FrameScript

Version 3.0

Basics

ELMSOFT INC.
7954 Helmart Drive
Laurel Maryland 20723
USA

Copyright © 1997-2003 ElmSoft, Inc. All rights reserved.

ElmSoft, Inc. ("ElmSoft") and its licensors retain all ownership rights to the FrameScript computer program and other computer programs offered by ElmSoft (hereinafter collectively called "ElmSoft Software") and their documentation. Use of ElmSoft Software is governed by the license agreement accompanying your original media. The ElmSoft Software source code is a confidential trade secret of ElmSoft. You may not attempt to decipher, decompile, develop, or otherwise reverse engineer ElmSoft Software, or knowingly allow others to do so. Information necessary to achieve the inter operability of the ElmSoft Software with other programs may be available from ElmSoft upon request. You may not develop passwords or codes or otherwise bypass the security features of ElmSoft Software.

This manual, as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of such license. 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 ElmSoft. ElmSoft assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of ElmSoft.

Please remember that existing artwork or images that you may desire to scan as a template for your new image may be protected under copyright law. The unauthorized incorporation of such artwork or images into your new work could be a violation of the rights of the author. Please be sure to obtain any permission required from such authors.

FrameScript and ElmSoft are trademarks of ElmSoft.

Adobe, the Adobe logo, Acrobat, Acrobat Exchange, Adobe Type Manager, ATM, Display PostScript, Distiller, Exchange, Frame, FrameMaker, FrameMaker+SGML, FrameMath, FrameReader, FrameViewer, FrameViewer Retrieval Tools, Guided Editing, InstantView, PostScript, and SuperATM are trademarks of Adobe.

IN NO EVENT WILL APPLE, ITS DIRECTORS, OFFICERS, EMPLOYEES, OR AGENTS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE APPLE SOFTWARE EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

The following are copyrights of their respective companies or organizations:

The following are trademarks or registered trademarks of their respective companies or organizations:

Apple, AppleLink, AppleScript, AppleTalk, Balloon Help, Finder, ImageWriter, LaserWriter, PowerBook, QuickDraw, QuickTime, TrueType, XTND System and Filters, Macintosh, and Power Macintosh are used under license / Apple Computer, Inc.

Microsoft, MS-DOS, Windows / Microsoft Corporation

Sun Microsystems, Sun Workstation, TOPS, NeWS, NeWSprint, OpenWindows, SunView, SunOS, NFS, Sun-3, Sun-4, Sun386*i*, SPARC, SPARCstation / Sun Microsystems, Inc.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Written and designed at Elmsoft, Inc., 7954 Helmart Drive, Laurel,MD 20723, USA

For civilian agencies: Restricted Rights Legend. Use, reproduction, or disclosure is subject to restrictions set forth in subparagraphs (a) through (d) of the commercial Computer Software Restricted Rights clause at 52.227-19 and the limitations set forth in ElmSoft's standard commercial agreements for this software. Unpublished rights reserved under the copyright laws of the United States. The contractor/manufacturer is Elmsoft, Inc. , 7954 Helmart Drive, Laurel,MD 20723, USA.

Table of Contents

1 Introduction - 1

FrameScript, FrameMaker and the FDK - 1
What is a script? - 1
Script usage - 2
Conventions - 3

2 Elements of FrameScript - - - - - - - - - - - 5

Format of a Script - 5
Standard Scripts - 5
Event Scripts: - 5
Format of an event script - 6
Initial Script - 6
Format of FrameScript commands - 7
Comments - 7
Include Directive - 8
Data Types - 8
Standard Object Information - 15
Constants - 16
    Integer Constants. - 16
    Real constants. - 16
    Metric constants. - 16
    String constants - 17
Predefined Named Constants - 17
Operators - 18
Identifiers - 20
Variables - 20
Variable Scope - 22
Objects and Properties - 22
Arrays and Collections - 24
    EArray - 24

EVector - 24
ECollection - 25
Expressions - 25

3 Basic Commands - 27
Set command - 27
GlobalVar command - 28
If, Else, ElseIf, EndIf - 28
Loop - 29
Loop--ForEach - 30
LeaveLoop - 30

4 Creating and Deleting Data - 33
Overview - 33
New Commands - 33
New Integer - 34
New Real - 34
New Metric - 34
New String - 35
New Object - 35
Delete Var - 36
Delete Object - 37

5 Built-in Dialogs - 39
DialogBox commands - 39
DialogBox--ChooseFile - 39
DialogBox--Prompt for Data - 40
DialogBox--ScrollBox - 41
DialogBox--Multi-Edit - 43
MsgBox - 47
Display - 48

6 Subroutines and Functions - 51
Overview - 51

Basic Subroutines - - - - - - - - - - 52
Local Variables - - - - - - - - - - 52
Passing Arguments to Subroutines - - - - - - - - - - 53
Returning values from Subroutines - - - - - - - - - - 54
Sub Command - - - - - - - - - - 55
Run Command - - - - - - - - - - 56
Local Command - - - - - - - - - - 56
LeaveSub - - - - - - - - - - 57
User Functions - - - - - - - - - - 58
Function Declaration Command - - - - - - - - - - 60
Calling a Function - - - - - - - - - - 61
Sub/Function Expressions - - - - - - - - - - 61

7 Modules - - - - - - - - - - 63

Introduction - - - - - - - - - - 63
$Main - - - - - - - - - - 63
Using Modules - - - - - - - - - - 63
Using a SubVar - - - - - - - - - - 64
Using a ScriptVar - - - - - - - - - - 64
Using a LibVar - - - - - - - - - - 65
Using a String - - - - - - - - - - 65
Summary - - - - - - - - - - 66
Some Examples - - - - - - - - - - 66
New LibVar - - - - - - - - - - 67
New ScriptVar - - - - - - - - - - 68
New SubVar - - - - - - - - - - 69

8 Standard Script Library - - - - - - - - - - 71

Introduction - - - - - - - - - - 71
DocUtils - - - - - - - - - - 71
Function DocIsAlreadyOpen - - - - - - - - - - 71
Function ForAllDocsInBook - - - - - - - - - - 72
Function GetCellXY - - - - - - - - - - 72
Sub AddParaToCellXY - - - - - - - - - - 73
Dual Select Dialog - - - - - - - - - - 73
Function DlgStringDualSelect - - - - - - - - - - 74

Database Utilities - - - - - 74
Function DlgDB_Connection - - - - - 74
Function DBTableExists - - - - - 75

9 Events - - - - - 77
Overview - - - - - 77
Event EventName - - - - - 77
Predefined Events - - - - - 78
Hypertext Events - - - - - 78
CanTerminate Function - - - - - 79

10 Script Commands - - - - - 81
Install Script - - - - - 81
Uninstall Script - - - - - 83
Exec Compile Command - - - - - 83
Exec Script Command - - - - - 84

11 Working with Text Files - - - - - 85
New Textfile - - - - - 85
Open Textfile - - - - - 85
Read command - - - - - 86
Write Command - - - - - 87
Close Textfile - - - - - 88

12 List Commands - - - - - 89
Introduction - - - - - 89
New List Data types - - - - - 89
Add Member - - - - - 90
Find Member - - - - - 91
Get Member - - - - - 92
Remove Member - - - - - 93
Replace Member - - - - - 94
Sort command - - - - - 95

13 Miscellaeous Commands - - - - - - - - - - 97

Find String - 97
Get String- 98
Exec Wait Command - 99

14 List of Error Messages - - - - - - - - - - 101

15 Common Script Errors- - - - - - - - - - - 105

Reserved Words - 105
EndIf, EndLoop, EndSub, EndEvent- 105
Logic Errors - 105
Check Error Codes- 105
Read-only variables - 106
Run Away Scripts - 106

16 Frame Architecture - - - - - - - - - - - - 107

Object Lists - 107
Session Object - 107
Book Object - 108
Document Object- 108
Body Page - 110

Chapter 1

Introduction

FrameScript, FrameMaker and the FDK

FrameScript is a high level, user-oriented, scripting (or macro) language designed to work with FrameMaker versions 5.5.6. 6.0 and 7.0. FrameScript allows users to customize their FrameMaker product with simple script commands, to create new functions for their FrameMaker product, to automate many current functions into one script command.

FrameMaker is a popular document publishing software system. Since it is a mainstream product, its goal is to appeal to a large client base. Like any large software vendor, Adobe has to carefully choose which ’features’ to put into each new release of the product. If it doesn’t put enough useful features in, it might lose customers. If it puts in too many, which appeal to only a small market segment, it will be accused of software bloat. The problem is that someone’s bloat is someone else’s need.

To allow users to customize the FrameMaker product, Adobe provides the Frame Developer’s Kit (FDK), which allows programmers access to FrameMaker’s capabilities. It requires the use of the Microsoft Visual C++ compiler plus the services of an experienced, expensive computer programmer. The FDK gives programmers the power to customize FrameMaker. FrameScript now brings that power to FrameMaker users instead of just programmers.

The script language itself is geared toward high-level users as opposed to programmers. The commands are simplified with many options but with defaults for almost everything. In the simplest case, a FrameScript script is just a sequence of commands in a simple text file; you can use FrameMaker itself to produce these script files (save as text). To run the script, the user selects the FrameScript->Run Script menu item, and chooses the script file from the resulting dialog box.

You may also install a script. In this case, the script (with a user defined label) appears on the FrameScript menu. The user can select a menu item to run the script.

There is also an initial script, which, optionally, runs when the FrameMaker product starts. You may use this 'initial script' to make general customizations for the product and to automatically install other predefined scripts.

Finally, you may also develop 'event scripts'. These are scripts which stay around and process FrameMaker events. In event scripts you may create your own custom menus items, have script commands run whenever a user opens a certain type of document (and even cancel the operation if it suits you), have script commands run before or after documents or books are closed or saved, plus many other events.

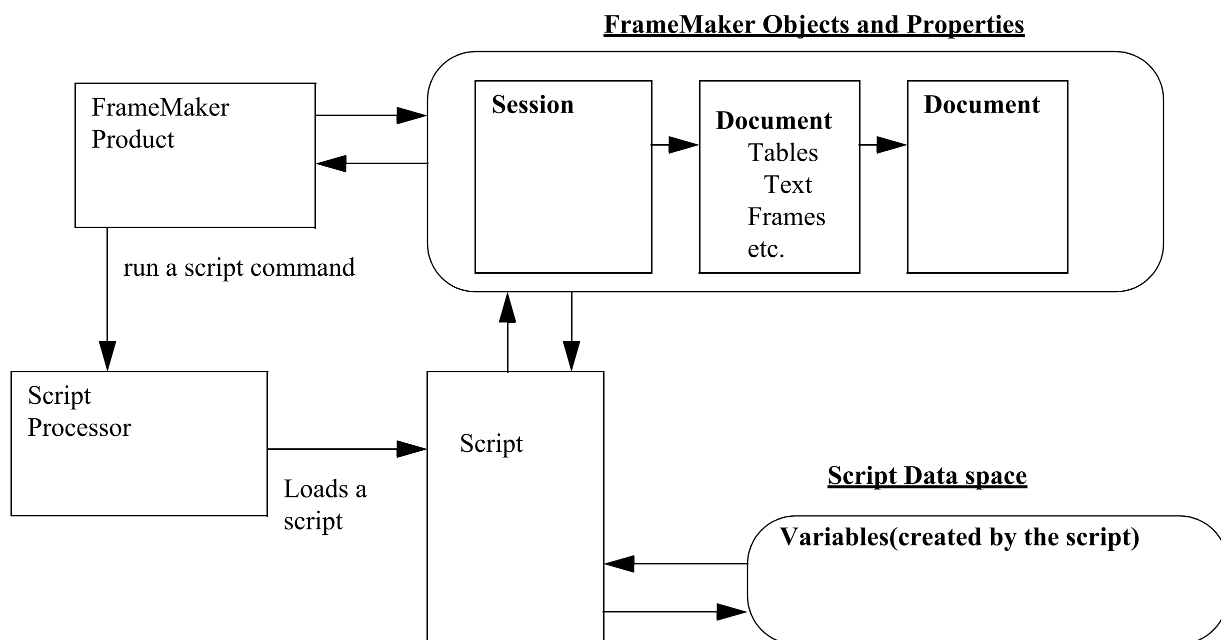
What is a script?

Standard Scripts

In the simplest terms, a FrameScript script is just a text file containing a set of FrameScript commands. The basic unit of a FrameScript script is the command. Commands are executed one at a time until it reaches the end of the script. You may control the execution sequence of commands by using control commands. These allow you to conditionally (based on run time conditions) execute a set of commands or repeatedly execute the same sequence of commands also based on run time conditions.

FrameScript runs under the FrameMaker product. It is completely dependent on FrameMaker for most (but not all) of its functionality. When the FrameMaker product starts it creates a session object. This session object has a list of properties associated with that session. Among these properties are: a list of the open documents (initially empty) in the session, a list of the open books (initially empty) in the session, a list of menus and commands, version number and so on. Whenever FrameMaker opens a document or book, it creates a large number of objects (with their associated properties) under that document or book. Each FrameScript script has access to all these objects and their properties. You may create new objects and delete old ones. You may modify the properties of these objects (if they are updatable). All this FrameMaker information is inherently part of each FrameScript script.

Each FrameScript script has its own data space to use as it wishes. In this data space, a script can create its own data names. These are called variables (because you may change the value of any of these data names whenever you wish). A data space is created when a script starts and it is deleted when the script ends. The following diagram illustrates this process.



Script usage

Customization.
Scripts may be used to customize your FrameMaker session. You can replace FrameMaker functions with your own functions if you want some special action to be taken or some special options to be invoked. You can automatically set various options depending on some data item, such as the User name, to individually customize the properties of the session, including selectively removing or adding functions.

Add new functionality to FrameMaker.
You may write scripts to add new functions that are not currently available in FrameMaker. These functions may be of interest to only a small number of customers making it impractical for Adobe to put them in the mainstream product. They may be on the list for a future release. You can have them now with FrameScript.

Automate tasks.
Sometimes there are processes that are available in FrameMaker but consists of number of manual steps to perform. You can automate these processes with FrameScript.

Information reporting.
Most of the properties of objects in a FrameMaker system are invisible (or at least hidden behind a layer of dialog boxes). You can use FrameScript to generate various reports, such as information about all the documents in a book with the component properties listed.

Conventions

The typeface for the standard text in this reference manual looks like this text in the sentence that you are now reading. It gives explanations for the topic under consideration. Samples of FrameScript source examples look like the following:

```
Command Option(value) Option(value);
```

Properties when list within explanations look like this `propertyname`. Commands within the explanations look like this. Here is the **`SampleCommand`** command.

Command options that are enclosed within straight brackets ([]) are optional. Default values will be used for them when the command is executed. Make sure that the default value is the one you wish. Most of the time it will be. Items enclosed in curly braces ({}) means that you should select one (or sometimes more than one) of the items within. Most of the options are, as the name implies, optional. Any command that needs a document will use the currently active document if not otherwise specified.

Example:

```
[option(value)]
```

This next example shows an optional selection list.

Example:

```
[option({item1 item2 ... itemn})]
```

Chapter 2

Elements of FrameScript

Format of a Script

There are two types of FrameScript scripts: Standard Scripts and Event scripts.

Standard Scripts

Standard scripts are used for creating new functions and automating a set of current (and new) functions. These functions (started using the run menu command or using the scripts sub-menu) run to completion before returning control to FrameMaker and the user. All script data variables are destroyed when the script ends.

The following illustrates the format of a standard script.

```
Command options;
Command options;
.
.
.
Command options;
```

When FrameScript runs a standard script, it performs the following steps:

- It loads the script into memory.
- It creates a data space for the script.
- It executes each command one at a time until is reaches the last line of the script.

  The execution order may be altered by the control commands (if, loop, sub, etc.)
- It destroys the data space.
- It removes the script from memory.

The user can run a standard script in one of two ways: Using the Run command from the FrameScript menu or clicking on a menu item for installed standard scripts. The Run command prompts the user for a file name. The user selects the script file and then the script runs. A standard script may also be installed. When a standard script is installed (via the install menu item or the install FrameScript command), a menu item is created (under the FrameScript -> Scripts menu). This provides a convenient shortcut for executing commonly used scripts.

Event Scripts:

Event scripts are a special type of FrameScript script which, instead of running to completion and terminating, stays loaded in memory (its data space is also kept active) and processes FrameMaker events. These events include user defined menus, hypertext messages, and various file (document) actions, see FrameMaker Reference for a list of events. An event script is divided into a set of event routines. Each event routine, though it uses the same data space,

acts like a separate script to itself. An event routine will be run when the specially assigned event occurs. This type of script is useful for defining your own functions and even replacing standard FrameMaker commands with your own functions. See the Event Scripts section for more information.

Format of an event script

The format of an event script is as follows:

```
Event eventname1
    command options
    command options
    . . .
EndEvent

Event eventname2
    command options
    command options
    . . .
EndEvent
. . .
Event eventnamen
    command options
    command options
    . . .
EndEvent
```

FrameScript does not run an event script. These scripts are installed with the *install* menu command (or by the **install** script command, see FrameMaker Reference for information on the Install command) and uninstalled with the *uninstall* menu command (or the **uninstall** script command).

