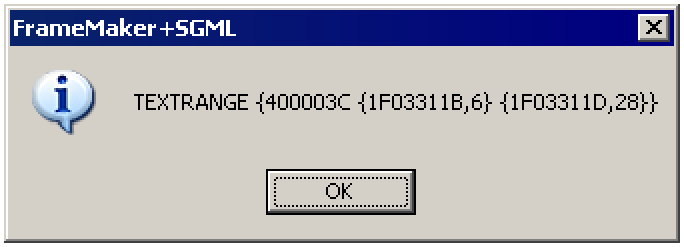
This will *demonstrate* the following techniques:¶

•) How to improve a FrameMaker feature.§

Run this code.

Code Listing 13-10

```
Display TextSelection;
```


FrameMaker+SGML

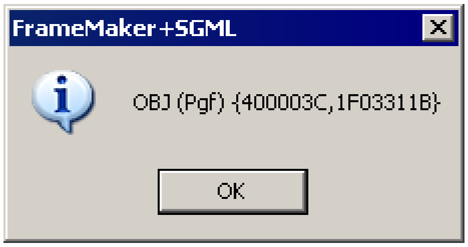
TEXTRANGE {400003C {1F03311B,6} {1F03311D,28}}

Earlier, we broke down the numbers that make up a **TextLoc**. Because the **TextSelection** is a **TextRange**, and a **TextRange** is made up of two **TextLoc** objects, you should be able see what each number represents. The first number is the **Unique Id** of the document and is followed by two sets of numbers in curly braces. Each set of numbers represents the **TextLoc** at each end of the **TextRange**. The two numbers in each **TextLoc** represent the **Unique Id** of the text object (**Pgf**) containing the **TextLoc** and the offset from the beginning of the text object.

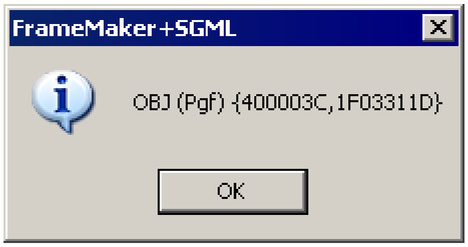
In our current **TextSelection**, each **TextLoc** has a different **Object**. You can illustrate that by running this code and looking carefully at the numbers.

Code Listing 13-11

```
Display TextSelection.Begin.Object;
Display TextSelection.End.Object;
```



OBJ (Pgf) {400003C,1F03311B}

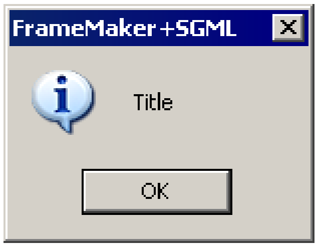


OBJ (Pgf) {400003C,1F03311D}

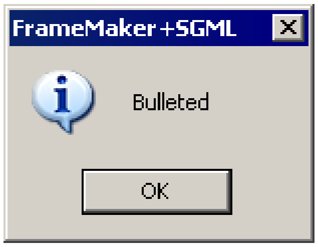
In our example, it becomes clearer if you try this.

Code Listing 13-12

```
Display TextSelection.Begin.Object.Name;
Display TextSelection.End.Object.Name;
```



Title



Bulleted

The key point is, because there is more than one paragraph in the selection, we can not apply the first paragraph's properties to all of the text in the selection. Even if the first and last paragraphs in the selection have the same name, you can't be sure that any paragraphs in between are the same format.

If you run the code below on the current TextSelection in our test document, you will see what happens. This is the code we used for a TextSelection in a single paragraph. Save the document first, so you can Revert to Saved to restore the document.

Code Listing 13-13

```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgf.Properties);
EndIf

New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(TextSelection)
  Properties(vPropertyList);
```

This is the same behavior we see if we apply the Default Paragraph Font in the FrameMaker interface.

Solving the Problem

This problem illustrates the two major aspects of programming in any computer language. The obvious aspect is knowing the syntax of the language—the various commands, parameters, and control structures that make up the language. The most important aspect—and usually the most difficult—is figuring out the steps or procedures involved in solving programming problems. This is what we might call the "logic" of the script.

You will use both aspects of programming hand-in-hand as you write FrameScript scripts. You will try to develop an "algorithm" for each of the tasks in your script. An algorithm is step-by-step problem-solving procedure for each task in your script. Then, you will look up the FrameScript syntax that is needed for the algorithm. As you become more familiar with the language, you can use your knowledge of the syntax to help you develop the logic of your scripts more quickly.

What we need is a way to loop through each paragraph in the TextSelection and apply the correct properties to each paragraph. There is no "canned" way of doing this, but we can use the **Get TextList** command with the **InRange** parameter to get started. Add three more short paragraphs to your sample document so you have at least six.

The **Get TextList** command gets a list of **TextItem** objects in a text object or text range. Since we are working with the **TextSelection**, which is a **TextRange**, we will use the **InRange** parameter. You specify which **TextItem** (text item) objects in the range that you want to return. In this case, we want the **PgfBegin** (paragraph begin) and **PgfEnd** (paragraph end) text items. Here is the basic syntax.

Code Listing 13-14

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Display vTextList;
```

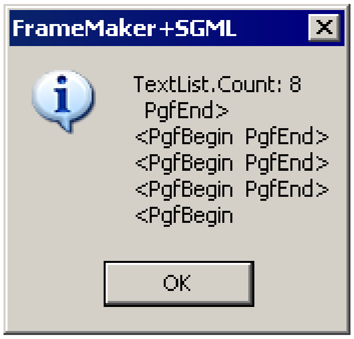
The essential feature of the **Get TextList** command is that *the text items are returned in the order they appear in the document*. Try the code on the following different text selections and examine each **vTextList** variable.

- Select a single word in a paragraph.
- Triple-click to select one or more entire paragraphs.
- Select one paragraph starting in the middle of the paragraph through the paragraph mark at the end of the paragraph.
- Start in the middle of the first paragraph and select through to the middle of the third paragraph.
- Select any paragraphs, starting in the middle of the first paragraph through the paragraph mark of the last paragraph.
- Select any paragraphs, starting at the beginning of the first paragraph through the middle of the last paragraph.

If you are having a difficult time visualizing the text items, try running the following code. It will isolate the **PgfBegin** and **PgfEnd** items and show how they relate to each paragraph in the selection. Each consecutive **PgfBegin** and **PgfEnd** member in the list constitutes a single paragraph in the text range.

Code Listing 13-15

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Set vMessageText = '';
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  If vTextItem.TextType = 'PgfBegin'
    Set vMessageText = vMessageText + '<PgfBegin';
  Else
    Set vMessageText = vMessageText + '  PgfEnd>'+CHARLF;
  EndIf
EndLoop
Display 'TextList.Count: '+vTextList.Count+CHARLF+vMessageText;
```



```
TextList.Count: 8
 PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin
```

We are looking for patterns to emerge that will help us solve the problem. We will use “pseudocode” to describe the pattern and possible solution. Pseudocode is a natural-language way of describing a programming solution. Using pseudocode allows us to focus on the logic and algorithm of the solution without worry about the specific FrameScript syntax.

There are two key properties of the vTextList variable that will give us clues to the solution. One, is the **vTextList.Count** property, which gives the number of members in the list; and two, whether the first member of the **vTextList** is a **PgfBegin** or **PgfEnd**. Again, notice that each consecutive **PgfBegin** and **PgfEnd** member in the list constitutes a single paragraph in the text range.

See if you observe the following five patterns as you experiment with the code. Remember that these patterns are pseudocode and not legal FrameScript syntax.

```
If the vTextList count = 0
  Then
    Process the current text selection (with our existing code).

If the vTextList count is an even number and the first member is a
  PgfBegin
  Then
    Process all of the members of the list (all paragraphs).

If the vTextList count is an odd number and the first member is a
  PgfBegin
  Then
    Process member 1 through the second last member in the list
      (count - 1).
    Process the last member (the partial last line).
```

```
If the vTextList count is an even number and the first member is a
  PgfEnd
  Then
    Process member 2 through the second last member in the list
      (count - 1).
    Process the first member (the partial first line).
    Process the last member (the partial last line).

If the vTextList count is an odd number and the first member is a
  PgfEnd
  Then
    Process member 2 through the last member in the list.
    Process the first member (the partial first line).
```

It turns out that no matter how any selection begins or ends, or the number of paragraphs in the selection, the **TextList** will follow one of the five patterns. This gives us a good basis for solving the problem. Let's begin to morph the pseudocode into FrameScript syntax.

Code Listing 13-16

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);

If vTextList.Count = 0
  Run ProcessTextSelection;
  LeaveSub;
EndIf

Get Member Number(1) From(vTextList) NewVar(vMemberOne);

If vMemberOne.TextType = 'PgfBegin'
  If (vTextList.Count % 2) = 0  // Even number
    Run ProcessParagraphs
  Else  // Odd number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial last line)
  EndIf
Else
  If (vTextList.Count % 2) = 0  // Even number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial first line)
    Run ProcessTextSelection (for the partial last line)
  Else  // Odd number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial first line)
  EndIf
EndIf
```

We now have a mixture of pseudocode and FrameScript syntax that deals with each one of our five possible patterns. Before going further, let's explain how we tell if the number of members in **vTextList** is odd or even. We are using the *modulus* (**%**) operator, which returns the remainder when two integers are divided. For example, 5 divided by 2 equals 2 with a remainder of 1, so the modulus of 5 divided by 2 is 1. Any even number divided by two will have a zero remainder (modulus), so this is a simple way to test for odd or even numbers.

Because we are still focusing on logic, we have referred to the actual formatting tasks as ProcessTextSelection and ProcessParagraphs and have left out the details. This tells us that it may be efficient to put each of these tasks in a subroutine, and call each subroutine from the main script where it is appropriate. Now we can turn our attention to the syntax of the two subroutines.

Processing Selected Text

We already know how to apply the Default Paragraph Font to the text selection.

Code Listing 13-17

```
Set vPgf = TextSelection.Begin.Object;
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(TextSelection)
    Properties(vPgf.Properties);
EndIf

New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(TextSelection)
  Properties(vPropertyList);
```

What we want to do is generalize the code so it can work with any **TextRange**, not just the document's **TextSelection**. A good place to start is with the syntax to make a new **TextRange**.

```
New TextRange NewVar(vTextRange) Object(vPgf)
  Offset(vBeginOffset) Offset(vEndOffset);
```

A **TextRange** can have a second **Object** parameter if it spans more than one paragraph, but for this script, we are only concerned with processing one paragraph at a time.

Once we have a **TextRange**, we can use the rest of the code to apply the Default Paragraph Font to it. The rest of the code is the same, except we have substituted **TextSelection** with the **vTextRange** variable that we created in the first two lines. We can now use this code on any **TextRange**, not just the current **TextSelection** in the document.

Code Listing 13-18

```
New TextRange NewVar(vTextRange) Object(vPgf)
  Offset(vBeginOffset) Offset(vEndOffset);

Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgf.Properties);
EndIf
```

```
New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(vTextRange)
  Properties(vPropertyList);
```

This code will form our **ProcessTextSelection** subroutine, so let's wrap it in the FrameScript **Sub**/**EndSub** statements.

Code Listing 13-19

```
Sub ProcessTextSelection
//
New TextRange NewVar(vTextRange) Object(vPgf)
  Offset(vBeginOffset) Offset(vEndOffset);

Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgf.Properties);
EndIf

New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(vTextRange)
  Properties(vPropertyList);
//
EndSub
```

When we call this subroutine, we will have to supply some parameters for the **New TextRange** command—**vPgf**, **vBeginOffset**, and **vEndOffset**. Let's see how we do that so it works for the current TextSelection. Follow these steps.

1. Copy the ProcessTextSelection subroutine into a new script.
2. Apply the Emphasis character format to one of the words in a paragraph in your document.
3. Select the emphasized word.
4. Type the following code above the subroutine in your script.

Code Listing 13-20

```
Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
  vBeginOffset(TextSelection.Begin.Offset)
  vEndOffset(TextSelection.End.Offset);
```

5. Run the script.

The script will apply the Default Paragraph Font to the selected word.

We can use this subroutine to process the partial first and last lines identified in our pseudocode. We simply have to modify the parameters that we pass to the subroutine. Here is how we call the subroutine for a selection that starts after the beginning of the paragraph and includes the end of the paragraph (as in the partial first line in the pseudocode).

Code Listing 13-21

```
Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
  vBeginOffset(TextSelection.Begin.Offset)
  vEndOffset(ObjEndOffset);
```

ObjEndOffset is a special FrameScript variable that indicates the end of a text object, in this case, the **Pgf**.

Here is how we call the **ProcessTextSelection** subroutine for a selection that starts at the beginning of the paragraph and stops before the end of the paragraph (as in the partial last line in the pseudocode).

Code Listing 13-22

```
Run ProcessTextSelection vPgf(TextSelection.End.Object)
  vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
```

We can also use this the subroutine to process an entire paragraph by using this call to the subroutine. We will use this for each paragraph in the **ProcessParagraphs** subroutine.

Code Listing 13-23

```
Run ProcessTextSelection vPgf(Paragraph Object)
  vBeginOffset(0) vEndOffset(ObjEndOffset);
```

We will need to supply the correct **Paragraph Object** in the **vPgf** parameter, and that will be the topic of the next section.

Processing Paragraphs

The key question now is, how do we process full paragraphs in our **vTextList** of **PgfBegin** and **PgfEnd** text items? Select two paragraphs in their entirety, and run the following code.

Code Listing 13-24

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Display vTextList; // Display the list.

Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Display vTextItem; // Display each text item.
EndLoop
```

The first **Display** command will display the **TextList**; the **Display** command inside the loop will display each of the text items in the list. A **TextItem** has three properties, **TextType**, **TextData**, and **TextOffset**. We've already used the **TextType** property to identify the first text item in the text list as either a **PgfBegin** or **PgfEnd**.

```
If vMemberOne.TextType = 'PgfBegin'
```

The **TextOffset** property gives the offset of the text item from the beginning of the text object. (A **PgfBegin** text item will always have a **TextOffset** of 0.) In this script, we won't use the **TextOffset** property.

The **TextData** property returns the **Pgf** (paragraph) object of the text item. This is what we need to process the paragraphs in the text list. Modify the code to display the TextData of each text item and run the script. Make sure two entire paragraphs are selected first.

Code Listing 13-25

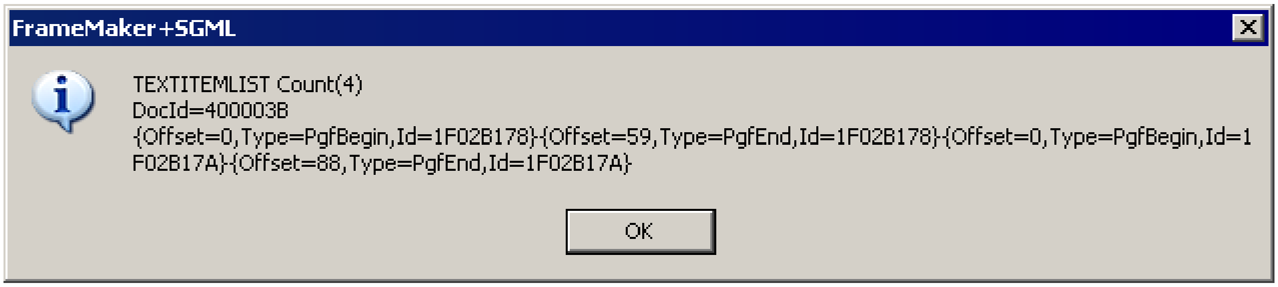
```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Display vTextList; // Display the list.

Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Display vTextItem.TextData; // Display each text item.
EndLoop
```

As before, the first **Display** command displays the **TextList**. The **Display** command inside the loop displays each **Pgf** object. You should notice that each **Pgf** object is displayed twice, once on the **PgfBegin** text item and again on the **PgfEnd** text item. The **PgfBegin** and **PgfEnd** items for each paragraph always occurs in pairs. Since we don't want to process each paragraph twice, we only need one of the items. For our script we will use the **PgfBegin** items so we want every other item, starting with the first **PgfBegin** item. With the loop we are using, it is easy to move through the loop by twos; simply change the value on the **Incr** parameter to **2**. Select two paragraphs in their entirety, and run the modified code.