When FrameScript installs an event script, it performs the following steps:

- It loads the script into memory.
- It creates a data space for the script.
- It executes the `Initialize` event in the script (if present). Each command of this event is executed one at a time until is reaches the last line of the event (the EndEvent command).

  The execution order may be altered by the control commands (if, loop, sub, etc.)
- The script goes into a wait mode while it waits for the events particular to this script occur.

When FrameScript uninstalls an event script, it performs the following steps:

- It executes the `Terminate` event in the script (if present). Each command of this event is executed one at a time until is reaches the last line of the event (the EndEvent command).

  The execution order may be altered by the control commands (if, loop, sub, etc.)
- It destroys the data space.
- It removes the script from memory, removing any defined events, such as menus or notifications.

Initial Script

When FrameScript starts it will run an initial script, if specified in the customizing options. This initial script is a convenient place to install other scripts, via the install command. This initial script also allows you to create *Session* variables for other scripts to read (See “Variable Scope” on page 22.).

IMPORTANT: Remember that any global variables that remain after the Initial script terminates will become read-only session variables for every other script that runs during this FrameMaker session. This might produce a conflict with the names of variables these other scripts are trying to create or use. There is a configurable run-time option to prevent this, if you wish. See the User's Guide for how to do this.

Format of FrameScript commands

The command is the basic unit of a FrameScript script. Each script, whether it is a standard script or an event script, executes each command consecutively one at a time. Each command begins with a command name and ends with a semicolon. When FrameScript loads a script, it treats every occurance of a command name as the start of a new command. A common error is to accidentally use a reserved command name as a data name. The semicolon acts as a command terminator. It is not always necessary to include it, since the command next name will begin a new command (and terminate the current one), but there are some cases where you may omit the command name (such Set or Run) and FrameScript may not be able to determine when one command stops and another begins.

FrameScript has commands that support the standard programming/scripting concepts, such as sequence, If-Then-Else (and now ElseIf), Looping, Subroutines, Functions and Modules, as described below.

Many FrameScript commands have the following form:

```
CommandName Option1(expression1) . . . OptionN(expressionN);
```

A reserved command name is followed by a series of options with the value for that option enclosed within parentheses. Some options do not have values and are specified by the name of the option itself. Some commands have a large number of options. Some have only a few. You need only specify the options that you wish. Any unspecifed options will be assigned default values. Sometimes the command name alone is enough.

Examples:

Since there is no file name specified, this command will display a dialog box for the user to select a document to open. The selected document will then be opened.

```
Open Document;
```

This command will open the specified document.

```
Open Document File('testdoc.fm');
```

Comments

In addition to commands and script structure elements, a FrameScript script may contain comment entries. Comments allow you to document the script. These aren't necessary for the actual script execution; they are ignored when the script runs. But when a script gets beyond a few lines it's important to give yourself reminders of what it is suppose to do.

Comments can occur in two forms: the line form and the block form. Putting two slashes (//) together indicates that the rest of the current line is a comment and not to be processed during execution. The following is an example:

```
Open document File('testdoc.fm'); // this command opens the test document
```

Everything following the // is ignored during script execution.

Another way to do comments is the block method. This is more convenient for a comment that contains many lines. Block comments use the /* and */ to delimit the start of comment and end of comment. For example:

```
/* The following commands loops through all the paragraphs in the currently
   active document and counts the number of paragraphs with the
   pargraph format 'Heading1' */
Set gvCount = 0;
Loop ForEach (Pgf) In(ActiveDoc) LoopVar(gvPgf)
   if gvPgf.Name = 'Heading1'
      set gvCount = gvCount + 1;
   EndIf
EndLoop
MsgBox 'The number of heading1 paragraphs are '+gvCount;
```

Include Directive

You can include one script file inside another using the #Include directive. The syntax is as follows:

```
<#Include 'filename'>
```

The **filename** can be a full path name or just a filename where the file is somewhere in the search list. This file should be a text file containing FrameScript script commands.

When FrameScript encounters some text like this in a source file (text script file), FrameScript opens the text file and continues processing the script as if the text in the **filename** were pasted directly into the main source file. These include files can be nested, that is, one include file may also have include directives.

Data Types

FrameScript supports the following data types for variables and properties.

Table 1: FrameScript Data Types

Data type	Description
Integer	An integer is a whole number (no decimal point) ranging from -2147483647 to 2147483647. FrameMaker also uses this data type for True and False values (0 is False, 1-True)
Metric	A metric value is used for measurements. By itself, it is the number of points (there are 72 points in an inch) on the screen (or printer). You may specify other units (in constants) when it is more convenient, but the value of the metric number itself is always in points. The range for metric values is 0 to 32767. Decimal points are allowed. 23.5, 89.99 are all allowable metric values (there represent 23.5 andn 89.99 points respectively). Some properties ask for percentages (e.g color). These use metric values, 0 through 100. See metric constants for a way to specify these numbers in inches, centimeters, etc.
String	A string variable is a list of characters (Case-sensitive). A string can be of any length that fits in memory.

Table 1: FrameScript Data Types

Data type	Description
`Real`	A real is a number variable that allows decimal points. Real number have an extremely large range of values (-10^{4932} to 10^{4932}). Most likely you will use numbers in the middle of that huge range. There is only an estimated 10^{80} protons, neutrons and electrons in the entire known universe. Unless you are using FrameScript for astronomical research (an unlikely scenario), this numerical range should prove adequate. Numbers such as 456.12, 1.0, -56.99 are all valid real numbers.
`Object`	An object value represents a FrameMaker object (document, paragraph, table, etc.). Objects cannot be used in computations. Their purpose is to access properties of the object and to specify some action on the individual object in a FrameScript command. See the discussion below on Objects. This is a FrameMaker specific data type.
`EslObject`	An EslObject data type represents a FrameScript object (database, form, etc.). Like FrameMaker Objects, these cannot be used in computations. Their purpose is to access properties of the object and to specify some action on the individual object in a FrameScript command.

`TextLoc`	A TextLoc identifier represents a location in a FrameMaker text object. Text objects are those objects which contain text, such as paragraphs (`Pgf`) and textlines (`TextLine`). A `TextLoc` identifier has two parts, an object and an offset within that object. The object part is an object identifier (see above). The offset part is an integer specifying the distance in the object of the location. In FrameScript you can get the object part of a TextLoc by specifying a modifier on the identifier as follows: if `tloc` is a TextLoc identifier, then
	`tloc.Object` is the object part
	`tloc.Offset` is the offset part.
	`tloc.TextRange` gives a text range from the text loc (begin and end)
	`tloc.TextRange1` gives the text range for one text position ahead.
	You may also get all the text properties at the location represented by the TextLoc variable by using the .Properties property, as follow:
	`SET propList = tloc.Properties;`
	You can also get individual properties for the location represented by the TextLoc variable by using the property name. For example, the following gets the color object for the text location represented by tloc:
	`SET colorVar = tloc.Color;`
	NOTE:Text in a FrameMaker text object contains many items that are not just text strings. It also contains table anchors, footnote anchors, etc. You cannot always count the characters to get an accurate offset value. This is a FrameMaker specific data type.

Table 1: FrameScript Data Types

Data type	Description
`TextRange`	A TextRange identifier represents a range of text in a FrameMaker text object. A text range is just two `TextLoc`s together. If `trange` is a `TextRange` identifier then: `trange.Text` is a string representing the text within the text range. `trange.Begin` is a TextLoc representing the starting text location. `trange.End` is a TextLoc representing the ending text location. You may also access the TextLoc fields directly by: `trange.Begin.Object` is the object of the beginning text loc. `trange.Begin.Offset` is the offset of the beginning text loc. `trange.End.Object` is the object of the ending text loc. `trange.End.Offset` is the offset of the ending text loc. `trange.Properties` allows you to assign properties (text properties) to the range of test specified by this TextRange or it allows you to get the text properties from the beginning point in the range. The following sets the area of text to the properties in `PropList`. `SET trange.properties = PropList;` The Text property may be used to get the text in a text range or to replace the text in a text range. One of the most important TextRange properties is the TextSelection document property. This indicates the current insertion point (if both TextLocs are the same) or the text selection (if the textlocs have different values). This is a FrameMaker specific data type.
`Point`	A Point identifier represents a point value for a FrameMaker graphic. If `pPoint` is a point variable then: `pPoint.X` is the X offset value. `pPoint.Y` is the Yoffset value. This is a FrameMaker specific data type.

Table 1: FrameScript Data Types

Data type	Description
`Tab`	A Tab identifier represents a tab value for a FrameMaker paragraph. If `tTab` is a tab variable then: `tTab.X` is the offset value. `tTab.Type` is the location type. This value can be one of the following: `TabLeft` - Left Tab `TabRight` - Right Tab `TabCenter` - Center Tab `TabDecimal` - Decimal Tab `TabRelativeLeft` - Relative Left Tab (Format Change List Only) `TabRelativeRight` - Relative Right Tab (Format Change List Only) `TabRelativeCenter` - Relative Center Tab (Format Change List Only) `TabRelativeDecimal` - Relative Decimal Tab (Format Change List Only) `tTab.Decimal` is the Decimal Tab character `tTab.Leader` is a string giving the characters before the tab. This is a FrameMaker specific data type.
`ElementLoc`	An `ElementLoc` identifier represents a location in a FrameMaker document or book. A `ElementLoc` identifier has three parts, a parent element, a child element and an offset within that object. The offset part is an integer specifying the distance in the object of the location. In FrameScript you can get the parts of an `ElementLoc` by specifying a modifier on the identifier as follows: if `eloc` is an ElementLoc identifier, then `eloc.Parent` is the parent element. `eloc.Child` is the child element. `eloc.Offset` is the offset. This is a FrameMaker specific data type.

Table 1: FrameScript Data Types

Data type	Description
`ElementRange`	An `ElementRange` identifier represents a range of elements in a FrameMaker document or book. An element range is just two `ElementLoc`s together. If `erange` is an `ElementRange` identifier then: `erange.Begin` is an `ElementLoc` representing the starting element location. `erange.End` is an `ElementLoc` representing the ending element location. You may also access the `elementloc` fields directly by: `erange.Begin.Parent` is the parent element of the beginning `elementloc`. `erange.Begin.Child` is the child element of the beginning `elementloc`. `erange.Begin.Offset` is the offset of the beginning `elementloc` You may use the text property (`erange.text`) of an `ElementRange` variable or property to obtain the text within that element range. One of the most important `ElementRange` properties is the `ElementSelection` document property. This indicates the currently selected element (if both `elementloc`s are the same) or the range of elements (if the `elementloc`s have different values). This is a FrameMaker specific data type.
`Attribute`	An `Attribute` identifier represents a attribute value for an element in a FrameMaker document or book. An attribute is a structure with several sub values. These are the attribute name, the attribute value(s), the special allow as special indicator and the AllowAsSpecial flag. If `attr` is an `Attribute` identifier then: `attr.attrName` is the name of the attribute. `attr.attrValues` is this list of values for the attribute. This is a `StringList` type. `attr.AllowAsSpecial` is a boolean, value True-it is allowed a special case, False-not. This is a FrameMaker specific data type.

Table 1: FrameScript Data Types

Data type	Description
`AttributeDef`	An `AttributeDef` identifier represents a attribute definition value for an element definition in a FrameMaker document or book. An attribute definition is a structure with several sub values. These are the attribute definition name, the attribute value(s), the special allow as special indicator and the AllowAsSpecial flag. If `attrDef` is an `Attribute` identifier then: `attrDef.AttrDefName` is the name of the attribute definition. `attrDef.AttrDefChoices` is this list of possible choices for the attribute definition. This is a `StringList` type. `attrDef.AttrDefDefaults` is this list of default values for the attribute definition. This is a `StringList` type. `attrDef.AttrDefRequired` is a boolean, value True-if it is required, False-if not. `attrDef.AttrDefFlags` is a bitwise value: Possible values are: AfReadOnly The attribute value is read-only AfHidden The attribute value is hidden. `attrDef.AttrDefType` is the type of the attribute definition: Possible values are: AtString Any string value. AtStrings A stringlist AtChoices A value from a list of choices. AtInteger An integer value (possibly in a range (Min/Max) AtIntegers An IntList value (possibly in a range (Min/Max) AtReal A real value (possibly in a range (Min/Max) AtReals A list of real values (possibly in a range (Min/Max) AtUniqueID A string that uniquely identifies the element. AtUniqueIDREF A reference to a UniqueID attribute AtUniqueIDREFSOne or more references to UniqueID attributes. `attrDef.AttrDefRangeMin` is the minimum value for a range test (if present): `attrDef.AttrDefRangeMax` is the maximum value for a range test (if present): This is a FrameMaker specific data type.
`File`	A file identifier represents a text file on your computer system. This is a special type that is used only with the file commands.
`SubVar`	A subroutine identifier represents a subroutine in your script. You can use this as an easy way to access (RUN) a subroutine.
`LibVar`	A Library identifier represents a directory on your computer system, which can be used as a library of scripts. This gives you an easy way to access other scripts. LibVar variables have the following properties: LibPath The name of the directory used as a LibVar

Table 1: FrameScript Data Types

Data type	Description
`ScriptVar`	A Script identifier represents a script file on your computer system. This gives you an easy way to access (RUN) this script. ScriptVar variables have the following properties: FilePath The name of the File used as a ScriptVar
`TextItem`	A text item is a representation of an item found in a text object. A text item is a structure containing the following members. `TextOffset` - The offset in the textlocation of the item. `TextType` - The type of the item. See the FrameMaker Reference for more information. `TextData` - The actual data. This is a string for string types and an object for object types. This is a FrameMaker specific data type.
`Property`	A property of an object. A property has two parts, `PropName` - The name of the property `PropVal` - The value of the property. This is a FrameMaker specific data type.
`List Types`	FrameMaker and FrameScript has several data types that are lists of other data types. You can use the Get Member command to access the invidual members of a list. The members are numbered from 1 to size, where size is the number of members in the list. You can also access these members using the indexing operator ([]). These data types come usually as FrameMaker properties and were designed to return lists of information. These can be updated (using the Add Member, Replace Member, Remove Member commands) but these data types were not designed for efficient updating. See Chapter 12, “List Commands,” for more information on using lists in your scripts.
`StringList`	A list of strings. FrameMaker uses these for lists of font names, marker names, etc. You may use them for specifying the list of entries for a scroll list dialog or just to keep a list of names together. This is a FrameMaker specific data type.
`MetricList`	A list of metric values. FrameMaker uses this type whenever it wants a list of measurements, such as the column widths of a table. This is a FrameMaker specific data type. This is a FrameMaker specific data type.
`IntList`	It is a list of integer values. FrameMaker uses it for a list of objects in some cases. For example, the InCond property is an IntList. This is a FrameMaker specific data type.
`TabList`	It is a list of Tab values. FrameMaker uses it for lists of tabs in paragraph and paragraph formats. This is a FrameMaker specific data type.
`PointList`	It is a list of Point values. FrameMaker uses it for lists of vertices in some graphic objects, such as PolyLine. This is a FrameMaker specific data type.
`UIntList`	This is a rarely used list. It is a list of unsigned integer values. FrameMaker uses it for a list of f-code values.This is a FrameMaker specific data type.
`TextItemList`	A list of text item values. You get this list using the Get TextItems command. It represents the elements of a paragraph or group of paragraphs. This is a FrameMaker specific data type.

Table 1: FrameScript Data Types

Data type	Description
`PropertyList`	A list of properties for an object. This is retrieved with the **`Get TextProperties`** command or the `object.properties` modifier. Each item in the list is a property type. This is a FrameMaker specific data type.
`AttributeList`	A list of attributes. FrameMaker uses these to keep a all the attributes for an element. This is a FrameMaker specific data type.
`AttributeDefList`	A list of attribute definitions. FrameMaker uses these to keep a all the attribute definitionss for an element defintion. This is a FrameMaker specific data type.

Standard Object Information

There are some properties that apply to all data types. Some apply to certain kinds of objects and variables. The following table illustrates some of these special properties.

Table 2: Standard Object Properties

Property Name	Description
`dataname.Objectname`	Returns a string value containing the name of the data type. This applies to any variable or property. Possible values are ’String’ ’Integer’ ’Doc’
`dataname.Size`	Returns the physical size of the data item (length of the string for string variables or properties)
`dataname.Count`	Returns the number of items in a list object. This value is 1 for single data items.
`dataname.Properties`	Returns a complete property list for this object. The user may use this (for frame objects) to assign the properties of one object to another. The dataname must be a FrameMaker object variable or property.
`dataname.Valid`	True or False, depending on whether the variable contains a valid object. The dataname must be a FrameMaker object variable or property.
`dataname.Text`	The text of a text object. The dataname must be a FrameMaker text object (paragraph, textrange, or textline) variable or property. The result is a text string.
`dataname.IsObject`	True or False, depending on whether the variable is a FrameMaker object.
`dataname.Doc`	The document part of a Frame Object variable.
`dataname.Menu`	The Menu part of a Command, Menu or MenuItemSeparator Object variable.
`dataname.Page`	The Page object (Bodypage, reference page, master page) where the Frame Object resides. If the dataname is not an object or if it does not reside on a page (e.g. paragraph format), then this value is zero.
`dataname.IsParm`	If the dataname is a parameter passed to a subroutine, then the value is True, otherwise the value is False.