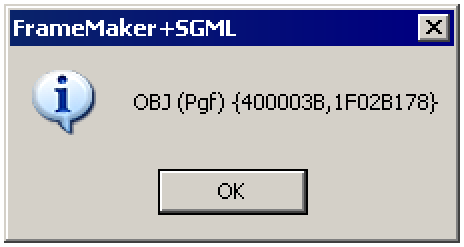
Code Listing 13-26

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Display vTextList; // Display the list.

Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(2)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Display vTextItem.TextData; // Display each text item.
EndLoop
```

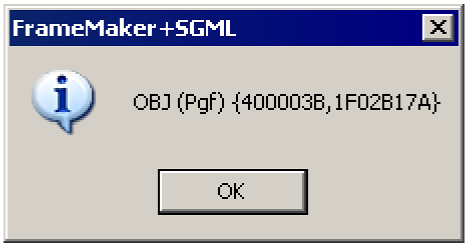
The TextList contains 4 items, but the loop will only display the 2 paragraph objects.



```
TEXTITEMLIST Count(4)
DocId=400003B
{Offset=0,Type=PgfBegin,Id=1F02B178}{Offset=59,Type=PgfEnd,Id=1F02B178}{Offset=0,Type=PgfBegin,Id=1
F02B17A}{Offset=88,Type=PgfEnd,Id=1F02B17A}
```



```
OBJ (Pgf) {400003B,1F02B178}
```



```
OBJ (Pgf) {400003B,1F02B17A}
```

There is one more important issue to deal with. Our modified loop will process each paragraph correctly, provided that the text list begins with a **PgfBegin** text item. If the text list begins with a PgfEnd item, we want the list to be processed beginning with the 2nd member of the list. We can do this by changing value on the **Init** parameter of the loop to 2.

Code Listing 13-27

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);
Display vTextList; // Display the list.

Loop While(n <= vTextList.Count) LoopVar(n) Init(2) Incr(2)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Display vTextItem.TextData; // Display each text item.
EndLoop
```

We also want to determine how many of the members of the list are processed. If the list ends with a **PgfBegin** item, we will not want to process it because it shows that the last paragraph is not entirely selected. The (**n <= vTextList.Count**) value will cause the loop to end at the last member; if we replace it with a **vStop** variable, we can determine if the last member is processed by substituting the appropriate value.

Let's generalize the code and put it into a subroutine called **ProcessParagraphs**.

Code Listing 13-28

```
Sub ProcessParagraphs
//
Loop While(n <= vStop) LoopVar(n) Init(vStart) Incr(2)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Run ProcessTextSelection vPgf(vTextItem.TextData)
    vBeginOffset(0) vEndOffset(ObjEndOffset);
EndLoop
//
EndSub
```

We are using a **vStart** variable for the Init parameter, so we can pass in the correct value—if the text list begins with a **PgfBegin** item, we will use **1**; if it begins with a **PgfEnd** item, we will use **2**. There is also a **vStop** variable which can be set to skip the last member of the list. To skip the last member of the list, we would use **vTextList.Count - 1** for the **vStop** value.

Also notice that there is a call to the **ProcessTextSelection** subroutine. Since ProcessParagraphs works with complete paragraphs, we are using **0** (zero) for the beginning offset and the **ObjEndOffset** variable for the ending offset. In other words, we are applying the Default Paragraph Font to each complete paragraph in the selection.

Passing Parameters to Subroutines

Let's look at each part of our modifed pseudocode and see how we can pass the correct parameters to the **ProcessTextSelection** and **ProcessParagraphs** subroutines. Keep in mind that we have to deal with each of the five possible patterns that any text selection will have.

Code Listing 13-29

```
Get TextList InRange(TextSelection) PgfBegin PgfEnd
  NewVar(vTextList);

If vTextList.Count = 0
  Run ProcessTextSelection;
  LeaveSub;
EndIf

...

Sub ProcessTextSelection
//
New TextRange NewVar(vTextRange) Object(vPgf) Offset(vBeginOffset)
  Offset(vEndOffset);
Get Object Type(PgfFmt) Name(vPgf.Name) NewVar(vPgfFmt);
If vPgfFmt
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgfFmt.Properties);
Else
  Apply TextProperties TextRange(vTextRange)
    Properties(vPgf.Properties);
EndIf

New PropertyList NewVar(vPropertyList) CharTag('');
Apply TextProperties TextRange(vTextRange)
  Properties(vPropertyList);
//
EndSub
```

The first possibility will occur if there is a text selection in a single paragraph where the selection doesn't contain either a paragraph begin or paragraph end. In this case, we simply want to process the text selection and exit the script. Look at the **ProcessTextSelection** subroutine and see what parameters it requires. It requires three of the parameters on the **New TextRange** command. Here is the modified call to the subroutine with the three parameters supplied.

Code Listing 13-30

```
If vTextList.Count = 0
  Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
    vBeginOffset(TextSelection.Begin.Offset)
    vEndOffset(TextSelection.End.Offset);
  LeaveSub;
EndIf
```

Passing parameters to a subroutine is straightforward; you supply the parameter name, and then include the parameter value in parentheses. The key is to make sure that you are passing valid values to the subroutine.

Look at the **ProcessParagraphs** subroutine and notice what parameters it passes to the **ProcessTextSelection** subroutine to apply the default paragraph properties to the entire paragraph.

Code Listing 13-31

```
Sub ProcessParagraphs
//
Loop While(n <= vStop) LoopVar(n) Init(vStart) Incr(2)
  Get Member Number(n) From(vTextList) NewVar(vTextItem);
  Run ProcessTextSelection vPgf(vTextItem.TextData)
    vBeginOffset(0) vEndOffset(ObjEndOffset);
EndLoop
//
EndSub
```

Returning to the modified pseudocode, let's see how to supply the required parameters on the subroutine calls. If the text selection contains one or more **PgfBegin** or **PgfEnd** items the script will test the first member of the **TextList**.

Code Listing 13-32

```
Get Member Number(1) From(vTextList) NewVar(vMemberOne);

If vMemberOne.TextType = 'PgfBegin'
  If (vTextList.Count % 2) = 0  // Even number
    Run ProcessParagraphs
  Else  // Odd number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial last line)
  EndIf
Else
  If (vTextList.Count % 2) = 0  // Even number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial first line)
    Run ProcessTextSelection (for the partial last line)
  Else  // Odd number
    Run ProcessParagraphs
    Run ProcessTextSelection (for the partial first line)
  EndIf
EndIf
```

Look first at the calls to the **ProcessTextSelection** subroutine and remember that we have to supply three parameters. If the first member of the **TextList** is a **PgfBegin** item, and and the number of text items is even, then one or more complete paragraphs are selected. If the number of text items is odd, then the end of the selection contains a partially selected paragraph. In that case, we use **TextSelection.End.Object** for the **vPgf** parameter, **0** (zero) for the **vBeginOffset** parameter, and **TextSelection.End.Offset** for the **vEndOffset** parameter. This situation also occurs if the first member of the **TextList** is a **PgfEnd** item and the number of members is even.

If the first member of the TextList is a **PgfEnd** item and the number of text items is odd, then the first paragraph in the text selection is partially selected. In this case, we use **TextSelection.Begin.Object** for the **vPgf** parameter, **TextSelection.Begin.Offset** for the **vBeginOffset** parameter, and **ObjEndOffset** for the **vEndOffset** parameter. This situation also occurs if the first member of the **TextList** is a **PgfEnd** item and the number of members is even.

Here is the code with the modified calls to **ProcessTextSelection** to handle selections that contain partially selected paragraphs.

Code Listing 13-33

```
Get Member Number(1) From(vTextList) NewVar(vMemberOne);

If vMemberOne.TextType = 'PgfBegin'
  If (vTextList.Count % 2) = 0
    Run ProcessParagraphs;
  Else
    Run ProcessParagraphs;
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  EndIf
Else
  If (vTextList.Count % 2) = 0
    Run ProcessParagraphs;
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  Else
    Run ProcessParagraphs;
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
  EndIf
EndIf
```

All that is left now is to pass the correct parameters to the **ProcessParagraphs** subroutine. We need to supply the correct **vStart** and **vStop** values so that only completely selected paragraphs are processed.

If the first item in the list is a **PgfBegin**, and the number of members is even, then the text selection contains only completely selected paragraphs. In this case **vStart** will be **1** and **vStop** will be **vTextList.Count**; in other words, we want to process every other member of the entire list. If the number of members is odd, we want to skip the last member in the list, which is a **PgfBegin** item; in this case, **vStop** will be **vTextList.Count** - **1**.

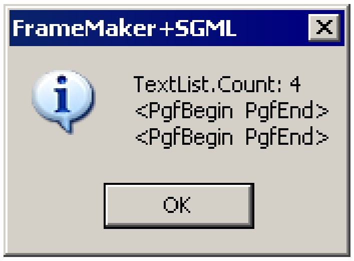
If the first member is a **PgfEnd** item, and the number of members is even, then we want **vStart** to be **2** (to skip the first item in the list), and **vStop** to be **vTextList.Count** - **1** (to skip the last item in the list). If the number of members is odd, we only want to skip the first item in the list; **vStart** will be **2** and **vStop** will be **vTextList.Count**. Here is the code with all of the parameters filled in.

Code Listing 13-34

```
Get Member Number(1) From(vTextList) NewVar(vMemberOne);

If vMemberOne.TextType = 'PgfBegin'
  If (vTextList.Count % 2) = 0
    Run ProcessParagraphs vStart(1) vStop(vTextList.Count);
  Else
    Run ProcessParagraphs vStart(1) vStop(vTextList.Count - 1);
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  EndIf
Else
  If (vTextList.Count % 2) = 0
    Run ProcessParagraphs vStart(2) vStop(vTextList.Count - 1);
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  Else
    Run ProcessParagraphs vStart(2) vStop(vTextList.Count);
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
  EndIf
EndIf
```

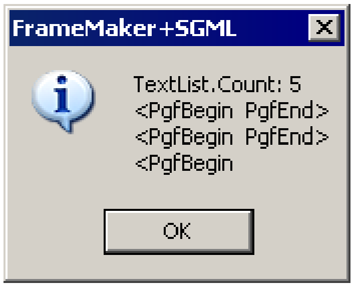
By now, the logic of the script should be clearer. To further solidify the concepts, here is the code with a screenshot showing each of the possibilities. As you examine the screenshots, remember that the number of members in the list does not matter in the script. What matters is kind of item the first member is, and whether the list contains an even or odd number of members. Try to associate each line in the screenshot with the corresponding subroutine call in the code.

```
Get Member Number(1) From(vTextList) NewVar(vMemberOne);

If vMemberOne.TextType = 'PgfBegin'
  If (vTextList.Count % 2) = 0
```



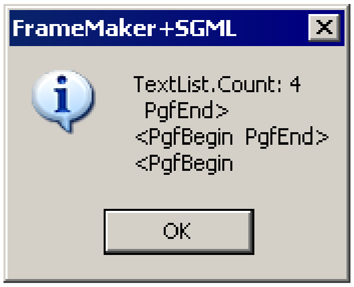
```
TextList.Count: 4
<PgfBegin PgfEnd>
<PgfBegin PgfEnd>
```

```
    Run ProcessParagraphs vStart(1) vStop(vTextList.Count);
  Else
```



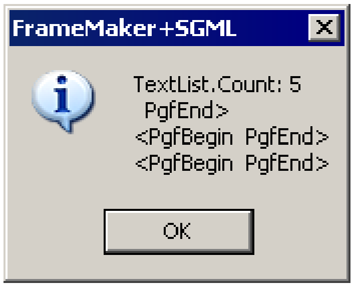
```
TextList.Count: 5
<PgfBegin PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin
```

```
    Run ProcessParagraphs vStart(1) vStop(vTextList.Count - 1);
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  EndIf
Else
  If (vTextList.Count % 2) = 0
```



```
TextList.Count: 4
 PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin
```

```
    Run ProcessParagraphs vStart(2) vStop(vTextList.Count - 1);
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
    Run ProcessTextSelection vPgf(TextSelection.End.Object)
      vBeginOffset(0) vEndOffset(TextSelection.End.Offset);
  Else
```



```
TextList.Count: 5
 PgfEnd>
<PgfBegin PgfEnd>
<PgfBegin PgfEnd>
```

```
    Run ProcessParagraphs vStart(2) vStop(vTextList.Count);
    Run ProcessTextSelection vPgf(TextSelection.Begin.Object)
      vBeginOffset(TextSelection.Begin.Offset)
      vEndOffset(ObjEndOffset);
  EndIf
EndIf
```

14 Code Tester and Extractor

This book contains many code listings that are included as text files on the companion CD-ROM. As the book was developed, most of the code was typed and tested directly in FrameMaker. I used scripts to help me test and format the code, and to extract the code listings to for inclusion on the CD.

All of the scripts in this chapter make some assumptions based on my knowledge of the book's templates and how the documents were formatted. If you modify these scripts for your own use, you may need to add more error handling to account for your unique circumstances.

Testing the Code

Many of the code listings were written in the FrameScript Script Window so that they could be tested before including them in the book. It was tedious copying and pasting the code back and forth. I wanted to use soft-returns at the end of each line of code and a hard-return at the end of a block of code. This meant that code pasted from the Script Window had to be reformatted with soft-returns.

A helpful solution was to type the code in the FrameMaker file. The code was then selected in the document, and a special script was used to run the selected script directly from the FrameMaker document. Here is an outline of the tasks involved in developing the script, which I called **RunSelectedCode.fsl**.

- Make sure one or more lines are selected in the document.
- Create a temporary text file on the disk.
- Write each selected line to the text file.
- Run the text file with the **Run** command.
- Delete the temporary file.

Of course, not all code listings can be run with RunSelectedCode. For example, if the code listing requires an insertion point in a table, it will not work. But in many cases, it works fine and saves a lot of time.

Test for Selected Lines

The first task is to make sure that there are lines selected in the active document. For completeness, though, we should first make sure that there is an active document.

Code Listing 14-1

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf
```

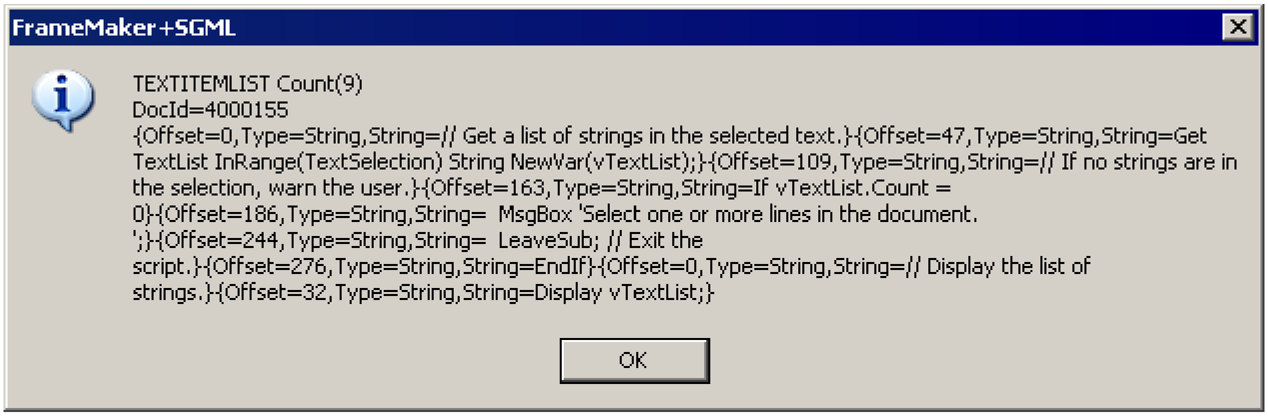
```
// Test for a text selection.
If TextSelection.Begin.Object = 0
  MsgBox 'Select one or more lines in the document.    ';
  LeaveSub; // Exit the script.
EndIf
```

Now we can get a list of the selected lines. We will use the **Get TextList** command with the **String** option. This is the simplest approach because we know that the code listings do not have any character formatting applied to them. This means that each **String** will correspond to a line of selected code.

Code Listing 14-2

```
// Get a list of strings in the selected text.
Get TextList InRange(TextSelection) String NewVar(vTextList);
// If no strings are in the selection, warn the user.
If vTextList.Count = 0
  MsgBox 'Select one or more lines in the document.    ';
  LeaveSub; // Exit the script.
EndIf

// Display the list of strings.   REMOVE FROM FINISHED SCRIPT.
Display vTextList;             // REMOVE FROM FINISHED SCRIPT.
```

Combine the previous two listings in the Script Window, select some text in the document, and try the code.