Constants

Constants are values that you specify directly in a FrameScript script (i.e. they are not in variables or properties). You may specify constants for several (but not all of the above data types).

Integer Constants.

An integer constant is just a whole number in the allowable integer range. The following are valid integer constants:

```
1, 25, -123
```

You may also specify a suffix (H or B) to indicate a hexidecimal value by using the H suffix and a binary value by using the B suffix, as follows:

```
10H - indicates a hexidecimal 10 which is 16 decimal.
10B - indecates a binary 10 which is 2 decimal.
```

Real constants.

A real constant is similar to an integer constant except that you can put in decimal places. The following are valid real constants:

```
123.33, 77.9, -12345.6789.
```

Metric constants.

Metric constants are similar to real constants except that you can specify the metric units as part of the constant. To represent a number as a point measurement, a metric constant looks just like a real constant. The value will be converted based on the context of its use. To specify units other than points, append a suffix onto the number *without any spaces intervening*. The following list shows the type of suffix and the corresponding units.

pts	Points
in	Inches (72 points per inch)
“	Inches (72 points per inch)
cm	Centimeters (~28 points per cm)
mm	Millimeters (~2.8 points per mm)
pica	Pica (12 points per pica)

The following are valid metric constants.

`345pts`	This represents 345 points (same as 345).
`5.5”`	This represents 5.5 inches (or 396 points)
`129.99cm`	This represents 129.99 centimeters (or 51.1 inches or 3684.75 points)

String constants

A string constant is a set of characters encloses in *single* quotation marks. Remember double quotation marks are used for metric values to represent inches. The value inside string constants are case sensitive (unlike most other things in FrameScript). The following are valid string constants:

```
'This is a string constant'
'This is also a string constant, but this is much longer than the first constant'
'1234.45'
```

IMPORTANT: Use the apostrophe (’) (not the slanted quote mark (’)) to enclose a string constant.

You can also use an integer value with an S suffix to specify a single character string. The integer value is the character code representation. For example,

```
65S - represents a single character string with the value ’A’.
```

since 65 is the character code for the letter A.

Predefined Named Constants

The following table presents a list of constand values identified by reserved names. You may use these identifiers in place of the values they reprersent.

Table 3: List of Named Constants

Global Variable Name	Description
BackSlash	This is a string value representing a backslash (\) character.You may use BKSL for short.
CharCR	This is a string value representing a carriage return character.
CharLF	This is a string value representing a linefeed character.
CharTAB	This is a string value representing a tab character.
ClientDir	A string value containing the directory name of the FrameScript product. This can be useful for locating files in the same directory as the FrameScript product.
ClientName	A string value, usually’fsl’. This is the name that FrameMaker uses to identify the FrameScript client. You can use this value in Hypertext markers to send messages to FrameScript scripts.
FslVersionMajor	This is an integer value indicating the current major version of the FrameScript program. For the current version this value should be 3.
FslVersionMinor	This is an integer value indicating the current minor version of the FrameScript program. For the current version this value should be 3.
InstallName	This is the name chosen when a script is installed.
InstalledScriptList	This is a stringlist value containing the names of all the currently (event types) installed scripts.
MainScript	A string value containing the name of the main (the one that you started) script.
NULL	This is a constant representing the NULL value.

Table 3: List of Named Constants

Global Variable Name	Description
`ObjEndOffset`	The last offset position in a paragraph. This is used for setting a text range to include the entire paragraph.
`ProductRevision`	The revision information for FrameScript. For example, R1. This goes with the FslVersionMajor and FslVersionMinor.
`Quote`	This value represents a single quote character. You can use this to insert a single quote in a string constant. **`e.g. set str = 'Can' + QUOTE + 't do it';`** The value of str will be Can’t do it.
`ThisScript`	A string value containing the name of the currently running script.This can be useful for doing call backs in subroutines.

Operators

Operators are tokens which allow you to perform computations and comparisons. The following are valid operators in FrameScript. Not all operators are valid will all data types.

+	Plus operator: This operator adds two numerical type data items together. This also works for string variables and acts to concatenate two strings together into one longer string.
-	Minus operator: This operator subtracts two numerical data items
*	Multiple operator: This operator multiplies two numerical data items.
/	Division operator: This operator divides the numerical expression on the right into the numerical expression on the left.
%	Modulus (remainder) operator: This operator divides the numerical expression on the right into the numerical expression on the left and returns the remainder.
=	Equal operator: This operator compares two expressions for equality. If they are equal, the result is `True`. Otherwise it is `False`.
>	Greater than operator: This operator compares two expressions. If the expression on the left has a greater value than the one on the right, the result evaluates to `True`. Otherwise it evaluates to `False`.
<	Less than operator: This operator compares two expressions. If the expression on the left has a lower value than the one on the right, the result evaluates to `True`. Otherwise it evaluates to `False`.
>=	Greater than or equal to operator: This operator compares two expressions. If the expression on the left has a value that is greater than or equal to the one on the right, the result evaluates to `True`. Otherwise it evaluates to `False`.
<=	Less than or equal to operator: This operator compares two expressions. If the expression on the left has a value that is lower than or equal to the one on the right, the result evaluates to `True`. Otherwise it evaluates to `False`.
not	Not operator: This operator reverses the effect of the following operator. E.g,. `not =` means not equal

and

And operator: This operator allows you to combine several comparative expressions into the same expression. Both expressions must be `True` for the entire expression to be `True`. E.g.

```
a = b and c = d
```

This expression evaluates to `True` if both `a` is equal to `b` `AND` `c` is equal to `d`.

or

And operator: This operator allows you to combine several comparative expressions into the same expression. If either expression is `True` the entire expression is `True`. E.g.

```
a = b or c = d
```

This expression evaluates to `True` if either `a` is equal to `b` `OR` `c` is equal to `d`.

. (dot)

The dot operator indicates a property. The left side is the property source and the right side is the property name. E.g.

```
Set gvName = gvObject.ObjectName
```

This expression returns the name (data type) of the value stored in the gvObject variable. This is especially useful for FrameMaker objects and EslObjects.

, (comma)

The comma operator indicates a join operation. An EVector is created or extended with the values in the operation. The join operation can continue with subsequent member as well. E.g.

```
Set gvVector = 1,2,3,4,5,6;
```

This expression creates an EVector with six members, the numbers one through 6.

Another example:

```
Set gvVector = (1,2),(3,4),(5,6)).
```

This creates an EVector consisting of 3 members, each of which is an EVector with 2 members.

[] (brackets)

Brackets are used for the indexing operation that allows access to members of arrays and pseudo-arrays. The format is as follows:

```
arrayValueType[expression];
```

The arrayValueType is any data type that allows indexing operations. This includes the built-in List types (StringList, IntList, etc.), the Array Objects (EArray, EVector and ECollection), and some EslObjects (such as EQuery and EForm). Usually the expression must evaluate to an integer value within the range of the data list. Sometimes (as with EForm, EQuery or ECollection) you can use other value types, such as Strings, as index values.

{} (braces)

Braces are used for passing arguments to functions. The brace indicate to FrameScript that it is a function call and not an Command Option. The format is as follows:

```
Set gvReturn = FuncCall{value1, value2, ..., valueN};
```

See the discussion of Functions and Subroutines for more information.

#& Arithmetic And operator: This operator allows you to do a bitwise AND operation on two integer variables. The result is a value in which there will be a 1 bit set in every position where a 1 bit occurs in both variables This is useful for certain Frame properties which are defined as bit-wise variables, such as ValidationFlags for an Element Object. E.g.

```
Set gvVF = gvElt.ValidationFlags #& ElemAttrValInvalid;
If gvVF
    Display 'Element has an invalid attr value';
EndIf
```

#| Arithmetic Or operator: This operator allows you to do a bitwise OR operation on two integer variables. The result is a value in which there will be a 1 bit set in every position where a 1 bit occurs in either variable .

Identifiers

Identifiers are the names you use for variables, subroutine (and function) names, property names, command names and option names. All FrameScript identifiers are case-insensitive. You can upper or lower case the names as as you wish. The identifier DocObject is the same as DOCOBJECT (or docobject or even DoCoBjEcT). An identifier may contain letters, digits and the Underscore (_) character and it must start with a letter. .

IMPORTANT: FrameScript and FrameMaker have a large number of predefined identifier names (see FrameMaker Reference). There are many object names and hundreds of property names, plus a selection of command and option names. When defining your own variable names you should make sure that your names do not conflict with a previously defined identifiers. Also, future versions of FrameScript may (and probably will) add more reserved names. Since most of these reserved names are real words (or combinations of words), when creating your own variable names, it is useful to supply a prefix (e.g. `vDocVar`) instead of using a name that means something. A convention has developed in the user community of using the letter v (short for variable) as a prefix to any variable name that you create in a script. When we, at ElmSoft, write scripts we use an extension to this convention. We use gv as a prefix for Global variables, lv for local variables, pv for parameters passed to subroutines and sv for variables that are part of a structure. Following this convention (or one similar to it) will reduce the probability that there will be naming conflicts.

Variables

When a FrameScript script starts, a Data Space is created for it. The data space contains the various types of user created variables. Variables are places to store data values. Unlike many other computer languages, you do not declare a variable to be of a certain data type. A type is assigned to a variable when the value is assigned to it. There are three kinds of variables in FrameScript, Session variables, Global variables, and Local variables. Session variables are created for you when a script starts. These are always present. You cannot create nor delete them. You can, however (except for those marked as Read-Only), change their values.

Table 4: List of Fixed Session Variables

Global Variable Name	Description
ErrorCode	An integer variable specifying an error code. Zero means that no error occurred. Other values indicate some type of error. Note that this value is never reset to zero by FrameScript. After you process an error condition, you should reset this value to zero.
ErrorMsg *(Read-Only)*	A string variable contain an explanation of the last error condition. This value corresponds to the errorcode variable explained above.
DeclareVarMode	An integer variable indicating whether automatic variable creation is allowed. If this variable is False (default), then global variables will be created automatically. If this value is True, then you must declare all global variables using the GlobalVar command, otherwise an error will occur.

IMPORTANT: Also note that any global variables still present when the "Initial Script" terminates (if this option is used) will become Read-Only Session variables for every other script in the system. There is an option to prevent this (See Users Guide for the list of options), if you do not want this to happen.

The second type of variable is the Global variable. This is probably the most common type of variable, especially for short scripts. You can create (and initialize) global variables with the **GlobalVar** command (See “GlobalVar command” on page 28), but this is optional. The default behavior is to create a new global variable automatically whenever you attempt to assign a value to a variable name that does not yet exist. For example

```
Set gvMyVar = 100; // This creates a global variable and assigns a value to it
Get Object Type(PgfFmt) Name('Heading1') NewVar(gvMyPgfVar);
   // This also creates a global variable called gvMyPgfVar
   //     (assuming that it does not already exist)
   //     because the Get Object command returns a value in the NewVar option.
GlobalVar gvMyVar2(100) gvMyStringVar('My String');
       // This creates two global variables and assigns values to them.
```

The third type of variable is the Local variable. Local variables are only present inside the subroutine or function in which they are created. When a subroutine or function terminates all of its local variables are destroyed. Local variables become more useful as scripts become larger. A large script may have many subroutines and user defined functions. Sometimes you may use a variable name in one subroutine that you also used in another, intending them to be different. A value may unexpectedly change. These are some of the most difficult problems to find and fix. It is good programming practice to use local variables inside subroutines or functions to limit the unplanned interaction between various parts of a script. Most of the sample scripts will show examples of using Local variables in subroutines.

Note: When you create a variable with the GlobalVar or Local commands (See “Local Command” on page 56) and do not provide an initial value, the value assigned will be 0 (zero).

Variable Scope

Any variable created in a script is accessible anywhere within that script. These variables are not accessible outside the script. That is, a variable created in one script will not interfere with a variable by the same name being created and used in another script. They exist in a different data space. There is one exception to this rule. Variables created in the initial script (see initial script) are global variables. Any script may read variables created within the initial script, but not change their value (or delete the variables themselves). Of course, if the initial script creates and then deletes a variable it will not be present for other scripts to access.

These global, read-only variables are useful for putting values for every script to use. This may be handy for customizing an installation (or a department within a larger institution) without having to put special variables into every script. Each department could put a department name in a variable and various True/False flags in the initial script and each department could still use a set of scripts without changing them.

Objects and Properties

There are two general types of values, Data Items and Objects. For data items (such as Strings and Integers), the value is stored with the variable itself. When the variable is destroyed, the data associated with it is also destroyed. Objects, on the other hand, are only references to something. The value in the variable is usually just an ID number (or sometimes a pair of numbers) that uniquely identifies something. Sometimes these are called handles or pointers. We use the word ’something’ here because an object can refer to different kinds of things. It might refer a set of functions. It might refer to a database or to a Form. FrameMaker objects can refer to documents, paragraphs, anchored frames, etc. Access to the actual object is via properties and methods (functions or subroutines).

When a variable with a non-object value is deleted, the value is also deleted. For example, if a variable contains a string value and you delete the variable (using the Delete Var command or a local variable when a subroutine ends), the variable itself goes away and the string value is also deleted. A variable with an object data type value works differently. When you delete this type of variable the variable itself goes away, but the object it references stays around. To delete an object, you must use the Delete Object command. This is one of the major differences between the value types. Another difference is when you assign a value to variable. When you assign a non-object value (such as a string), the whole value is copied. For example, if one variable contains a string value and you assign it to another variable a second copy of the string is made.

```
Set gvMyVar1 = 'My String';
Set gvMyVar2 - gvMyVar1;
```

Now there are two copies of the string ’My String’. However, when you assign the value from a variable with an object value only the ID is copied, not the object. A new object is not created. For example,

```
Set gvMyPgfObject = FirstPgfInDoc;
Set gvMyPgfObject2 = gvMyPgfObject;
```

The variable gvMyPgfObject2 contains the same ID (handle, pointer, whatever), but it does not create a new paragraph object.

Objects are usually created with the New command and removed with the Delete Object command. For example,

```
New Paragraph NewVar(gvPgfVar);
```

creates a new paragraph in the currently active document and places the object value in the gvPgfVar variable.

```
Delete Object(gvPgfVar);
```

This would delete the above paragraph.

The properties of an object are accessed by the dot (.) operator. The format is as follows:

```
gvMyObjectvar.propertyname
```

where gvMyObjectvar is a scriptwriter defined variable name that refers to an object and propertyname is the name of a property for the object type.

You can get an object representation into a variable as the return value in some commands, such as Open Document or New Document, or New Table, etc. The return value for these commands will be a variable representing a FrameMaker object (in the NewVar option). Some properties of objects also supply object variables. A FrameMaker document object contains many lists of FrameMaker objects that can be access those properties (see below).

If you omit the object variable and just specify an property name, FrameScript will still attempt to find the value. It will first check the properties of the FrameMaker Session object to see if that property name exists, then, failing that, it will look up properties from the currently active document (if any). If that also fails, it will look up properties from the currently active book (if any). If all this fails, FrameScript will assume that the identifier is a scriptwriter defined variable.

This means that you can specify the following session property names without specifying an object name.

```
UserName, FirstOpenDoc, FirstOpenBook
```

and many others (see Session properties)

It also means that you can specify various document properties without specifying a document object variable. This assumes that there is a currently active document available. Among these properties are:

```
FirstPgfFmtInDoc, FirstPgfInDoc, CurrentPage, DocIsViewOnly
```

The same can be applied to book properties for the currently active book.

> **IMPORTANT:** The properties `ActiveDoc` and `ActiveBook` are FrameMaker session properties that are available all of the time FrameMaker is running (see FrameMaker Reference). However, these values contain zero whenever there is no currently active document or book, respectively. Before using these objects and their properties you should check this value for zero. If ActiveDoc is zero and you try to access one of its properties, the command will fail. The same is true for the ActiveBook object. Also note that the ActiveBook value is zero when a FrameMaker document is in the current window, and the ActiveDoc is zero when the active FrameMaker window contains a book.

The property names for all FrameMaker objects are completely described in the FrameMaker Reference. There are some properties that apply to all data types. Some apply to certain kinds of objects and variables. The FrameMaker Reference contains a list of these special properties.

Arrays and Collections

EArray

The first of the FrameScript array types is the EArray. This is used for fixed length arrays. To use an EArray, you first create the array with the New command as follows:

```
New EArray NewVar(gvArray) Count(10);
```

This creates an array of 10 members, each of which as a NULL value. You can access and modify the members of this array using the indexing operator (brackets). For example,

```
Set gvArray[1] = 100;
Set gvArray[2] = 'String Data';
Loop InitVal(3) Incr(1) LoopVar(gvIdx) While(gvIdx <=10)
   Set gvArray[gvIdx] = 1000;
EndLoop
```

The above script fragment sets the first member of the array to the integer value 100, the second to the string value 'String Data' and the third through 10th members to the integer value 1000.