```
TEXTITEMLIST Count(9)
DocId=4000155
{Offset=0,Type=String,String=// Get a list of strings in the selected text.}{Offset=47,Type=String,String=Get
TextList InRange(TextSelection) String NewVar(vTextList);}{Offset=109,Type=String,String=// If no strings are in
the selection, warn the user.}{Offset=163,Type=String,String=If vTextList.Count =
0}{Offset=186,Type=String,String=  MsgBox 'Select one or more lines in the document.
';}{Offset=244,Type=String,String=  LeaveSub; // Exit the
script.}{Offset=276,Type=String,String=EndIf}{Offset=0,Type=String,String=// Display the list of
strings.}{Offset=32,Type=String,String=Display vTextList;}
```

There should be one member in the list for each selected line in the document. In the above example, there are 9 lines selected and the **TextList** contains 9 members. If your TextList has more members than you have lines selected, it is because there are formatting changes or anchor objects in the selection.

Make a Temporary Script

FrameScript has a special variable, **TmpDir**, that returns the operating system's temporary folder. This is a good place to create a temporary file to hold the script commands.

Code Listing 14-3

```
// Set a variable for the temporary script file.
Set vScriptName = TmpDir+DIRSEP+'Script.fsl';
// Make the temporary script file.
New TextFile File(vScriptName) NewVar(vScript);
```

Before writing the strings to the temporary script file, we have to make sure that there are no non-ASCII characters that will cause problems with FrameScript. In some code listings, I used non-breaking spaces to indent the text. We will replace them with regular spaces before writing them to the text file. We will need to make a string variable for the non-breaking space.

Code Listing 14-4

```
// Make a string variable for the non-breaking space character.
New String NewVar(vNbsp) IntValue(17);
```

Before moving on, let's discuss how we determined the "17" on the **New String** command. FrameMaker represents characters internally using ASCII numbers; it is one of these numbers that you use in the **IntValue** parameter. Here is a script that will help you determine the ASCII number for a particular character.

Code Listing 14-5

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub;
EndIf

// Make sure there is a text selection.
If TextSelection.Begin.Object = 0
  MsgBox 'Select a character and rerun the script.    ';
  LeaveSub;
EndIf

// Find and display the ASCII number of the character.
Loop While(n <= 255) LoopVar(n) Init(1) Incr(1)
  New String NewVar(vString) IntValue(n);
  Find String(vString) InRange(TextSelection) ReturnStatus(vFound);
  If vFound
    Display n;
    LeaveLoop;
  EndIf
EndLoop
```

To use the script, type the special character in a FrameMaker document. Select the character (make sure you select only one character) and run the script; it will display the ASCII number of the selected character.

Write the Selected Lines to the File

We can loop through the selected strings and write them line-by-line to the text file. Before writing each line, we will use the **Get String** command to replace non-breaking spaces with regular spaces. After writing all of the lines to the text file, we will close it.

Code Listing 14-6

```
// Loop through the list of strings in the selection.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  // Get the string from the list.
  Get Member Number(n) From(vTextList) NewVar(vString);
  // Replace any non-breaking spaces with regular spaces.
  Get String FromString(vString.TextData) NewVar(vString)
    ReplaceAll(vNbsp) With(' ');
  // Write the string to the text file.
  Write Object(vScript) vString;
EndLoop

// Close the temporary script file.
Close TextFile Object(vScript);
```

Run the Script

To run the script, we must first set up an object variable for the script by supplying the path to the **New ScriptVar** command. Then you can use the **Run** command with the **ScriptVar** object.

Code Listing 14-7

```
// Set a variable for the script.
New ScriptVar NewVar(vScript) File(vScriptName);
// Run the script.
Run vScript;
```

Delete the Temporary Script

To delete the temporary script, use the **Delete** command with the **File** parameter.

Code Listing 14-8

```
// Delete the temporary script file.
Delete File(vScriptName);
```

Now you can combine the code and try it out. Below is the complete code listing.

IMPORTANT: Be careful when running this script, because, in many cases, it will alter the document containing the selected text. You should save the changes before running the script so you can revert to the last saved version, if necessary.

Code Listing 14-9

```
// RunSelectedCode.fsl
// Runs the selected text in the active document with FrameScript.

// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf
```

```
// Test for a text selection.
If TextSelection.Begin.Object = 0
  MsgBox 'Select one or more lines in the document.    ';
  LeaveSub; // Exit the script.
EndIf

// Get a list of strings in the selected text.
Get TextList InRange(TextSelection) String NewVar(vTextList);
// If no strings are in the selection, warn the user.
If vTextList.Count = 0
  MsgBox 'Select one or more lines in the document.    ';
  LeaveSub; // Exit the script.
EndIf

// Set a variable for the temporary script file.
Set vScriptName = TmpDir+DIRSEP+'Script.fsl';
// Make the temporary script file.
New TextFile File(vScriptName) NewVar(vScript);

// Make a string variable for the non-breaking space character.
New String NewVar(vNbsp) IntValue(17);

// Loop through the list of strings in the selection.
Loop While(n <= vTextList.Count) LoopVar(n) Init(1) Incr(1)
  // Get the string from the list.
  Get Member Number(n) From(vTextList) NewVar(vString);
  // Replace any non-breaking spaces with regular spaces.
  Get String FromString(vString.TextData) NewVar(vString)
    ReplaceAll(vNbsp) With(' ');
  // Write the string to the text file.
  Write Object(vScript) vString;
EndLoop

// Close the temporary script file.
Close TextFile Object(vScript);

// Set a variable for the script.
New ScriptVar NewVar(vScript) File(vScriptName);
// Run the script.
Run vScript;

// Delete the temporary script file.
Delete File(vScriptName);
```

Formatting the Code

Early draft editions of this book did not have code listing titles or the rule above the code listing. All code was simply formatted with the “Code” paragraph format. I needed a way to identify the code for the CD, as well as set it apart from the rest of the text. Here is a description of the addition formats I created, followed by two sample listings.

Paragraph Format	Purpose
CodeTitle	Autonumbered paragraph above the code listing.
CodeFirst	The first paragraph of code in the listing. Adds the rule between the code listing title and the first paragraph of code.
CodeLast	The last paragraph of code in the listing. At this point, it has the same appearance as Code, but allows flexibility to format the last paragraph uniquely in the future.
CodeOne	For single paragraph code listings.
Code	Used code paragraphs between the first and last.

Code Listing 1-15

```
CodeOne
```

Code Listing 1-16

```
CodeFirst

Code

CodeLast
```

To format the code, I needed a script that would apply the correct format to the selected code lines. I chose to work with selected lines instead of all of the code in the document, so that I could selectively leave out some code listings.

First of all, we should test for an active document, as well as a text selection in a paragraph.

Code Listing 14-10

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
EndIf

// Test for a text selection.
If TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Select one or more code paragraphs in the document.    ';
  LeaveSub; // Exit the script.
EndIf
```

The next step is to get a list of paragraphs in the selected text. The **TextSelection** property represents all of the selected text in the document. We will use the **Get TextList** command to get all of the **PgfBegin** text items in the selection. That means that you must have the beginning of at least one paragraph selected in order for the script to work.

To make it easier to select entire paragraphs, triple-click on the first one you are selecting, and hold down the mouse and drag to select any additional paragraphs. Or, you can click in the first paragraph, and press Shift+Control+Down Arrow to select paragraphs.

Here is the code to get the list of paragraphs in the selection.

Code Listing 14-11

```
// Get a list of paragraph begin text items.
Get TextList InRange(TextSelection) PgfBegin NewVar(vTextList);
// If there are no paragraphs in the selection, warn the user.
If vTextList.Count = 0
  MsgBox 'Select one or more code paragraphs in the document.    ';
  LeaveSub; // Exit the script.
EndIf
```

If no paragraphs are selected, the script warns the user and exits. If there are paragraphs in the selection, we can add the CodeTitle paragraph above the first paragraph.

Code Listing 14-12

```
// Get first paragraph in the selection.
Get Member Number(1) From(vTextList) NewVar(vPgfBegin);
// Set a variable for the first paragraph.
Set vPgf = vPgfBegin.TextData;
// Add the CodeTitle paragraph before the first paragraph.
New Pgf NewVar(vTitlePgf) PrevObject(vPgf.PrevPgfInFlow)
  PgfFmtName('CodeTitle');
```

The **New Pgf** command assumes that the CodeTitle paragraph format exists in the document; if it doesn't the Body paragraph format will be used instead. This wasn't a problem here, but in some scripts, you should test for the existence of the paragraph format before using it with the **PgfFmtName** parameter.

Now we can process the code paragraphs. We have already extracted the first one from the list. If it is the only paragraph in the list, it will be formatted with CodeOne; otherwise, we will apply the CodeFirst format.

Code Listing 14-13