You can also create an EArray object using a utility function.

```
Set gvArray = eUtl.EArray{100,'String Data',1000,1000,1000,1000,1000,1000,1000,1000};
```

This will produce the same result as the above loop.

EVector

Another type of array is the EVector. This is similar to the EArray except that it is designed to be extensible. When you create the object, it is initially empty. You use the PushBack property to push values onto the end of the array. You can then access the members the same way you do an array (via the index operator).

```
New EVector NewVar(gvVector);
Set gvVector.PushBack = 100;
Set gvVector.PushBack = 'String Data';
Loop InitVal(3) Incr(1) LoopVar(gvIdx) While(gvIdx <=10)
   Set gvVector.PushBack = 1000;
EndLoop
```

The above creates a vector similar to the Array created in the previous example. You can access the members via the index, but you can keep on adding members to the end of a vector. You can even insert members, if you wish.

You can also create an EVector using a utiltity function.

```
Set gvVector = eUtl.EVector{100,'String Data',
   1000,1000,1000,1000,1000,1000,1000,1000};
```

A third way to create an EVector is with the Join operator (,).

```
Set gvVector = (100,'String Data',1000,1000,1000,1000,1000,1000,1000,1000);
```

ECollection

A third type of array is the ECollection. This is the most flexible, in the you can store members of various kinds, you can use it as a Linked List and you can access the members using other data types besides integers.

```
New ECollection NewVar(gvColl);
Set gvColl.PushBack = 'First Member Added';
Set gvColl.PushBack = 'Next Member Added';
Set gvMemberVar = gvColl.FirstMember;
Loop While(gvMemberVar)
   write console 'Member='+gvMemberVar;
   Set gvMemberVar = gvMemberVar.NextMember;
EndLoop
```

This code fragment creates a collection, pushes (adds) two members, then dumps the whole collecion to the console.

```
New ECollection NewVar(gvColl);
Set gvColl['John Smith'] = 1000;
Set gvColl['Jane Doe'] = 2000;
...
Set gvValue = gvColl['John Smith']; // Get the value for John Smith
```

This code fragment creates a collection and adds two members indexed by string values, then retrieves one of the values

See the EslObject Reference Manual for more information on the array objects

Expressions

An expression is any valid combination of constants, variables, properties, delimiters (parentheses), and operators. An expression can be as simple as one constant or one variable (or property name). It can also be a long sequence of tokens. Data types are converted automatically. The order of precedence for two unlike data items is as follows:

String, Real, Metric, Integer.

A string and anything else becomes a string. An integer or a metric combined with a real becomes a real, etc.

The following are examples of expressions.

`ActiveDoc`	A property name representing the currently active document.
`4.33”`	A metric constant indicating a value of 4.33 inches.
`a * b + c`	An arithmetic expression (assuming that a, b and c are variable), where a and b are multipled together then the value of c is added to the result.
`a * (b + c)`	This is similar to the above expression, except that the parentheses cause the values in b and c to be added before the value of a is multiplied.

`'This is' + ' an expression'`	This is an expression containing two strings with the plus (+) operator between them. The result is the string expression as follows: **`'This is an expression'`**
`ActiveDoc.SnapGridUnits + 100pts`	This is an expression which adds the value of the SnapGridUnits property of the currently active document and 100 points together.
`ActiveDoc.Name+' is the filename'`	This expression concatenates the file name of the currently active document with a string.

Chapter 3

Basic Commands

The following describes the basic FrameScript commands. The FrameMaker related commands are described in the FrameScript FrameMaker Command and Object Reference.

Set command

The **Set** command assigns the value of an expression to a variable name or property. The data type of the variable will be the same as the expression. If the assignment is to a property, the expression is converted to the type of data required by the property. If the assignment is to a variable name and the variable does not already exist, this command will create a new variable in the global data space, unless the **DeclareVarMode** session variable is set to True. In this case, the command will fail.

Format:

```
Set varname = expression;
set property = expression;
set varname.property = expression;
```

Table 5: Set Options

Option Name	Option Description
varname *(Required)*	The name of an existing variable or a new variable name. FrameScript will create a new variable if it doesn't already exist, unless the **DeclareVarMode** session variable is set to True.
property	When a property name appears as the target of a Set command, the property must be either a session property, a document property (of the currently active document) or a book property (of the currently active book).
varname.property	The property of an object.
expression	The expression to compute. This can be any valid arithmetic or string expression, using variables and/or properties.

Example:

This example sets the session's active document to the document represented by the variable docvar. Sets the page numbering style of the document represented by docvar and then performs an absolutely meaningless computation.

```
. . .
Set ActiveDoc = docvar;
Set docvar.pageNumStyle = PageNumAlphaLC;
set var1 = (10 * docvar.FirstPageNum + 100) / 5;
. . .
```

The **Set** command and the control commands are the exception to the above format. The **Set** command assigns a new value to a property or data variable. It will also create a data variable if it does not already exist. The control commands

(**If**, **Loop**, and **Run**) work with blocks of commands which terminate with a termination command (**EndIf**, **EndLoop**, **EndSub**). See FrameMaker Reference for more information.

GlobalVar command

The **GlobalVar** command creates or updates a variable in the global data space for a script. You may create or update more than one variable with one command.

Format:

```
GlobalVar VarName1[(expression)][ VarName2[(expression2)]] . . .
   { VarNameN[(expressionN)]];
```

Table 6: GlobalVar Options

Option Name	Option Description
VarNameI *(Required)*	The name of an existing variable or a new variable name. FrameScript will create a new variable in the global data space if it doesn't already exist. This will create a new global variable (if necessary) regardless of the value in the **DeclareVarMode** session variable.
expressionI	The expression to compute. This can be any valid arithmetic or string expression, using variables and/or properties. If the expression is not specified, then the variable is assigned the value of NULL.

If, Else, ElseIf, EndIf

The If command allows you to selectively execute a list of commands based on the evaluation of an expression.

Format:

```
If expression
   command list;
[ElseIf expression
   command list;]
[Else
   command list;]
EndIf
```

If the expression evaluates to a `True` value, then the first set of commands are executed. If the expression evaluates to `False`, then the second set of commands are executed (if present). The expression can be any type of valid expression. A numerical value is `False` if the value is zero, otherwise it is `True`. A String expression is `False` if the string is empty, otherwise it is True. If an object is zero, the expression is `False`, otherwise it is `true`. For list variables, if there are items in the list then it is True; if it is an empty list (no members) then it is `False`.

Be sure to use the `Endif` at the end of the command list otherwise the command list under the `If` command goes on until the end of the subroutine or script.

Example:

```
DialogBox Type(int) Title('Enter a non-zero number') NewVar(testnum)
If testnum = 0
   MsgBox 'Invalid number entered!!!';
Endif
```

Loop

The loop command allows to execute a sequence of commands repeatedly until a predefined set of conditions occur.

Format:

```
Loop [While(expression)] or [Until(expression)]
        LoopVar(loopvarname)
        InitVal (initialvalue) Incr(incrimentExpression)
...
EndLoop
```

Table 7: Loop While/Until Options

Option Name	Option Description
`While`	The loop continues as long as the expression evaluates to True. It stops when it evaluates to False. For this type of loop you must supply a While or Until option.
`Until`	The loop continues as long as the expression evaluates to False. It stops when it evaluates to True. For this type of loop you must supply a While or Until option.
`LoopVar` *(Required)*	This option identifies the loop variable. When the loop starts this variable is given the value in the `InitVal` option and it is incrimented by the `Incr` expression at each subsequent iteration of the loop.
`InitVal` *(Required)*	This option specifies the initial value of the loop variable the first time through the loop. When the loop starts, this is equivalent to the following command. `Set loopvarname = initialvalue;`
`Incr` *(Required)*	This option specifies the incriment value of the loop variable at each subsequent iteration of the loop. When the loop continues, this is equivalent to the following command. `Set loopvarname = loopvarname + incrimentExpression;`

When this version of the loop command starts, the loop variable (LoopVar) is set to the value defined in the `InitVal` option and the while and/or until expressions are tested. If these tests fail (the `While` evaluates to `False` or the `Until` evaluates to `True`), the script continues after the `EndLoop` command. Whenever the script reaches the `EndLoop` command, the Loop variable is incrimented by the value in `Incr` and the `While` and/or `Until` conditions are tested again.

Examples:

This example runs through the loop ten times, modifying the variable `gvIndex` with values 1 through 10.

```
Set gvCount = 0;
Loop LoopVar(gvIndex) While(gvIndex <= 10) InitVal(1) Incr(1)
   Set gvCount = gvCount + gvIndex;
EndLoop
write console 'The sum of the numbers between 1 and 10 is '+gvCount;
```

This example runs through the loop ten times, modifying the variable `gvIndex` with values 10 through 1.

```
Set gvCount = 0;
Loop Until(gvIndex < 1) LoopVar(gvIndex) InitVal(10) Incr(-1)
   Set gvCount = gvCount + gvIndex;
EndLoop
write console 'The sum of the numbers between 1 and 10 is '+gvCount;
```

Loop--ForEach

The loop command allows to execute a sequence of commands repeatedly for each member in the list type specified. This can loop through a directory once for each file name or through a list data type once for each member.

Format:

```
Loop ForEach(MemberType) In(List or directory) LoopVar(varname)
   commandlist;
EndLoop
```

Table 8: Loop ForEach Options

Option Name	Option Description
`ForEach` *(Required)*	This option identifies a member type. This should be File or Member.
`In`	This option identifies a list type variable or a directory name This is used in conjunction with the ForEach optionT
`LoopVar`	This option identifies the name of the loop variable. When the loop starts this variable is given the value of the first item in the list (file name or list member).

IMPORTANT: The ForEach option on the Loop command has many more options. These are described in the FrameMaker Reference.

Example 1:

Loop through each string in the string list variable and writes the string to the console.

```
New StringList NewVar(gvStrList) Value('String1') Value('String2') Value('String3');
Loop ForEach (Member) In(gvStrList) LoopVar(gvStringValue)
   write console 'String = '+gvStringValue;
Endloop
```

Example 2:

Loop through each file in the specified directory.

```
Loop ForEach (File) In('C:\FrameScript\') LoopVar(filename)
   write console 'File name='+filename;
Endloop
```

LeaveLoop

The **LeaveLoop** command jumps out of the current loop and starts the script again after the **EndLoop** command.

Format:

```
LeaveLoop;
```

Example:

This example searches the list of paragraph formats in the currently active document for the first one which uses autonumbering. If it finds one, it saves the paragraph format object into the `gvFoundPgfFormat` variable and does the LeaveLoop comand to skip the check all the other paragraph formats in the list.

```
Set gvFoundPgfFormat = NULL;
Loop ForEach(PgfFmt) In(ActiveDoc) LoopVar(gvPgfFormat)
   If gvPgfFormat.PgfIsAutoNum
      Set gvFoundPgfFormat = gvPgfFormat;
      LeaveLoop;
   EndIf
EndLoop
If gvFoundPgfFormat
   Display 'First format with auto numbering is '+gvFoundPgfFormat.Name;
Else
   Display 'No paragraph formats with auto numbering exist in this doc';
EndIf
```

Chapter 4

Creating and Deleting Data

Overview

Many times you can create data values as the result of expressions using the Set command. For example, the following commands create several different types of data and places the values into variables:

```
Set gvVar1 = 100;
Set gvVar2 = gvVar1*200 + 50;
Set gvVar3 = 'A string fragment';
Set gvVar4 = gvVar3 + '--Add some to the end';
Seg gvVar5 = 2"; // Create a metric value of 2 inches
Set gvVar6 = ActiveDoc;
```

The expressions on the right side of the equals (=) sign created a data value (the first two create integer values, the second two create string values, the fifth creates a Metric value and the sixth one evaluates to the ID of a FrameMaker object (the currently active document)) and assigns that value to a variable name (on the left side of the equals sign).

Some commands (such as Get String and Find String) also return values, which are assigned to variables.

Many times though, you will need to create data or objects using the New command.

New Commands

The New command is the standard way of creating a data items, data structures and objects in FrameScript. You can create any type of allowable data using this command. Described below are methods for creating simple data values. Most of these are here for completeness only because it is easier to use the Set command (describe above) for most of these. There are a few case, however, where you need to use these commands. These deal mostly with data conversion. The Set command will evaluate the expression and determine the resulting data value type from the expression. The New command allows you to determine the type of the resulting data. For example, the following two commands have expressions that evaluate to the same value.

```
Set gvVar1 = 100+300;
New Real NewVar(gvVar2) Value(100+300);
```

The first one creates an integer value, so the value assigned to the variable will be an integer type. The second command also evaluates to an integer value, but it converts it to a real value before assigning it to the variable.

IMPORTANT: The new command is also used for creating FrameMaker Objects (Paragraphs, anchored frames, documents, etc.), EslObjects (Forms, Database connections, Queries, etc.) and FrameMaker structures (TextRanges, TextLocs, etc.) See the FrameMaker reference and the EslObject reference for more information.

New Integer

This commands evaluates the expression in the Value option, converts it to an integer value (if possible), then assigns it to the variable identified by the NewVar option.

Format:

```
New Integer NewVar(varname) Value(expression);
```

Table 9: New Integer Options

Option Name	Option Description
Value	The expression to evaluate. If it is not an integer data type, it will be converted, if possible.
NewVar	The name of the variable to place the result.

Example:

```
set gvVar1 = 1.234;
set gvVar2 = var1;
New Integer NewVar(gvVar3) Value(gvVar1);
display 'var1='+gvVar1+' var2='+gvVar2+' var3='+gvVar3;
```

This will display the following:

var1=1.234 var2=1.234 var3=1

New Real

This commands evaluates the expression in the Value option, converts it to an real value (if possible), then assigns it to the variable identified by the NewVar option.

Format:

```
New Real NewVar(varname) Value(expression);
```

Table 10: New Real Options

Option Name	Option Description
Value	The expression to evaluate. If it is not a real data type, it will be converted, if possible.
NewVar	The name of the variable to place the result.

New Metric

This commands evaluates the expression in the Value option, converts it to an metric value (if possible), then assigns it to the variable identified by the NewVar option.

Format:

```
New Metric NewVar(varname) Value(expression);
```

Table 11: New Metric Options

Option Name	Option Description
`Value`	The expression to evaluate. If it is not a metric data type, it will be converted, if possible.
`NewVar`	The name of the variable to place the result.

New String

This commands evaluates the expression in the Value or IntValue options, converts it to an string value (if possible), then assigns it to the variable identified by the NewVar option.

Format:

```
New String NewVar(varname) Value(expression) or IntValue(int exspression);
```

Table 12: New String Options

Option Name	Option Description
`Value`	The expression to evaluate. If it is not a real data type, it will be converted, if possible.
`IntValue`	The integer value of the new single character string variable . This is a method for creating a single character string variable when the character is not printable.
`NewVar`	The name of the variable to place the result.

Example:

```
New String NewVar(gvCopyRightVar) IntValue(169);
Write Console 'FrameScript' + gvCopyRightVar;
```

This will write the following on the Frame console window:

FrameScript©

New Object

This commands evaluates the expression in the Value option (or IntValue and DocObject options), converts it to an FrameMaker Object value (if possible), then assigns it to the variable identified by the NewVar option.

Format:

```
New Object NewVar(varname) Value(expression);
or
New Object NewVar(varname) IntValue(int expression) DocObject(Fm DocObject);
```

The first format is unnecessary because there is no conversion from any other type to a FrameMaker object type. It is here for completeness only. The second format is occasionally necessary. In some cases, FrameMaker returns an integer value or an IntList of values which are really FrameObject IDs. For example, the InCond property returns a list of Condition Objects as a list of integers (IntList). In order to use these individual objects you must use this command to convert the integer into an object. Be sure to include the Document Object for a valid FrameMaker Object.

Table 13: New Object Options

Option Name	Option Description
Value	The value of the new variable. If it is not the same type as the target data, it will be converted, if possible.
IntValue	The integer value of the object value. This is a method for converting an integer into an object. Be sure that the integer is really an FrameMaker Object value before doing this or the resulting FrameMaker Object will be invalid.
DocObject	The document object (for Object types only).
NewVar	The name of the variable to hold the newly created text file object.

Example:

```
Set gvCondList = ActiveDoc.InCond;
If gvCondList.Count > 0
   New Object NewVar(gvCondObject) IntValue(gvCondList[1]) DocObject(ActiveDoc);
   Display 'First Condition Name is '+gvCondObject.Name;
Else
   Display 'The current insertion point has no conditions applied!';
EndIf
```

Delete Var

The **`Delete Var`** command deletes a variable from your FrameScript script data space. Ordinarily you wouldn't need to delete variables; the go away when the script terminates. Occasionally, though, when variables use a large amount of memory, such as large text fields or string lists, you should delete them when you don't need them anymore. This is especially true of Event Scripts, where the variables stay around until you delete them or the Frame Session terminates.

> **IMPORTANT:** Deleting a variable containing an Object value does not delete the object itself. It only deletes the variable containing the ID of the object. To delete an object (such as a paragraph, anchored frame, etc.), you need to use the Delete Object command.

Format:

```
Delete Var(varname);
```

Table 14: Delete Var Options

Option Name	Option Description
Var *(Required)*	The variable to delete.

Example
The following script deletes the large string variable:

```
. . .
set pgftextstring = FirstPgfInDoc.text;
. . .
Delete Var(pgftextstring);
. . .
```

Delete Object

The **Delete Object** command deletes a FrameMaker object from a document. It also can be used to delete EslObjects (such as Forms or Database queries). When an object is deleted, all the objects contained in that object are also deleted!

Format:

```
Delete Object(objectvar);
```

Table 15: Delete Object Options

Option Name	Option Description
Object *(Required)*	The object variable representing the object to delete.

Example
The following script deletes all the markers in the current document:

```
. . .
set mrkobj = FirstMarkerInDoc; // Get first marker in doc
Loop while (mrkobj)
   /* As each marker is deleted, its NextMarkerInDoc property
   ** becomes invalid, so it is necessary to get the next marker
   ** before deleting the current one.
   */
   SET delmrkobj = mrkobj;
   SET mrkobj = delmrkobj.NextMarkerInDoc;
   Delete Object(delmrkobj);
EndLoop
. . .
```

Chapter 5

Built-in Dialogs

The following describes the basic FrameScript commands. The FrameMaker related commands are described in the FrameScript FrameMaker Object Reference.

DialogBox commands

The **DialogBox** command displays several different styles of dialog boxes. These dialog boxes give you several ways to get information from the user.

DialogBox--ChooseFile

Format:

```
DialogBox Type(ChooseFile)
    [Directory(directoryName)] [Title(dialogTitle)]
    [Init(initialstring)] NewVar(varname) [Button(buttonvar)]
    [Mode({SelectFile OpenFile SaveFile OpenDirectory})];
```

This dialog box allows the user to select a file name to open or save, or a directory name.

Table 16: DialogBox ChooseFile Options

Option Name	Option Description
Type *(Required)*	This option identifies the dialog as a Choosefile dialog box.
Title	A string expression containing the title of the dialog box.
Directory	The initial directory to start the file selection.
Init	The initial value to put in the selection text line.
NewVar	The variable name to put the file name that the user selected.

Table 16: DialogBox ChooseFile Options

Option Name	Option Description
Button	The variable name to put the value of the button that the user pressed. The value returned in the variable will be one of the following: **OkButton** User pressed the Ok button. **CancelButton** User pressed the Cancel button.
Mode	This dialog box has several forms depending on the type of file selection you wish. The possible values are: SelectFile OpenFile SaveFile OpenDirectory

Example:

This sample displays a dialog box allowing the user to select a file name. The selection will start in the c:\maker directory. It will have the *.fm in the text box. After the user clicks on a button the value in the text box is copied to filevarname and the button pressed is copied to buttonvar.

```
. . .
DialogBox Type(Choosefile) Title('Please select a file name')
   Directory('c:\maker') Mode(OpenFile) Button(buttonvar)
   NewVar(filevarname) Init('*.fm');
if buttonvar = OKButton
   MsgBox 'The user pressed the OK button and selected this filename-'+filevarname;
EndIf
```

DialogBox--Prompt for Data

Format:

```
DialogBox Type({Int Metric String}) [Title(titlestring)]
   [Init(initialvalue)] [Units({Inch mm cm pica didot})]
   [NewVar(varname)] [Button(buttonvarname)];
```

This format allows the user to enter a single value of predefined type.

Table 17: DialogBox Enter Simple Data Options

Option Name	Option Description
Type *(Required)*	This option identifies the dialog as a simple data input type. The allowable values are: Int - Input an integer. Metric - Input a metric value. String - Input a string value.
Title	A string expression containing the title of the dialog box.
Init	The initial value to put in the text box.

Table 17: DialogBox Enter Simple Data Options

Option Name	Option Description
`Units`	If the type value is metric, you may specify the default units of data the user will enter if there are no units specified on the input line. You may specify one of the following values: INCHES POINTS CM MM PICA DIDOT CICERO
`NewVar`	The variable name to put the data that the user entered.
`Button`	The variable name to put the value of the button that the user pressed. The value returned in the variable will be one of the following: **OkButton** `User pressed the Ok button.` **CancelButton** `User pressed the Cancel button.`

Example:

This example displays a dialog box prompting the user to enter a metric value.

```
. . .
DialogBox Type(Metric) Title('Enter a Metric value for the size of something')
   Init(3) Units(Inch) NewVar(metricvar) Button(buttonvar);
if buttonvar = OKButton
   MsgBox 'The user pressed the OK button and entered this data-'+metricvar;
EndIf
. . .
```

DialogBox--ScrollBox

Format 3:

```
DialogBox Type(ScrollBox) [Title(titlestring)]
    [Caption(titlebarText)]
    [Init(initialselection)] List(listvar)
    [NewVar(varname)] [Button(buttonvarname)]
    [ReturnIndex(idxvarname);
```

Table 18: DialogBox Scrollbox Options

Option Name	Option Description
`Type` *(Required)*	This option identifies the dialog as a scrollbox. The value is `ScrollBox`.
`Caption`	A string expression containing the caption of the dialog box in the title bar.
`Title`	A string expression containing the title of the dialog box.
`Init`	The initial number of the selected member. If this value is -1, then no item is selected.
`List`	This is a stringlist variable containing the list of strings to display in the scrolling dialog box.

Table 18: DialogBox Scrollbox Options

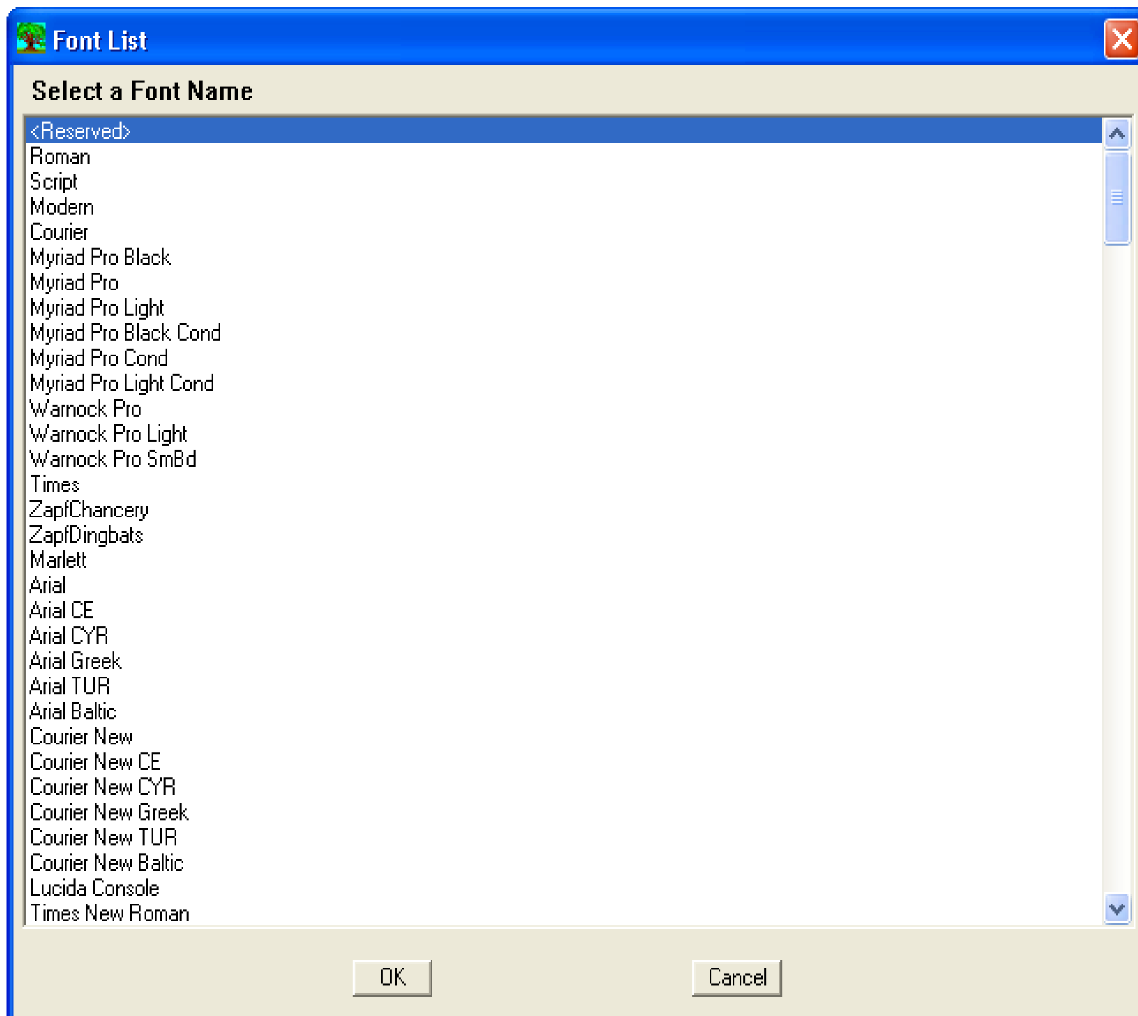
Option Name	Option Description
`NewVar`	The variable name to put the string selected.
`Button`	The variable name to put the value of the button that the user pressed. The value returned in the variable will be one of the following: **`OkButton`** `User pressed the Ok button.` **`CancelButton`** `User pressed the Cancel button.`
`ReturnIndex`	The variable name to put the index of the string selected (starts at 1).

Example:

This example displays the list of fonts (from the Session property FontFamilyNames) and allows the user to select one of them, see “ScrollBox DialogBox” on page 43.

```
DialogBox Type(ScrollBox) Title('Select a Font Name')
   Caption('Font List')
   Init(-1) List(FontFamilyNames)
   NewVar(fontstrvar) Button(buttonvar);
if buttonvar = OKButton
   MsgBox 'The user pressed the OK button and selected this font name-'+fontstrvar;
EndIf
. . .
```

Figure 5-1 *ScrollBox DialogBox*



DialogBox--Multi-Edit

Format:

```
DialogBox Type(MEdit) [Title(DialogTitleString)] [Title2(TitleString)]
    [Label1(Label1String)] [Label2(Label2String)] [Label3(Label3String)]
    [Edit1(Edit1Var)] [Edit2(Edit2Var)] [Edit3(Edit3Var)]
    [CheckBox1Label(Cbx1LabelString) CheckBox1(Cbx1Var)]
    [CheckBox2Label(Cbx2LabelString) CheckBox2(Cbx2Var)]
    [CheckBox3Label(Cbx3LabelString) CheckBox3(Cbx3Var)]
    [CheckBox4Label(Cbx4LabelString) CheckBox4(Cbx4Var)]
    [Button3Label(string)] [Button4Label(string)]
    [Button(buttonvarname)];
```

This format allows you to display a dialog box with up to three edit fields, up to four checkboxes and two optional command buttons. NOTE: Most of these items are optional. They will appear on the dialog box only if you specify them.

This was designed for a quick way to have the user enter a few bits of information. Use the custom forms facility to create dialog boxes that more precisely suit your needs.

Table 19: DialogBox MEdit Options

Option Name	Option Description
`Type` *(Required)*	This option identifies the dialog as a scrollbox. The value is `ScrollBox`.
`Title`	A string expression containing the title of the dialog box.
`Title2`	A string expression containing a title line inside the dialog box.
`Label1`	Specifies the label for the first edit field.
`Edit1`	Specifies the variable in which to place the data in the first edit field.
`Label2`	Specifies the label for the second edit field.
`Edit2`	Specifies the variable in which to place the data in the second edit field.
`Label3`	Specifies the label for the third edit field.
`Edit3`	Specifies the variable in which to place the data in the third edit field.
`CheckBox1Label`	Specifies the label for the first checkbox.
`CheckBox2Label`	Specifies the label for the second checkbox.
`CheckBox3Label`	Specifies the label for the third checkbox.
`CheckBox4Label`	Specifies the label for the fourth checkbox.
`CheckBox1`	Specifies the variable in which to place the state of first checkbox. A value of 0 means the box is unchecked and a value of 1 means the box was checked.
`CheckBox2`	Specifies the variable in which to place the state of second checkbox. A value of 0 means the box is unchecked and a value of 1 means the box was checked.
`CheckBox3`	Specifies the variable in which to place the state of third checkbox. A value of 0 means the box is unchecked and a value of 1 means the box was checked.
`CheckBox4`	Specifies the variable in which to place the state of fourth checkbox. A value of 0 means the box is unchecked and a value of 1 means the box was checked.
`Button3Label`	Specifies the label for the first optional button (the third button altogether).
`Button4Label`	Specifies the label for the second optional button (the fourth button altogether).
`Button`	The variable name to put the value of the button that the user pressed. The value returned in the variable will be one of the following: **`OkButton`** `User pressed the Ok button.` **`CancelButton`** `User pressed the Cancel button.` **`BUTTON3`** `User pressed the first optional button.` **`BUTTON4`** **`User pressed the second optional button.`**

Example:

This example display a set of data entry fields for the user to fill in.

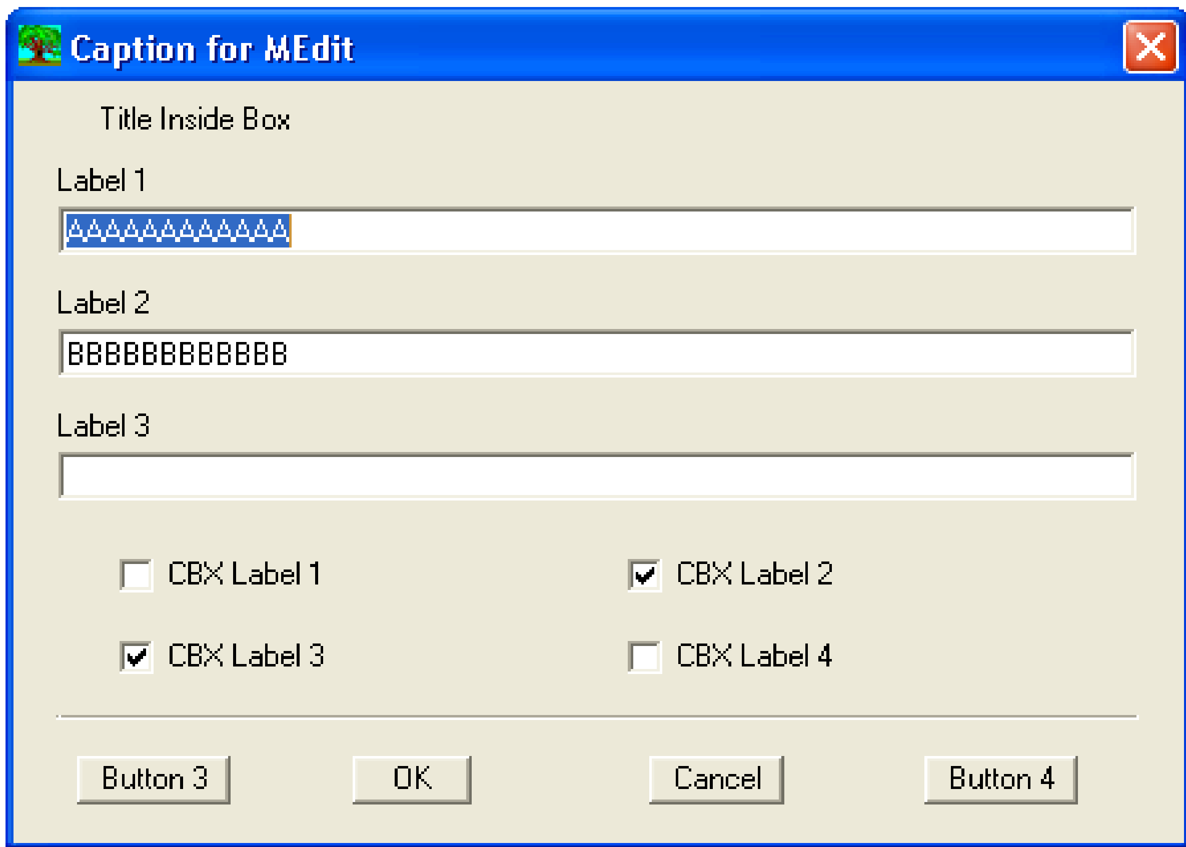
```
DialogBox Type(MEdit) Title('Demographic Information')
    Title2('Enter Name and Address')
    Label1('Name') Edit1(namevar)
    Label2('Address') Edit2(Addressvar)
    Label3('City,State,Zip') Edit3(CityStateVar)
    Button3Label('Use Default') Button(buttonvar);
If buttonvar = OKButton
    MsgBox 'The user pressed the OK button';
Else
    If buttonvar = Button3
        MsgBox 'User pressed the Use Default button';
    EndIf
EndIf
. . .
```

Example:

This example displays all three edit boxes along with all four check boxes, see “Multi-Edit DialogBox” on page 47.

```
SET editVar1 = 'AAAAAAAAAAAA';
SET editVar2 = 'BBBBBBBBBBBB';
SET editVar3 = '';
SET cbxVar1 = False;
SET cbxVar2 = True;
SET cbxVar3 = True;
SET cbxVar4 = False;
DIALOGBOX Type(MEdit)
          Title('Caption for MEdit')
          Title2('Title Inside Box')
          Label1('Label 1')
          Label2('Label 2')
          Label3('Label 3')
          Edit1(editVar1)
          Edit2(editVar2)
          Edit3(editVar3)
          CheckBox1Label('CBX Label 1')
          CheckBox2Label('CBX Label 2')
          CheckBox3Label('CBX Label 3')
          CheckBox4Label('CBX Label 4')
          CheckBox1(cbxVar1)
          CheckBox2(cbxVar2)
          CheckBox3(cbxVar3)
          CheckBox4(cbxVar4)
          Button3Label('Button 3')
          Button4Label('Button 4')
          Button(btnVar)
          ;
IF btnVar = OKBUTTON
   write console 'OK Button Pressed';
ELSE
   IF btnVar = CANCELBUTTON
      write console 'Cancel Button Pressed';
   ELSE
      IF btnVar = BUTTON3
         write console 'BUTTON3 Button Pressed';
      ELSE
         IF btnVar = BUTTON4
            write console 'BUTTON4 Button Pressed';
         ELSE
            write console 'Unknown Button Pressed';
         ENDIF
      ENDIF
   ENDIF
ENDIF
. . .
```

Figure 5-2 *Multi-Edit DialogBox*



MsgBox

The MsgBox command displays a message in a dialog box. There are six different modes for this dialog box, which puts up a set of buttons for the user to push.