```
// See how many paragraphs are in the list.
If vTextList.Count = 1
  // Get the CodeOne format object.
  Get Object Type(PgfFmt) Name('CodeOne') DocObject(ActiveDoc)
    NewVar(vPgfFmt);
  // Apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
Else
  // Get the CodeFirst format object.
  Get Object Type(PgfFmt) Name('CodeFirst') DocObject(ActiveDoc)
    NewVar(vPgfFmt);
  // Apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
EndIf
```

We are making the safe assumption that the selected paragraphs were already formatted with the Code paragraph format. That means that now we only have to deal with the last paragraph in the selection. We do this by getting the last member of the **TextList**. This will be done in the **Else** section of the above code. Here is the entire **If/Else/EndIf** block.

Code Listing 14-14

```
// See how many paragraphs are in the list.
If vTextList.Count = 1
  // Get the CodeOne format object.
  Get Object Type(PgfFmt) Name('CodeOne') DocObject(ActiveDoc)
    NewVar(vPgfFmt);
  // Apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
Else
  // Get the CodeFirst format object.
  Get Object Type(PgfFmt) Name('CodeFirst') DocObject(ActiveDoc)
    NewVar(vPgfFmt);
  // Apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
  // Get the last paragraph from the text list.
  Get Member Number(vTextList.Count) From(vTextList)
    NewVar(vPgfBegin);
  // Set a variable for the last paragraph.
  Set vPgf = vPgfBegin.TextData;
  // Get the CodeLast format object.
  Get Object Type(PgfFmt) Name('CodeLast') DocObject(ActiveDoc)
    NewVar(vPgfFmt);
  // Apply the properties to the paragraph.
  Set vPgf.Properties = vPgfFmt.Properties;
EndIf
```

Extracting the Code Listings

Using FrameScript to extract the code listings to text files will be a huge time saver. We want to create a separate folder for each listing and then name each listing with the same number as its CodeTitle autonumber. For example, chapter 1's listings will be in a folder called **01Listings** and its listings will be **01_001.fsl**, **01_002.fsl**, **01_003.fsl**, etc., corresponding to Code Listings 1-1, 1-2, 1-3, etc. We are padding the numbers with zeros so that they sort correctly on the CD.

Like most scripts, we have a good idea but we need a plan for developing the code. The best approach is to make a list of tasks and tackle each one individually.

- Get the folder of the existing document.
- Set the correct folder name, based on the document's chapter number.
- Make the code listing folder.
- Convert the CodeTitle autonumber string to the correct file name.
- Make a text file at the correct location.
- Write the appropriate code to the text file.
- Loop through all of the code listings in the document.

Get the Folder of the Existing Document

The code listing folders will go in the folder that contains the current document. We need a little bit of code that will drop the document name and leave us with the path. The easiest way to do this is to find the last folder separator in the path using the **Backward** option of the **Find String** command.

Code Listing 14-15

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Find the last folder separator in the document's path.
Find String(DIRSEP) InString(vCurrentDoc.Name) Backward
  ReturnPos(vPos);
// Drop the filename from the path, leaving the separator.
Get String FromString(vCurrentDoc.Name) EndPos(vPos)
  NewVar(vDocFolder);

// Display the document folder.
Display vDocFolder;
```

This code will fail if you run it on an Untitled document. You should make sure that the document's **Name** property is not an empty string before running the script. We will use the following test at the beginning of the final script.

Code Listing 14-16

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Test that the document has been saved.
  If ActiveDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub; // Exit the script.
  Else
    // Set a variable for the active document.
    Set vCurrentDoc = ActiveDoc;
  EndIf
EndIf
```

Set the Correct Folder Name

Setting the correct folder name is a matter of getting the value of the chapter number variable. If the chapter number is less than two digits, we will pad the front of it with a zero.

Code Listing 14-17

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Make a string from the chapter number.
New String NewVar(vListingFolder) Value(vCurrentDoc.ChapterNumber);

// If the size of the string is less than two characters,
// pad it with a zero.
If vListingFolder.Size = 1
  Set vListingFolder = '0' + vListingFolder;
EndIf
// Add the "Listings" suffix.
Set vListingFolder = vListingFolder + 'Listings';
```

```
// Display the listing folder.
Display vListingFolder;
```

Make the Listing Folder

With Windows FrameScript 2.1 and above, you can use code to make the listing folders. On the Macintosh, you will have to create the folders ahead of time.

```
// Make an ESystem object.
New ESystem NewVar(vESystem);
// Make the folder.
Run vESystem.CreateDirectory DirName(vDocFolder+vListingFolder);
```

You will have to combine this code with the scripts that get the folder names. Here is the complete script so far.

Code Listing 14-18

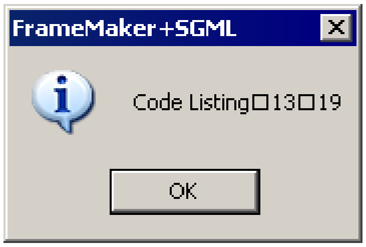
```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Test that the document has been saved.
  If ActiveDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub; // Exit the script.
  Else
    // Set a variable for the active document.
    Set vCurrentDoc = ActiveDoc;
  EndIf
EndIf

// Find the last folder separator in the document's path.
Find String(DIRSEP) InString(vCurrentDoc.Name) Backward
  ReturnPos(vPos);
// Drop the filename from the path, leaving the separator.
Get String FromString(vCurrentDoc.Name) EndPos(vPos)
  NewVar(vDocFolder);

// Make a string from the chapter number.
New String NewVar(vListingFolder) Value(vCurrentDoc.ChapterNumber);

// If the size of the string is less than two characters,
// pad it with a zero.
If vListingFolder.Size = 1
  Set vListingFolder = '0' + vListingFolder;
EndIf
// Save the chapter number for later use.
Set vChapterNumber = vListingFolder;
// Add the "Listings" suffix.
Set vListingFolder = vListingFolder + 'Listings';

// Make an ESystem object.
New ESystem NewVar(vESystem);
// Make the folder.
Run vESystem.CreateDirectory DirName(vDocFolder+vListingFolder);
```

Find the Correct File Name for a Listing

To get the correct file names from the CodeTitle paragraphs, we will begin by working with a single paragraph. We can click in a CodeTitle paragraph to test the code.

Code Listing 14-19

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Test for a text selection in a paragraph.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = vCurrentDoc.TextSelection.Begin.Object;

// Make sure you are in a CodeTitle paragraph.
If vPgf.Name not= 'CodeTitle'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Display the CodeTitle paragraph's autonumber.
Display vPgf.PgfNumber;
```

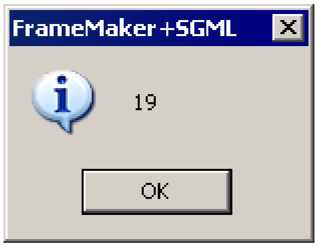


Code Listing□13□19

We know from the CodeTitle autonumber format that there is a non-breaking hyphen before the actual listing number. We can use this information to drop the front of the autonumber from the **PgfNumber** property. Combine the following code with the previous and try it out.

Code Listing 14-20

```
// Set a variable for the autonumber string.
Set vListingNumber = vPgf.PgfNumber;
// Make a string variable for the non-breaking hyphen.
New String NewVar(vNbhyphen) IntValue(21);
// Find the non-breaking hyphen in the number.
Find String(vNbhyphen) InString(vListingNumber) ReturnPos(vPos);
// Drop the front of the autonumber string.
Get String FromString(vListingNumber) StartPos(vPos+1)
  NewVar(vListingNumber);

// Display the listing number.
Display vListingNumber;
```

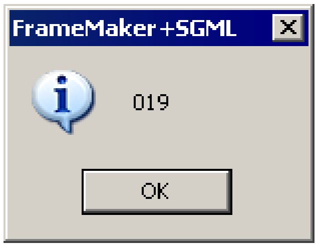


19

We still have to make sure the listing number is three digits, and pad it with one or more zeros, if necessary. It was simple with the chapter numbers because we only had a maximum of one zero to worry about. With the listing number, we can use this.

Code Listing 14-21