Format:

```
MsgBox expression [Mode(modetype)] [Button(buttonpushedvar)];
```

Table 20: MsgBox Options

Option Name	Option Description
`expression` *(Required)*	A computational expression usually a string.

`Mode`	This identifies the type of buttons to put in the box.
	Possible values:
	`OkCancel` `Ok and Cancel buttons, Ok is default.`
	`CancelOk` `Ok and Cancel buttons, Cancel is default.`
	`YesNo` `Yes and No buttons, Yes is default.`
	`NoYes` `Yes and No buttons, No is default.`
	`Warn` `Ok button with a warning icon.`
	`Note` `Ok button with a note icon.`
	`YesNoCancel` `Yes, No and Cancel Buttons,`
	`Yes is default.` *Frame 7.0 or greater*.

`Button`	This option identifies the variable where FrameScript will store the value from which button the user pushed. Possible Values: **OkButton** `User pressed the Ok button.` **CancelButton** `User pressed the Cancel button.` **YesButton** `User pressed the Yes button.` **NoButton** `User pressed the No button.`

Example:

This example displays a message and lets the user select a button.

```
MsgBox 'Press Yes for Yes and No for No' Mode(YesNo)
       Button(btnvar);
If btnvar = YesButton
   MsgBox 'The user pressed the Yes button';
Else
   MsgBox 'The user pressed the No button';
EndIf
```

Display

The display command will display an expression in a message box. You may display any variable or property. If the property is a list property, it will display each member of the list in successive message boxes. The user may cancel the command at each step. This command is very useful for testing and debugging a script.

Format:

```
Display Expression Expression ... Expression
```

Table 21: Display Options

Option Name	Option Description
`Expression`	Any expression.

Example

The following script displays the fonts in the FrameMaker system:

```
. . .
Display FontFamilyNames;
. . .
```

Chapter 6

Subroutines and Functions

Overview

Subroutines are a way of grouping a set of commands together, usually to perform a specific operation. For small scripts this is not usually necessary. As scripts become larger though, they will probably consists of several (perhaps many) steps. Just as in non-scripting tasks, it is important to be able to divide a large problem into a set of smaller (and hopefully easier) sub-problems. In turn, a sub-problem can sometimes be divided into even smaller parts. In programming or scripting languages, these sub-problems and sub-parts are implemented as subroutines.

It is also a good idea to be able test each sub-problem separately. If a task is designed correctly from the beginning, you may be able to reuse some subroutines many times in other scripts.

It is not always easy to determine which group of commands should be written as a subroutine. For example, suppose that you wrote a short script to change all the paragraph formats in the currently active document from ’Heading1’ to ’Heading2’, as follows:

```
Get Object Type(pgfFmt) Name('Heading2') NewVar(gvToFmt);
Loop ForEach(Pgf) In(ActiveDoc) LoopVar(gvPgf)
   If gvPgf.Name = 'Heading1'
      Set gvPgf.Properties = gvToFmt.Properties;
   EndIf
EndLoop
```

This short script gets the paragraph format object named ’Heading2’, loops through each paragraph in the active document and for each paragraph that has its Name property equal to ’Heading1’, it sets all the properties of the paragraph to those of the paragraph format. If you copy these lines into a text file and save it, all you have to do is select this script file with the Run command anytime you want to change all the ’Heading1’ paragraphs to ’Heading2’ paragraphs in the active document. This looks like a candidate for a stand-alone script with no subroutiunes. We will discuss this later on.

> **IMPORTANT:** A real script would do some error checking. One, to make sure there *is* an active document and, two, to make sure that it contains a ’Heading2’ paragraph format. In these examples, we are just focusing on the issue at hand.

Basic Subroutines

You start a subroutine using the keyword **Sub** followed by an identifier representing the name of the subroutine. This name must be unique in the script file where it is defined. It is the name you will refer to when you want to run the commands in the subroutine. This is followed by the commands that you want inside the subroutine. You end a subroutine with the **EndSub** keyword. The following shows the basic syntax of a subroutine definition. For a complete description of the syntax for subroutines, see "Sub Command" on page 55.

```
Sub SubName
    command1;
    command2;
    ...
    commandN;
EndSub
```

To run the commands in a subroutine, you use the **Run** command. The simplest form of the Run command is the keyword **Run** followed by the name of the subroutine you wish to run. Look at the following example.

```
Run sbChangePgfFormat;

Sub sbChangePgfFormat;

    Get Object Type(pgfFmt) Name('Heading2') NewVar(gvToFmt);
    Loop ForEach(Pgf) In(ActiveDoc) LoopVar(gvPgf)
        If gvPgf.Name = 'Heading1'
            Set gvPgf.Properties = gvToFmt.Properties;
        EndIf
    EndLoop

EndSub
```

This script performs the same function as in the first example ("Overview" on page 51), except this time the work is done in a subroutine. The main script consists of just the Run command. Is there some advantage in doing it this way? The answer is Yes! Once you have the commands in a subroutine, you can cause them to run by just using the Run `sbChangePgfFormat` command. In many ways, it is just like creating a new command consisting of a series of other commands. If you have a large script, you may want to perform this operation from different places within that script. Without subroutines, you would have to copy and paste the commands everywhere you wanted to use them and whenever you made a change, you would have to do it in all those places.

Local Variables

Possible Problem

FrameScript creates variables as you need them. These variables will be global variables. They can be accessed (and their values changed) from anywhere in the script (inside subroutines or outside them). In the example above, the subroutine creates two variables, gvToFmt and gvPgf. The Get Object command puts a value in gvToFmt that is an Object value for the new paragraph format. The gvPgf variable is created by the Loop command to store the paragraph object for each iteration of the loop. You want to be able to run this subroutine whenever you want all the paragraph formats changed from 'Heading1' to 'Heading2'. What if you already had a variable named gvToFmt or gvPgf? In this case, whenever you call the subroutine, it will change the values perhaps unexpectedly. This is the cause of some of the most difficult scripting problems to debug (in any language), unexpected interaction from different parts of the

script. It is always a good practice to mimimize the dependence of one part of the script (subroutine) with the other parts of the script. The problem here is that we are using global variables. We need to use variables whose scope is limited to the subroutine itself, so changing its values will not affect other parts of the script.

Local Data Space

When a script starts, a data space is created for all global variables. This data space exists as long as the script is running. Whenever you set a variable's value (with the Set command or otherwise) and the variable does not exist, FrameScript will create the new variableand put it in this global data space. In addition, whenever a subroutine is run (with the Run command), a local data space is created. The variables in this data space are only accessible by commands within that subroutine. When FrameScript looks for a variable, it looks first in the local data space of the current subroutine before looking in the global data space. If it finds it there, it does not look further. We create local variables with the **Local** command.

Local Command

The Local command starts with the keyword Local and is followed by a list of one or more variable names with optional initialization ("Local Command" on page 56 for a complete syntax description). If you do not include an inital value, a NULL value is assumed. Now we will update our subroutine using local variables for names only used in the subroutine.

```
Run sbChangePgfFormat;

Sub sbChangePgfFormat;
    Local lvToFmt lvPgf;

    Get Object Type(pgfFmt) Name('Heading2') NewVar(lvToFmt);
    Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgf)
        If lvPgf.Name = 'Heading1'
            Set lvPgf.Properties = lvToFmt.Properties;
        EndIf
    EndLoop

EndSub
```

Since we have used the Local command to create the local variables, lvPgf and lvToFmt, we do not have to worry about unintentionally modifying a variable created in some other part of the script. Notice also that we have changed the name of the variable to use an lv prefix instead of the gv in the original example. This is just a convention that we chose to use. We use gv as a prefix for global variables and lv as a prefix for local variables. We also use sb as a prefix for subroutine names.

Passing Arguments to Subroutines

The above subroutine could be a useful bit of code if you need to change all Heading1 paragraphs to Heading2 paragraphs. But what if you wanted to change Heading1 paragraphs to Heading3 or Body to Normal or whatever to whatever else?. Do do we need to need to write a separate subroutine for each combination paragraph formats? No, we simply need a way to give the subroutine some more information when we run it. This involves passing values to the subroutine. These values are known as arguments or parameters and these are stored as variables in a special parameter data space. Like the local data space, there is one of these for each subroutine and it is created when the subroutine is run.

In FrameScript, there is a very loose calling convention. Parameters are passed on the **Run** command. The keyword Run is followed by the subroutine name which is followed by zero or more parameters in the form:

```
Run subname ParmName1(expr1) ParmName2(expr2) ..., ParmNameN(exprN);
```

For each Parameter name on the Run command, a variable is created in the parameter data space for that subroutine using the parameter name. The expression is evaluated and the corresponding variable is given that value. In the following example, the Run command creates a parameter data space with two variables, `pvFromFmtName` and `pvToFmtName` and their values will be 'Heading1' and 'Heading2', respectively. The Sub command also mentions the variable names, but this is just for documentation. When the subroutine runs and it looks for a variable name, it looks first at the local data space, then at the parameter data space before looking for global variables.

```
Run sbChangePgfFormat pvFromFmtName('Heading1') pvToFmtName('Heading2');

Sub sbChangePgfFormat using pvFromFmtName pvToFmtName;
   Local lvToFmt lvPgf;

   Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
   Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgf)
      If lvPgf.Name = pvFromFmtName
         Set lvPgf.Properties = lvToFmt.Properties;
      EndIf
   EndLoop

EndSub
```

Now we have a subroutine that we can run anytime we want to change all paragraphs with one paragraph to another of any type. All we have to do is supply the from paragraph tag and the to paragraph tag.

Returning values from Subroutines

One more thing you might want to do with subroutines is to return a value (or more than one value). The subroutine might compute a value that you want to use back in the main part of the script. For example, suppose you wanted to know how many paragraphs were changed in our example script. You can return values from a subroutine using a special type of parameter identified by the **returns** keyword. When you put the **returns** keyword before a

parameter on the **Run** command, it makes the parameter updatable. You must use a variable name as the parameter value. Here is our sample subroutine updated to return the number of paragraphs changed.

```
Run sbChangePgfFormat pvFromFmtName('Heading1') pvToFmtName('Heading2')
      returns pvPgfChgCount(gvCount);
Display 'The number of paragraphs changed was '+gvCount;

Sub sbChangePgfFormat using pvFromFmtName pvToFmtName pvPgfChgCount;
   Local lvToFmt lvPgf lvCount(0);

   Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
   Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgf)
      If lvPgf.Name = pvFromFmtName
         Set lvPgf.Properties = lvToFmt.Properties;
         Set lvCount = lvCount + 1;
      EndIf
   EndLoop
   Set pvPgfChgCount = lvCount;

EndSub
```

After this subroutine runs, the gvCount variable should contain the number of paragraphs changed. We used a new local variable (lvCount) to hold the running count. We could have just used the pvPgfChgCount variable and it would have worked just as well, but it was a good time to illustrate setting an initial value for a local variable.

Sub Command

The following shows the format of the Sub command.

Format:

```
Sub subname [using arg1[ arg2]...[ argN]];
. . .
EndSub
```

In between the Sub and EndSub lines there can be any number of commands.

Table 22: Sub Options

Option Name	Option Description
`subname` *(Required)*	The name of the subroutine. This name has to be unique within the script. You cannot have two subroutines with the same name in the same script file.
`using`	An optional filler word to enhance readability.
`argI`	The name of an argument. These are names (identifiers) of your choosing. Each name represents the name of a variable in the parameter data space for this subroutine. These are used as variable names inside the subroutine. These names are for documentation purposes and are treated as comments by FrameScript. The Run command supplies the actual names that are inserted into the parameter data space.
`EndSub` *(Required)*	The command that terminates a subroutine.

Run Command

The **Run** command allows you to run FrameScript subroutines. A subroutine is a defined set of command (see “Sub Command” on page 55). The subroutine runs the subroutine commands until it gets to the end of the subroutine or a LeaveSub command is executed. You may pass data names to a subroutine, which the subroutine may use during its run. You may also pass data names to the subroutine, which the subroutine may modify. **NOTE**: Though you may pass properties names and constant values *to* a subroutine, you may only return variable names *from* a subroutine.

Format:

```
Run subExpression [Parm1[(value1)] ... ParmN[(valueN)] [returns retparm(varname)];
```

Table 23: Run Options

Option Name	Option Description
subExpression	This is an expression that evaluates to a subroutine (SubVar data type). Many times this is just the name of the subroutine (from the Sub command). If the expression is a string value, the value in the string value will be used as the subroutine name.
Parm	These options allow you to pass information to the subroutine. This can be any name that is not otherwise in use. You don’t have to use the word Parm. If the value is not present, then the name of the parm is passed to the subroutine as an integer variable with the value of True (1).
returns	This option specifies that the following parm name is modifiable in the subroutine. It must be a variable name and not a value.

Local Command

The Local command declares the named variable to be local to the current subroutine. This is useful for writing subroutines and keeping the information in the subroutine separate from the rest of a script. Ordinarily all variables created in a script are global (accessible) to the entire script. Therefore, in a subroutine, you would have to worry about possible naming conflicts with other subroutines or with the main part of the script. This command lets you create variables that are only accessible from within the subroutine.

Format:

```
Local var1[(expresssion)] [var2[(expression)]];
```

Examples:

This example creates two varaiables var1 and var2 in the main script and assigns them integer values. It then calls a subroutine which has a local variable called var2. This var2 is different than the var2 from the main part of the script. var1, however, is a global variable and a new version was not declared in the subroutine. So the display from the following script should be;

- Var1=10 Var2=99
- In Sub1--Var1=30 Var2=88
- Var1=30 Var2=99

```
SET var1 = 10;
set var2 = 99;
Display 'Var1='+var1+'  Var2='+var2;
Run Sub1;
Display 'Var1='+var1+'  Var2='+var2;

Sub Sub1
   Local var2(0);
   Set var1 = 30;
   Set var2 = 88;
   Display 'In Sub1--Var1='+var1+'  Var2='+var2;
EndSub
```

LeaveSub

The **LeaveSub** command jumps out of the current subroutine and continues running as if the subroutine ended normally.

Format:

```
LeaveSub;
```

Example:

This example searches the list of paragraph formats in the currently active document for the first one which uses autonumbering.

```
Sub LookUpPgfFormatAutonum returns ReturnPgf

   Loop ForEach(PgfFmt) In(ActiveDoc) LoopVar(pformat)
      If pformat.PgfIsAutoNum
         Set ReturnPgf = pformat;
         LeaveSub;
      EndIf
   EndLoop

EndSub
```

Example:

The following script illustrates a useful FrameScript subroutine. This subroutine checks to see if a document is already open in the current FrameMaker session. The double slash indicates a comment entry. Any text after the double slash will not be processed and is only there to document the script for the scriptwriter.

```
. . .
//-----------------------------------------------------------------------------
//   This subroutine checks the list of open documents to see if the
//   specified document is already open.
//
//   Format:
//       Run IsDocAlreadyOpen Filename(testfilename) returns DocObj(retdocvar)
//
//    If it is open, it returns the document object if it is already open
//    and returns zero if it is not open.
//-----------------------------------------------------------------------------
Sub isDocOpenAlready using filename DocObj

      // Upper case the string
   Get String FromString (filename) Uppercase ReturnString(tfilename);
   Set DocObj = 0;
   Loop foreach(Doc) In(Session) LoopVar(testdocobj)
      // Upper case the string
      Get String FromString (testdocobj.Name) Uppercase ReturnString(tname);
      If tname = tfilename
         Set DocObj = testdocobj;
         LeaveLoop;
      EndIf
   EndLoop

EndSub
//---------------------------------------------------------------------------------
```

User Functions

User functions are similar to subroutines in that they consist of groups of commands identified by a name (identifier). There are some major differences however.

- A user function is designed to work as part of an expression and not started by the Run command.
- Functions return a value.
- Arguments are surrounded by **Braces** ({}) and are separated by commas.
- If no arguments are required, then you must use a pair of empty braces.
- The order of the arguments is important.
- The parameter names on the function command are no longer just a comment. They show the expected order of the parameters when the function is called.
- Updatable parameters are labeled with the keyword ByRef

Here is our sample subroutine modified to work as a function.

```
Set gvCount = fnChangePgfFormat{'Heading1','Heading2'};
Display 'The number of paragraphs changed was '+gvCount;

Function fnChangePgfFormat using pvFromFmtName pvToFmtName;
    Local lvToFmt lvPgf lvCount(0);

    Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
    Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgf)
        If lvPgf.Name = pvFromFmtName
            Set lvPgf.Properties = lvToFmt.Properties;
            Set lvCount = lvCount + 1;
        EndIf
    EndLoop
    Set Result = lvCount;

EndFunction
```

Notice that the function body looks almost identical to the subroutine version of the same thing. In fact, the only difference in the body of the function is the command after the **EndLoop**. **Result** is a reserved name. Every function has a **Result** variable defined. Its initial value is **NULL**. Whatever value is in the **Result** variable when the function ends is returned to the expression where the function was called. If you do not assign the Result variable a value, when the function ends the original **NULL** value will be returned.

With subroutine calls, each parameter is named on the **Run** command. Since functions are run from inside expressions, there is no way to name any of the values when the function is run. The names of parameters come from the names listed on the function declaration command. The order of the parameter names must match the order in which they appear between the braces. In the subroutine version, we could have put the pvFromFmtName and pvToFmtName options is any order we wished on the Run command. With functions the order matters, as follows:

```
Run sbChangePgfFormat pvFromFmtName('Heading1') pvToFmtName('Heading2')
        returns pvPgfChgCount(gvCount);
```

is the same as

```
Run sbChangePgfFormat pvToFmtName('Heading2') pvFromFmtName('Heading1')
        returns pvPgfChgCount(gvCount);
```

On the other hand:

```
Set gvCount = fnChangePgfFormat{'Heading1','Heading2'};
```

is very different than

```
Set gvCount = fnChangePgfFormat{'Heading2','Heading1'};
```

IMPORTANT: This is just a reminder that the parameters are enclosed inside braces ({}) and not parentheses. Most languages use parentheses, but the syntax of FrameScript, since it uses parentheses for option names, requires braces. In fact, I'll mention it once again. Use BRACES for function arguments.

Functions can have a variable number of arguments. If you supply fewer arguments when the function is called than on the declaration, the remaining parameters are given NULL values. If you have more arguments when the function is called than on the declaration then each additional argument is given a name with the following pattern:

```
FuncArgN
```

where N is the number of the argument. Also, there is an argument pseudo-array, called **`Args`** which represents each of the arguments passed. **`Args.Count`** gives the number of arguments. **`Args[1]`** is the first argument, etc.

Functions can only return one value, but, like subroutines, you can modify the parameters if you use a variable instead of a value. Also, you have to tell the function to expect a variable to using the **`ByRef`** keyword before the name in the parameter list on the function command. For example, if we wanted to modify our sample script to return to total number of paragraphs in the document as well as the count of the number of paragraph changed. We could do the following:

```
Set gvTotalPgfs = 0;
Set gvCount = fnChangePgfFormat{'Heading1','Heading2',gvTotalPgfs};
Display 'The number of paragraphs changed was '+gvCount+
       ' Total Pgfs-'+gvTotalPgfs;

Function fnChangePgfFormat using pvFromFmtName pvToFmtName ByRef pvTotal;
   Local lvToFmt lvPgf lvCount(0) lvTotalPgfs(0);

   Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
   Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgf)
      Set lvTotalPgfs = lvTotalPgfs + 1;
      If lvPgf.Name = pvFromFmtName
         Set lvPgf.Properties = lvToFmt.Properties;
         Set lvCount = lvCount + 1;
      EndIf
   EndLoop
   Set pvTotal = lvTotalPgfs;
   Set Result = lvCount;

EndFunction
```

Be sure to use a variable name and not a value for any ByRef argument.

Function Declaration Command

The following shows the format of the Function declaration command.

Format:

```
Function functionName [using [ByRef ]arg1[ [ByRef ]arg2]...[ [ByRef ]argN]];
. . .
EndFunction
```

In between the Function and EndFunction lines there can be any number of commands.

Table 24: Function Options

Option Name	Option Description
`functionName` *(Required)*	The name of the subroutine. This name has to be unique within the script. You cannot have two subroutines with the same name in the same script file.
`using`	An optional filler word to enhance readability.

Table 24: Function Options

Option Name	Option Description
argI	The name of an argument. These are names (identifiers) of your choosing. Each name represents the name of a variable in the parameter data space for this function. These are used as variable names inside the subroutine. These ofder of these names determine the order of the arguments required when the function is called in some expression. The Args array is an alternate way to access the arguments.
EndFunction *(Required)*	The command that terminates a function.

Calling a Function

The following shows the format of calling a Function.

Format:

```
Set value = [(]functionExpression()){[arg1[,arg2[...][,argN]};
```

Table 25: Calling Function Options

Option Name	Option Description
functionExpression *(Required)*	An expression resulting in a function (SubVar) value. You may have to enclose this in parentheses
argI	An expression or a variable name (if ByRef is used in the function declaration). The arguments are enclosed in Braces. If a function has no arguments, then a set of empty braces is required.

Sub/Function Expressions

In the above discussion, we have been running subroutines and calling functions using the name of the subroutine or function. In fact, we are using subroutine and function expressions. When a suborutine or function is declared, a read-only variable is created to represent that subroutine or function, using the function name as the variable name. This subroutine or function variable name has a data type of **SubVar**. Running a subroutine or calling a function means computing a SubVar data type. So any expression that evaluates to a SubVar data value can be used to identify a subroutine or function. Since the name of the subroutine or function is a variable with a SubVar value, using the subroutine or function name is the easiest way to access the subroutine or function. Also, as a special case, if the expression evaluates to a **String** data type and the string value is the name of a subroutine or function, then it re-evaluates it into a SubVar using that subroutine or function name.

Most of the time you will just use the subroutine or function name, but, occasionally, you may want to use an expression. This allows you to determine which subroutine or function to call at run time.

Here is an example of using a string variable to run a subroutine.

```
Set gvString = 'MyTestSub';
Run gvString pvParm1(100) pvParm2('QQQQ');
...
Sub MyTestSub using pvParm1 pvParm2;
...
EndSub
```

You can do the same thing with a string expression, as follows:

```
Run 'MyTest'+'Sub' pvParm1(100) pvParm2('QQQQ');
...
Sub MyTestSub using pvParm1 pvParm2;
...
EndSub
```

Sub and Function expressions will be more important in the next chapter about Modules.

Chapter 7

Modules

Introduction

Many (if not most) scripts consist of one text file (or object file, .fso), not counting any text files that are 'Included' in another file. It is possible for one logical script to consist of several physical script text files. Each of these script files are refered to as **`Modules`**. There is always one Main script. This is the one that you **`Run`** (for standard scripts) or **`Install`** (for Event scripts). The **`MainScript`** session variable (read-only) gives the name of this script. This main script, however, can run subroutines and functions in other physical script files. Most of the time these other script files are used to store a set of utility functions or subroutines. One can develop a library of utilities that are used in many other scripts. FrameScript comes with a few sets of utilities, located in the Lib folder under the FrameScript folder. Also, if you have a very large and complicated script, it may be more practical to break it down into several source files. This is especially true if only some parts of the script are used in any one run.

The Main script is loaded into memory when the script is Run or Installed. The other script files are loaded as needed, whenever you try to Run a subroutine or function located in that script file. It stays around until the main script is finished.

$Main

Even though we've been discussing subroutines versus the main script, all FrameScript commands are inside some subroutine. Anytime you have commands outside of a subroutine, FrameScript automatically creates a subroutine called $Main. When FrameScript runs a script it actually runs the $Main subroutine. Most of the time you don't have to know about this. There are times, however, in dealing with Modules, that this concept will be important.

Using Modules

As described in the last chapter, to run a subroutine or to call a function, an expression is evaluated that results in a SubVar value. A SubVar data value contains all the information necessary to locate a subroutine or function. To run a subroutine or call a function in another script file, you must do the same thing. A SubVar data value not only has the name of the subroutine or function, it also has the name of the script file where the subroutine or function is located. SubVar data values also have information about calling subroutines and functions inside EslObjects, but that is described in the EslObject Reference. There are two other data types that are useful in subroutine and function expressions. There are LibVar and ScriptVar. A LibVar represents a folder on a disk and a ScriptVar represents a script file. Both of these can be involved in expressions that result in SubVar values. A string variable can also represent a script file.

Using a SubVar

The simplest way to access a subroutine in another script file is to create a SubVar variable using the New SubVar command (See “New SubVar” on page 69 for a complete description of the New SubVar command). Of course, you can use a SubVar variable to access a subroutine or function in the same script as well. Here are two examples:

```
New SubVar NewVar(gvMySub) SubName('Sub1');
Run gvMySub;
. . .

Sub Sub1
    . . .;
EndSub
```

and

```
New SubVar NewVar(gvMySub) SubName('Sub1') File('C:\TestScripts\Test.fsl');
Run gvMySub;
. . .
```

The first example creates a SubVar that identifies a subroutine in the same script. The second identifies a subroutine in the script file called c:\TestScripts\Test.fsl.

Using a SubVar may be the simplest way to access a subroutine or function in another script file, it may not be the most convenient or easiest to use. Suppose you have a script file that has many subroutines and functions. Using SubVars to access them would mean creating a separate subvar variable for each one. This might be tedious for a file with a large number of subroutines and functions.

Using a ScriptVar

An alternative is to use a ScriptVar instead. A ScriptVar represents an entire script file (See “New ScriptVar” on page 68 for a complete description of the New ScriptVar command). You can access individual subroutines and functions using the property operator(.). The following creates a ScriptVar, then calls a subroutine (called Sub1) inside that script file.

```
New ScriptVar NewVar(gvMyScript) File('c:\TestScripts\Util.fsl');
Run gvMyScript.Sub1;
. . .
```

Using the property operator on a ScriptVar (except for the standard properties) causes it to evaluate to a SubVar value with the script file taken from the ScriptVar variable and the SubName comes from the property (in this case, Sub1). Using a ScriptVar, you can create one variable, yet be able to access each subroutine or function inside that script file. You can also Run the script file itself:

```
New ScriptVar NewVar(gvMyScript) File('c:\TestScripts\Util.fsl');
Run gvMyScript;
. . .
```

In this case, it evaluates to a SubVar value with the script file taken from the ScriptVar and the SubName will be $Main, running the main script (if any) inside the script file.

There is another way to build a script file other than reading commands from a script file. You can build a string value that contains the commands (with or without subroutines or functions) and use the ScriptText option of the New ScriptVar command.

```
Set gvString = ' Set gvCount = 0; ';
Set gvString = gvString + ' Loop ForEach(Pgf) In(ActiveDoc) LoopVar(gvPgf); ';
Set gvString = gvString + ' Set gvCount = gvCount + 1; '
Set gvString = gvString + ' EndLoop; '

New ScriptVar NewVar(gvMyScript) ScriptText(gvString);
Run gvMyScript;
. . .
```

This small script counts the total number of paragraphs in the currently active document.

Using a LibVar

Another way is to use the LibVar variable (See “New LibVar” on page 67 for a complete description of the New ScriptVar command). A LibVar represents an entire folder. A property of a LibVar data value (except for the standard properties) evaluates to a ScriptVar. For example:

```
New LibVar NewVar(gvMyLib) Path('c:\TestScripts');
Run gvMyLib.extFile;
```

The gvMyLib.extFile evaluates to a ScriptVar where the script name it composed of a combination of the folder (c:\TestScripts), the file name (extFile), and the extensions in the Run options (fsl or fso). Since there is no SubName, it uses $Main to run the main script in that file (C:\TestScripts\extFile.fsl).

Since the LibVar property expression evaluates to a ScriptVar, you can also continue the expression by adding a subroutine of function name to completely identify a subroutine or function starting with the LibVar, as follows:

```
New LibVar NewVar(gvMyLib) Path('c:\TestScripts');
Run gvMyLib.extFile.Sub1;
```

This evaluates to a SubVar with subname Sub1 located in the idenified script file.

Using a String

In the last chapter we illustrated how to run a subroutine in the same script file by putting the name of the subroutine in a string value. You can also run a subroutine in another script file in a similar manner. You put the script file name into a string variable and use the name of the subroutine as the property. For example:

```
Set gvStrFileName = 'c:\TestScripts\Util.fsl';
Set gvSubName = 'Sub1';
Run gvStrFileName.gvSubName;
```

Summary

The prefered way to access subroutines and functions in other scripts is to use the SubVar data type for single subroutines or functions, but use the ScriptVar method when you have many subroutines or functions in a script file, such as a set of utility subroutines or functions. Using string variables is discouraged. It is an older method and, since string values have many possible properties, there is much more danger of a naming conflict. The LibVar method has problems because it expects a file name and the naming conventions of system file names are different than FrameScript identifiers. Some script files cannot be identified this way.

Some Examples

Example 1:

This sample script runs the same subroutine if it were located in another script file (`c:\TestScripts\util.fsl`).

```
. . .
Set retvar = 99;
set LibScript = 'c:\FrameScript\util.fsl';
set subroutinestr = 'MySubroutine';
Run Libscript.subroutinestr intval(60) sval('MyString') returns GetIt(retvar);
Display 'This subroutine modified the retvar variable to be '+retvar;
. . .
```

Example 2:

This sample script runs a script called MyExternalSub.fsl located in the directory (`c:\FrameScript\lib`).

```
. . .
Set retvar = 99;
NEW LibVar NewVar(myLib) Path('c:\FrameScript\lib');
Run myLib.MyExternalSub intval(60) sval('MyString') returns GetIt(retvar);
Display 'This subroutine modified the retvar variable to be '+retvar;
. . .

--->IN THE FILE c:\FrameScript\lib\MyExternalSub.fsl

    IF sval = 'MyString'
        Set Getit = intval * 2;
    Else
        Set Getit = intval * 10;
    EndIf
```

Example 3:

This sample script runs a subroutine called MyExternalSub.fsl located in the script file (`c:\FrameScript\Util.fsl`).

```
. . .
Set retvar = 99;
NEW ScriptVar NewVar(myScript) File('c:\FrameScript\Util.fsl');
Run myScript.MyExternalSub intval(60) sval('MyString') returns GetIt(retvar);
Display 'This subroutine modified the retvar variable to be '+retvar;
. . .

--->IN THE FILE c:\FrameScript\lib\Util.fsl
SUB MyExternalSub using intval sval returns Getit
    IF sval = 'MyString'
        Set Getit = intval * 2;
    Else
        Set Getit = intval * 10;
    EndIf
ENDSUB
```

Example 4:

This example creates a SubVar variable and uses it pass the name of the subroutine to the subroutine called TestSub, which runs it.

```
. . .
New SubVar NewVar(mySub) subname('Sub1');
RUN TestSub CallBack(mySub) returns RVal(myRetVal);
DISPLAY myRetVal;
. . .
. . .

SUB Sub1 using Parm1, Parm2, Val
    Set Val = Parm1 + 999;
    DISPLAY parm2;
ENDSUB

. . .
SUB TestSub using CallBack, RVal
    LOCAL     bbb;
    RUN CallBack parm1(444) Parm2('qqq') returns Val(bbb);
    SET RVal = bbb;
ENDSUB
```

New LibVar

Format:

```
New LibVar
    Path(DirectoryName) NewVar(varname);
```

This creates a LibVar variable. This type of variable makes it easier to run scripts outside the current script. A directory on your hard disk (Folder on the Macintosh), can be used as a library of scripts. The file extension is added by checking the list of file extensions available.

Table 26: New LibVar Options

Option Name	Option Description
Path *(Required)*	The name of the directory or folder on your hard disk that will be used for the library.
NewVar *(Required)*	The name of the variable to hold the newly created LibVar.

Example 1:

This example creates a LibVar variable and uses it to run a script called GetInfo in the c:\FrameScript\Lib directory. The File extension (e.g fsl) is added from the list of available file extensions (See Options in the users guide)

```
. . .
New LibVar NewVar(myLib) Path('C:\FrameScript\Lib');
RUN myLib.GetInfo Parm1(123) Parm2('Val2') returns Val(xxx);
. . .
```

New ScriptVar

Format:

```
New ScriptVar NewVar(varname)
    File(FileName)
or
    ScriptText(StringValue)
;
```

This creates a ScriptVar variable. This type of variable makes it easier to run scripts outside the current script. You can use this variable type to run an entire file as a script or to run a subroutine inside another script.

Table 27: New ScriptVar Options

Option Name	Option Description
File	The name of the file on your hard disk that will be used for the script.
ScriptText	A string value consisting of script commands. These commands will be compiled into a script. You must use either the ScriptText option or the File option.
NewVar *(Required)*	The name of the variable to hold the newly created ScriptVar.