```
// Pad the listing number with zeros.
If vListingNumber.Size = 2
  Set vListingNumber = '0' + vListingNumber;
Else
  If vListingNumber.Size = 1
    Set vListingNumber = '00' + vListingNumber;
  EndIf
EndIf

// Display the padded number.
Display vListingNumber;
```



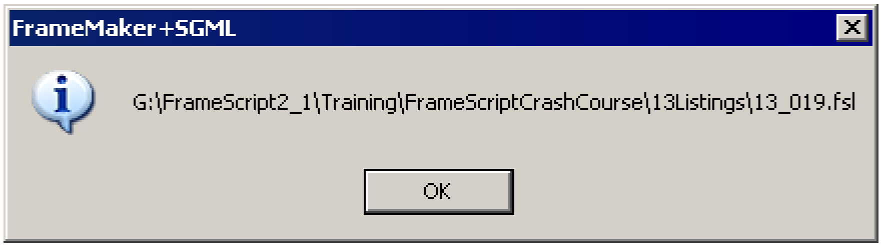
019

Now we can combine the variables to give us a complete path to the code listing text file. To do this, we need to combine parts of the previous code listings. This can be tricky because you usually will only need parts of your "task scripts." Be careful when combining the scripts so that you maintain the correct syntax. Here is the complete listing so far.

Code Listing 14-22

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Test that the document has been saved.
  If ActiveDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub; // Exit the script.
  Else
    // Set a variable for the active document.
    Set vCurrentDoc = ActiveDoc;
  EndIf
EndIf

// Test for a text selection in a paragraph.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = vCurrentDoc.TextSelection.Begin.Object;

// Make sure you are in a CodeTitle paragraph.
If vPgf.Name not= 'CodeTitle'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Find the last folder separator in the document's path.
Find String(DIRSEP) InString(vCurrentDoc.Name) Backward
  ReturnPos(vPos);
// Drop the filename from the path, leaving the separator.
Get String FromString(vCurrentDoc.Name) EndPos(vPos)
  NewVar(vDocFolder);

// Make a string from the chapter number.
New String NewVar(vListingFolder) Value(vCurrentDoc.ChapterNumber);

// If the size of the string is less than two characters,
// pad it with a zero.
If vListingFolder.Size = 1
  Set vListingFolder = '0' + vListingFolder;
EndIf
// Save the chapter number for later use.
Set vChapterNumber = vListingFolder;
// Add the "Listings" suffix.
Set vListingFolder = vListingFolder + 'Listings';

// Make an ESystem object.
New ESystem NewVar(vESystem);
// Make the folder.
Run vESystem.CreateDirectory DirName(vDocFolder+vListingFolder);
```

```
// Set a variable for the autonumber string.
Set vListingNumber = vPgf.PgfNumber;
// Make a string variable for the non-breaking hyphen.
New String NewVar(vNbhyphen) IntValue(21);
// Find the non-breaking hyphen in the number.
Find String(vNbhyphen) InString(vListingNumber) ReturnPos(vPos);
// Drop the front of the autonumber string.
Get String FromString(vListingNumber) StartPos(vPos+1)
  NewVar(vListingNumber);

// Pad the listing number with zeros, if necessary.
If vListingNumber.Size = 2
  Set vListingNumber = '0' + vListingNumber;
Else
  If vListingNumber.Size = 1
    Set vListingNumber = '00' + vListingNumber;
  EndIf
EndIf

// Assemble the code listing path.
Set vListingPath = vDocFolder+vListingFolder+DIRSEP+vChapterNumber+
  '_'+vListingNumber+'.fsl';

// Display the code listing path.
Display vListingPath;
```



G:\FrameScript2_1\Training\FrameScriptCrashCourse\13Listings\13_019.fsl

Since we now have the correct folder and text file path, let’s finish this part of the script by adding the code to create the text file. Add this code to the end of the previous listing.

Code Listing 14-23

```
// Add the code listing text file.
New TextFile NewVar(vListingObj) File(vListingPath);

// Close the text file.
Close TextFile Object(vListingObj);
```

At this point, the script creates an empty text file for the CodeTitle paragraph at the insertion point. We can put this script aside and move on to the next task.

Write the Code to the Text File

Our goal with this task is to write a single code listing to a text file. Initially, though, it will be easier to test the script by writing the listing to the Console window. We will be able to see the results without opening the text file. To develop this part of the code, we will work with the cursor in a CodeTitle paragraph. We will need to make a loop to process the all of the code paragraphs in a listing.

Code Listing 14-24

```
// Set a variable for the current document.
Set vCurrentDoc = ActiveDoc;

// Make a string list of all of the code paragraph formats.
New StringList NewVar(vCodePgfs);
Add Member('CodeFirst') To(vCodePgfs);
Add Member('CodeOne') To(vCodePgfs);
Add Member('Code') To(vCodePgfs);
Add Member('CodeLast') To(vCodePgfs);

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object; // CodeTitle paragraph

// Move to the next paragraph after CodeTitle.
Set vPgf = vPgf.NextPgfInFlow;
// Begin a loop.
Loop While(vPgf) LoopVar(n) Init(1) Incr(1)
  // See if the paragraph is in the list of code paragraph formats.
  Find Member(vPgf.Name) InList(vCodePgfs) ReturnStatus(vFound);
  If vFound
    // If it is, write the text to the Console.
    Write Console vPgf.Text;
  Else
    // If not, exit the loop.
    LeaveLoop;
  EndIf
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

Each paragraph is tested against the paragraph format names in the string list. This is a convenient way to compare one string against many. If you need to add or subtract paragraphs to compare, you simply modify the members of the string list. Because the paragraphs in each listing are consecutive, we continue writing the paragraph text until a non-Code paragraph is found. This causes the **LeaveLoop** command to execute.

If our final script, we will have to write each line of each paragraph. We can do this by getting a list of strings in the paragraph. Like the **RunSelectedCode** script, we will also need to convert non-breaking spaces to regular spaces.

Code Listing 14-25

```
// Set a variable for the current document.
Set vCurrentDoc = ActiveDoc;

// Make a string list of all of the code paragraph formats.
New StringList NewVar(vCodePgfs);
Add Member('CodeFirst') To(vCodePgfs);
Add Member('CodeOne') To(vCodePgfs);
Add Member('Code') To(vCodePgfs);
Add Member('CodeLast') To(vCodePgfs);

// Make a variable for the non-breaking space character.
New String NewVar(vNbsp) IntValue(17);
```

```
// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object; // CodeTitle paragraph

// Move to the next paragraph after CodeTitle.
Set vPgf = vPgf.NextPgfInFlow;
// Begin a loop.
Loop While(vPgf) LoopVar(n) Init(1) Incr(1)
  // See if the paragraph is in the list of code paragraph formats.
  Find Member(vPgf.Name) InList(vCodePgfs) ReturnStatus(vFound);
  If vFound
    // If it is, get a list of strings in the paragraph.
    Get TextList InObject(vPgf) String NewVar(vTextList);
    // Loop through the strings.
    Loop While(m <= vTextList.Count) LoopVar(m) Init(1) Incr(1)
      Get Member Number(m) From(vTextList) NewVar(vString);
      // Replace non-breaking spaces with regular spaces.
      Get String FromString(vString.TextData) NewVar(vString)
        ReplaceAll(vNbsp) With(' ');
      // Write the string to the Console.
      Write Console vString;
    EndLoop
  Else
    // If not, exit the loop.
    LeaveLoop;
  EndIf
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

You should notice that the **LoopVar** variable on the inner loop (**m**) is different from the one on the outer loop (**n**). The reason we use a **LoopVar** on the outer loop is because we need to add a blank line before each new paragraph in the listing, *except the first one*. We can use the value of **n** to test which paragraph we are writing.

Code Listing 14-26

```
// Set a variable for the current document.
Set vCurrentDoc = ActiveDoc;

// Make a string list of all of the code paragraph formats.
New StringList NewVar(vCodePgfs);
Add Member('CodeFirst') To(vCodePgfs);
Add Member('CodeOne') To(vCodePgfs);
Add Member('Code') To(vCodePgfs);
Add Member('CodeLast') To(vCodePgfs);

// Make a variable for the non-breaking space character.
New String NewVar(vNbsp) IntValue(17);

// Set a variable for the paragraph containing the insertion point.
Set vPgf = TextSelection.Begin.Object; // CodeTitle paragraph
```

```
// Move to the next paragraph after CodeTitle.
Set vPgf = vPgf.NextPgfInFlow;
// Begin a loop.
Loop While(vPgf) LoopVar(n) Init(1) Incr(1)
  // See if the paragraph is in the list of code paragraph formats.
  Find Member(vPgf.Name) InList(vCodePgfs) ReturnStatus(vFound);
  If vFound
    // If the paragraph is not the first one, write a blank line.
    If n > 1
      Write Console '';
    EndIf
    // Get a list of strings in the paragraph.
    Get TextList InObject(vPgf) String NewVar(vTextList);
    // Loop through the strings.
    Loop While(m <= vTextList.Count) LoopVar(m) Init(1) Incr(1)
      Get Member Number(m) From(vTextList) NewVar(vString);
      // Replace non-breaking spaces with regular spaces.
      Get String FromString(vString.TextData) NewVar(vString)
        ReplaceAll(vNbsp) With(' ');
      // Write the string to the Console.
      Write Console vString;
    EndLoop
  Else
    // If not, exit the loop.
    LeaveLoop;
  EndIf
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
```

This code is ready to go, except that we must modify it to write to the code listing text file instead of the Console window. To do this, replace **Write Console** with **Write Object(vListingObject)** in the code above. Now we can combine the code above with the script that creates the text file. We will use a subroutine to make the combined script more organized.

Code Listing 14-27

```
// Test for an active document.
If ActiveDoc = 0
  MsgBox 'There is no active document.    ';
  LeaveSub; // Exit the script.
Else
  // Test that the document has been saved.
  If ActiveDoc.Name = ''
    MsgBox 'Please save the document and rerun the script.    ';
    LeaveSub; // Exit the script.
  Else
    // Set a variable for the active document.
    Set vCurrentDoc = ActiveDoc;
  EndIf
EndIf
```

```
// Test for a text selection in a paragraph.
If vCurrentDoc.TextSelection.Begin.Object.ObjectName not= 'Pgf'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Set a variable for the paragraph containing the insertion point.
Set vPgf = vCurrentDoc.TextSelection.Begin.Object;

// Make sure you are in a CodeTitle paragraph.
If vPgf.Name not= 'CodeTitle'
  MsgBox 'Click in a CodeTitle paragraph and rerun the script.    ';
  LeaveSub;
EndIf

// Make a string list of all of the code paragraph formats.
New StringList NewVar(vCodePgfs);
Add Member('CodeFirst') To(vCodePgfs);
Add Member('CodeOne') To(vCodePgfs);
Add Member('Code') To(vCodePgfs);
Add Member('CodeLast') To(vCodePgfs);

// Make a variable for the non-breaking space character.
New String NewVar(vNbsp) IntValue(17);

// Find the last folder separator in the document's path.
Find String(DIRSEP) InString(vCurrentDoc.Name) Backward
  ReturnPos(vPos);
// Drop the filename from the path, leaving the separator.
Get String FromString(vCurrentDoc.Name) EndPos(vPos)
  NewVar(vDocFolder);

// Make a string from the chapter number.
New String NewVar(vListingFolder) Value(vCurrentDoc.ChapterNumber);

// If the size of the string is less than two characters,
// pad it with a zero.
If vListingFolder.Size = 1
  Set vListingFolder = '0' + vListingFolder;
EndIf
// Save the chapter number for later use.
Set vChapterNumber = vListingFolder;
// Add the "Listings" suffix.
Set vListingFolder = vListingFolder + 'Listings';

// Make an ESystem object.
New ESystem NewVar(vESystem);
// Make the folder.
Run vESystem.CreateDirectory DirName(vDocFolder+vListingFolder);
```

```
// Set a variable for the autonumber string.
Set vListingNumber = vPgf.PgfNumber;
// Make a string variable for the non-breaking hyphen.
New String NewVar(vNbhyphen) IntValue(21);
// Find the non-breaking hyphen in the number.
Find String(vNbhyphen) InString(vListingNumber) ReturnPos(vPos);
// Drop the front of the autonumber string.
Get String FromString(vListingNumber) StartPos(vPos+1)
  NewVar(vListingNumber);

// Pad the listing number with zeros, if necessary.
If vListingNumber.Size = 2
  Set vListingNumber = '0' + vListingNumber;
Else
  If vListingNumber.Size = 1
    Set vListingNumber = '00' + vListingNumber;
  EndIf
EndIf

// Assemble the code listing path.
Set vListingPath = vDocFolder+vListingFolder+DIRSEP+vChapterNumber+
  '_'+vListingNumber+'.fsl';

// Add the code listing text file.
New TextFile NewVar(vListingObj) File(vListingPath);

// Run a subroutine to write the code listing to the text file.
Run WriteCode;

// Close the text file.
Close TextFile Object(vListingObj);
```

```
Sub WriteCode
//
// Move to the next paragraph after CodeTitle.
Set vPgf = vPgf.NextPgfInFlow;
// Begin a loop.
Loop While(vPgf) LoopVar(n) Init(1) Incr(1)
  // See if the paragraph is in the list of code paragraph formats.
  Find Member(vPgf.Name) InList(vCodePgfs) ReturnStatus(vFound);
  If vFound
    // If the paragraph is not the first one, write a blank line.
    If n > 1
      Write Object(vListingObj) '';
    EndIf
    // Get a list of strings in the paragraph.
    Get TextList InObject(vPgf) String NewVar(vTextList);
    // Loop through the strings.
    Loop While(m <= vTextList.Count) LoopVar(m) Init(1) Incr(1)
      Get Member Number(m) From(vTextList) NewVar(vString);
      // Replace non-breaking spaces with regular spaces.
      Get String FromString(vString.TextData) NewVar(vString)
        ReplaceAll(vNbsp) With(' ');
      // Write the string to the Console.
      Write Object(vListingObj) vString;
    EndLoop
  Else
    // If not, exit the loop.
    LeaveLoop;
  EndIf
  // Move to the next paragraph in the flow.
  Set vPgf = vPgf.NextPgfInFlow;
EndLoop
//
EndSub
```

The remaining task is to make the script work for all of the code listings in the document. We will keep the current script as a separate script because it can be used to extract a single listing. This will be useful for last minute changes to code listings.

Before moving to the final task, let's modify the current script to add some copyright information to the top of each script. We will add these lines right after the line that creates the text file:

```
// Add the code listing text file.
New TextFile NewVar(vListingObj) File(vListingPath);
```

Code Listing 14-28

```
// Add copyright information to the code listing file.
Write Object(vListingObj) '// FrameScript Crash Course Listing.';
Write Object(vListingObj) '// Copyright 2002, Carmen Publishing. '+
  'All rights reserved.';
Write Object(vListingObj) '//';
Write Object(vListingObj) '// Unauthorized duplication prohibited.';
Write Object(vListingObj) '//';
Write Object(vListingObj) '// For more information, contact '+
  'rick@frameexpert.com
Write Object(vListingObj) '// or see www.frameexpert.com';
```

Because there will be this header content in the code listing file, we should remove the test for the first paragraph so the script will add a space before all of the paragraphs. To do this, change this

```
// If the paragraph is not the first one, write a blank line.
If n > 1
  Write Object(vListingObj) '';
EndIf
```

to this:

Code Listing 14-29

```
// Write a blank line before the paragraph.
Write Object(vListingObj) '';
```

Loop Through all of the Code Listings in the Document

This task is straightforward, especially compared to the previous tasks. We need a simple loop that will process the paragraphs in the main flow in document order.

Code Listing 14-30

```
// Set a variable for the active document.
Set vCurrentDoc = ActiveDoc;

// Set a variable for the first paragraph in the main flow.
Set vPgf = vCurrentDoc.MainFlowInDoc.FirstPgfInFlow;
// Loop through the paragraphs in the main flow.
Loop While(vPgf)
  // See if the paragraph is a CodeTitle paragraph.
  If vPgf.Name = 'CodeTitle'
    // Run a subroutine to process the code paragraphs.
    Run ProcessListing;
  EndIf
  // Move to the next paragraph in the flow.
EndLoop
```

The **ProcessListing** subroutine will be made using code from the previous script. We will leave it to you to figure out how to put it together; the finished script is on the CD. Make sure you save a copy of the previous script for extracting individual code listings.

Happy scripting!