Example 1:

This example creates a ScriptVar variable and uses it to run the script `C:\FrameScript\MyScripts\GetIt.fsl`.

```
. . .
New ScriptVar NewVar(myScript) File('C:\FrameScript\MyScripts\GetIt.fsl');
RUN myScript Parm1(123) Parm2('Val2') returns Val(xxx);
. . .
```

Example 2:

This example creates a ScriptVar variable and uses it to run a subroutine called GetInfo inside the `C:\FrameScript\MyScripts\Utils.fsl` file.

```
. . .
New ScriptVar NewVar(myScript) File('C:\FrameScript\MyScripts\Utils.fsl');
RUN myScript.GetInfo Parm1(123) Parm2('Val2') returns Val(xxx);
. . .
```

Example 3:

This example builds a script from a string then runs the script, Since it runs a ScriptVar and not a subroutine, it runs the $Main subroutine.

```
Set AAA = 1;
Set BBB = 3;
New ScriptVar NewVar(MyMemoryScript)
   ScriptText('Set vResult = AAA + BBB');
Run MyMemoryScript;
MsgBox 'New value is '+vResult;
```

New SubVar

Format:

```
New SubVar
   [File(FileName)] Subname(subname) NewVar(varname);
```

This creates a SubVar variable. This type of variable makes it easier to run scripts that our not necessarily known at design time. You can use this variable type to run subroutine that may different depending on run-time considerations. It is particularly useful to pass a subvar to a another subroutine, which may, in turn, run the subvar.

Table 28: New SubVar Options

Option Name	Option Description
`File`	The name of the file on your hard disk that contains the script. If not specifed, the current script will be used.
`Subname` *(Required)*	The name of the subroutine..
`NewVar` *(Required)*	The name of the variable to hold the newly created ScriptVar.

Example 1:

This example creates a SubVar variable and uses it to run the subroutine called Sub1.

```
. . .
New SubVar NewVar(mySub) subname('Sub1');
RUN mySub Parm1(123) Parm2('Val2') returns Val(xxx);
. . .
. . .

SUB Sub1 using Parm1, Parm2, Val
   Set Val = Parm1 + 999;
   DISPLAY parm2;
ENDSUB
```

Example 2:

This example creates a SubVar variable and uses it pass the name of the subroutine to the subroutine called TestSub, which runs it.

```
. . .
New SubVar NewVar(mySub) subname('Sub1');
Run TestSub CallBack(mySub) returns RVal(myRetVal);
Display myRetVal;
. . .
. . .

Sub Sub1 using Parm1 Parm2 Val
   Set Val = Parm1 + 999;
   Display parm2;
EndSub
. . .
Sub TestSub using CallBack, RVal
   Local     bbb;
   Run CallBack parm1(444) Parm2('qqq') returns Val(bbb);
   Set RVal = bbb;
EndSub
```

Chapter 8

Standard Script Library

Introduction

The Lib folder under your FrameScript directory contains a set of scripts that make up the standard library. These scripts should not be changed as they are used by many of the sample scripts. When the product is installed, the Lib folder is part of the search path (see Options). If you wish to have your own script library, it is best that you create a new folder (for example, 'MyLib') and place the scripts there. You can use the Options dialogs to add this folder to the search path (in addition to the Lib folder).

In the standard script library (Lib Folder), there are three script files, DocUtils, DlgDualSelect and DBUtils (in addition to the EDBUtils which is left over from version 2.1).

DocUtils

The DocUtils script contains the following Functions and Subroutines. These are useful for working with documents. In order to use the subroutines and functions in this script, you have to first create a ScriptVar variable, which identifies the DocUtils script, as follows:

```
New ScriptVar NewVar(eDocUtils) File('DocUtils');
```

Since the Lib folder is in the search path, you do not have to specify a full path name to locate it. FrameScript will search the folders in the search path for a script if a complete path is not specified. Also, it will use the file extensions from the file extension list, if no file extension is specified. So, as long as the DocUtils.fsl file is located somwhere in the search path, the above command will create a variable that can access the subroutines and functions in the file.

You only have to create this variable once if you make it a global variable.

Function DocIsAlreadyOpen

This function takes one parameter (a file name) and returns a document object if the file is already open and it returns NULL if it is not open.

Format:

```
Set gvDoc = eDocUtils.DocIsAlreadyOpen{filename};
```

Where **`filename`** is a string containing the name of the file.

Example:

```
New ScriptVar NewVar(eDocUtils) File('DocUtils');

Set gvDoc = eDocUtils.DocIsAlreadyOpen{'C:\MyFiles\MyTestFile.fm'};
If gvDoc
   MsgBox 'File is Already Open';
Else
   MsgBox 'File is Not Open';
EndIf
```

Function ForAllDocsInBook

This function is a utiltity function that makes it easier to process all the documents in a book. It takes care of some of the housekeeping issues and lets the script write concentrate on what to do with each component. This function takes two parameters, a book object and a SubVar variable. For each document in the book, this function will open the document (if not already open) and it will call the subroutine specified by the SubVar parameter with the document object as its parameter. For an example of this, see the BookFindReplace.fsl sample script.

Format:

```
Set gvCount = eDocUtils.ForAllDocsInBook{bookVar,mySubVar};
```

Where **`bookVar`** is a book object variable and **`mySubVar`** is a SubVar variable.

Example:

```
New ScriptVar NewVar(eDocUtils) File('DocUtils');
Set gvBook = ActiveBook;
If gvBook = 0
   LeaveSub;
EndIf
New SubVar NewVar(gvCallBackSub) SubName('ProcessDoc');
Set gvCount = eDocUtils.DocIsAlreadyOpen{gvBook,gvCallBackSub};

...

Sub ProcessDoc using pvDocVar

   Write console 'Processing Doc-'+pvDocVar.Label;

EndSub
```

Function GetCellXY

This function is a utiltity function that returns the Table Cell object of the specified table, row and column number, if it exists. It returns NULL otherwise. This will save you the trouble of navigating through the table rows and columns.

Format:

```
Set gvCellVar = eDocUtils.GetCellXY{tableVar,rowNumber,colNumber};
```

Where **`tableVar`** is a table object variable, **`rowNumber`** is the row number of the cell and **`colNumber`** is the column number of the cell that you want.

Example:

```
New ScriptVar NewVar(eDocUtils) File('DocUtils');
Set gvTable = FirstTableInDoc;
If gvTable = 0
   LeaveSub;
EndIf
Set gvCellVar = eDocUtils.GetCellXY{gvTable,3,5};

...
```

In this example, the value returned from the function should be the cell object for the third row and fifth column of the specified table.

Sub AddParaToCellXY

This function is a utiltity function that allows you to specify the text of the first paragraph of the specified table cell, identified by row and column number. This will save you the trouble of navigating through the table rows and columns.

Format:

```
Run eDocUtils.AddParaToCellXY pvTable(tableVar) pvRowNum(rowNumber)
       pvColNum(colNumber) pvText(text) pvPgfFmt(pgfFmtVar);
```

Where **`tableVar`** is a table object variable, **`rowNumber`** is the row number of the cell and **`colNumber`** is the column number of the cell that you want. **`text`** is the text string to be the first paragraph in the cell and **`pgfFmtVar`** is a paragraph format object that specifies the paragraph format for this paragraph. The **`pvPgfFmt`** parameter is optional.

Example:

```
New ScriptVar NewVar(eDocUtils) File('DocUtils');
Set gvTable = FirstTableInDoc;
If gvTable = 0
   LeaveSub;
EndIf
Get Object Type(PgfFmt) Name('Body') NewVar(gvPgfFmt);
Run eDocUtils.AddParaToCellXY pvTable(gvTable) pvRowNum(3) pvColNum(5)
       pvText('Text for first para') pvPgfFmt(gvPgfFmt);

...
```

Dual Select Dialog

The Dual Select dialog script (DlgDualSelect) contains one function that displays a dialog box allowing the user to select multiple items from one string list data type into another. These are useful for presenting the user a way of selecting a group of items instead of just one (as in the standard ScrollBox). In order to use the function in this script, you have to first create a ScriptVar variable, which identifies the DlgDualSelect script, as follows:

```
New ScriptVar NewVar(eDlgScript) File('DlgDualSelect');
```

Since the Lib folder is in the search path, you do not have to specify a full path name to locate it. FrameScript will search the folders in the search path for a script if a complete path is not specified. Also, it will use the file extensions

from the file extension list, if no file extension is specified. So, as long as the DlgDualSelect.fsl file is located somwhere in the search path, the above command will create a variable that can access the function in the file.

You only have to create this variable once if you make it a global variable.

Function DlgStringDualSelect

This function is presents a dialog box to the user with two list boxes, one is the source list and the other is the selected list. You may specify a title and headings for each list box

Format:

```
Set gvCellVar = eDlgScript.DlgStringDualSelect{srcList, InitToList,
    titleString, Head1String, Head2String};
```

Where **`srcList`** is a string list providing the source list and **`InitToList`** is a string List providing the initial contents of the selected list (this parameter is options), **`titleString`** is the string in the caption of the dialog and **`Head1String`** is the heading for the source list and **`head2String`** is the heading for the selected list.

Example:

```
New ScriptVar NewVar(eDlgScript) File('DlgDualSelect');
Local lvPgfSourceList(DocPgfFmtNameList);
Local lvPgfList;
Set lvPgfList = eDlgScript.DlgStringDualSelect{lvPgfSourceList,,
    'Select Paragraph Formats',
    'Paragraph Formats',
    'Selected Formats'};
...
```

In this example, a list of pargraph format names is passed as the source list and the value returned is a list of paragraph formats that the user selected.

Database Utilities

The database utilities script (DBUtils) contains the following Functions and Subroutines. These are useful for working with databases. In order to use the subroutines and functions in this script, you have to first create a ScriptVar variable, which identifies the DBUtils script, as follows:

```
New ScriptVar NewVar(eDBUtils) File('DBUtils');
```

Since the Lib folder is in the search path, you do not have to specify a full path name to locate it. FrameScript will search the folders in the search path for a script if a complete path is not specified. Also, it will use the file extensions from the file extension list, if no file extension is specified. So, as long as the DocUtils.fsl file is located somwhere in the search path, the above command will create a variable that can access the subroutines and functions in the file.

You only have to create this variable once if you make it a global variable.

Function DlgDB_Connection

This function is presents a dialog box to the user allowing the user to select a data source from a list of available data sources or to specify an MS Access or MS Excel file name. This function will open the database (make a connection to it) and return the database object back to the calling script.

Format:

```
Set gvDatabase = eDBUtils.DlgDB_Connection{mode};
```

Where **mode** is an optional string value. If this value is 'Update', then the database will be opened for updating. Otherwise it is read-only.

Example:

```
New ScriptVar NewVar(eDBUtils) File('DBUtils');
Set gvDatabase = eDBUtils.DlgDB_Connection{'Update'};
If gvDatabase
...
Else
   MsgBox 'User did not select a database';
EndIf
```

In this example, the user is presented with a dialog box to select a data source or database file. If the user selects one and it opens successfully, the database object is returned to the calling script.

Function DBTableExists

This function returns True if the specified table exists in the database and False otherwise.

Format:

```
Set gvPresent = eDBUtils.DBTableExists{databaseObject,tableName};
```

Where **databaseObject** is a database object variable and **tableName** is a string value specifying the name of the table.

Example:

```
New ScriptVar NewVar(eDBUtils) File('DBUtils');
Set gvPresent = eDBUtils.DBTableExists{gvDatabase,'MyTableName'};
If gvPresent
   MsgBox 'The Table is present';
Else
   MsgBox 'The Table is not present';
EndIf
```

Chapter 9

Events

Overview

Events in FrameScript are the same as subroutines except that they are meant to be Run by FrameScript itself (or FrameMaker) and not Run directly in a script by the Run command. Events are known as Callback subroutines. Also, they have a fixed number and type of parameters determined by the event type. The format for an event declarataion is similar to the Sub command, except that you do not include a parameter list.

Some event names are pre-defined, such as those involving notification of FrameMaker events. FrameMaker uses these names and runs the events as it chooses.

With other events you choose the name of the event yourself. These occur when you define menu commands and in other places as well.

Event EventName

The **`Event EventName`** command indicates the start of a new procedure that will process a FrameMaker event. Events include responses to FrameMaker Events (notifications) (such as before and after opening a document) and menu comamnd EventProcs.

Format:

```
Event eventname
. . .
EndEvent
```

In between the Event and EndEvent lines there can be any number of commands.

Table 29: Event Options

Option Name	Option Description
`eventname` *(Required)*	The name of the event, either predefined as in notifcations or a user defined name as in menu command procedure.
`EndEvent` *(Required)*	The command that terminates an event.

Example:
The following script illustrates a notification event: This one displays a message showing the file name whenever someone opens a document.

```
. . .
Event NotePreOpenDoc
   MsgBox 'The user just opened a document='+FileName;
EndEvent
```

Predefined Events

When you create a menu item, you specify the name of an event that you wish to run when the user clicks on the menu item. There are other FrameMaker events that occur during a FrameMaker session that you may respond to, if you choose. Responding to these events is optional. If you wish to have a FrameScript event run whenever one of these predefined FrameMaker events occur, all you have to do is declare the event with the reserved event name that corresponds to the event you wish. See Appendix B for a complete list of predefined events. Event of this type are passed three parameters some of which may not apply to all events of this type. The parameters are as follows:

`FrameDoc`	The document object which was active when the event occurred.
`FileName`	The name of the filename for file type events.
`IParm`	A special parameter passing the f-code for certain functions.

These events are run *as if* they were subroutines run with the following command:

```
Run eventname FrameDoc(ActiveDoc) Filename(filenamestring) IParm(fcode);
```

See the table to see which parameters are valid for which events.

For example, the following script fragment, shows how to display a message on the screen before a any document is opened.

```
Event NotePreOpenDoc
   MsgBox 'Somebody just opened a document named-'+Filename;
EndEvent
```

The name `NotePreOpenDoc` tells FrameScript that you wish to run this event just before any standard FrameMaker document is opened.

Note: See the FrameMaker reference for a complete list of predefined event names.

Hypertext Events

A Hypertext event is similar to the above events, except that this event is run when the user presses a hypertext marker in a document. The name of the event is `Message`. If you specify a Message event, this event will be run whenever the user clicks a hypertext marker with a marker text in the following form:

```
message fsl scriptname message
```

The message is the hypertext command name. fsl is the name of the FrameScript client and scriptname is the name of the script to receive the message. This is one difference between the other notification events and the message event.

Many scripts (as well as other clients) can receive the same notification event. It will occur one after the other. But only one script (or client) will receive a message event.

Format:

```
Event Message
. . .
EndEvent
```

The following parameters are passed to a message event.

FrameDoc	The document object of the document containing the hypertext marker.
FrameObject	The object variable of the marker causing the hypertext event.
Message	A string variable containing the message in the marker text.

CanTerminate Function

In Event Scripts, you can have an optional function called CanTerminate. This function is called by FrameMaker before the Terminate function to determine whether the script can exit. This should only happen if the script is keeping some data that has not been saved yet. For example, if you have unsaved data in a form, you might want to ask the user to save, don't save or cancel. If the user chooses Cancel then the CanTerminate function should return False. If this function is not present in an event script, then FrameScript will assume that it can be safely terminated, which is like the behavior of earlier releases. See "User Functions" on page 58.

> **IMPORTANT:** Do not include this function in your script unless you have some reason to cancel the termination of a script. If you accidentally have an error where it always returns False, then the script will never be able to be uninstalled. You will probably have to quit FrameMaker with the Task Manager under MS Windows!

Format:

```
Function CanTerminate
    Set Result = True;

    ...
EndFunc
```

Chapter 10

Script Commands

Install Script

The **Install Script** command registers a script with FrameScript. If the script is an event script the script is loaded into memory and the 'Initialize' Event is run. If it is a standard script, the menu item (under the FrameScript -> Scripts menu) is created as a shortcut to run this script. Note: the user may also install a script via a menu command.

Format:

```
Install Script File(scriptfilename) [Name(scriptname)]
    [Label(menulabel)] [Shortcut(string)] When(whenenabled);
```

Table 30: Install Script Options

Option Name	Option Description
File *(Required)*	A string value containing the name of the script file.
Name	The internal name of the script. If not specified this name will be created by FrameScript. It is used to uninstall the script (if desired) and it is also used to identify the script for message events.
Label	The menu label for standard scripts.
Shortcut	A string value specifying the keyboard shortcut. The text for the shortcut has the following forms: For Alt-Fn (where Fn is any function key) use **'~/Fn>'** For Ctl-n (where n is any letter) use **'^n'** For Ctl-Fn (where Fn is any function key) use **'^/Fn>'** For Sft-Fn (where Fn is any function key) use **'+/Fn>'** For the standard (platform independent) Frame shortcuts use **'\!mn'**, where nn is a series of letters. This shortcut is activated by typing in ESC mn, where m and n are the two letters chosen.
When or EnabledWhen	This value determines when the menu command is available. Select from one of the values in the enabled-when table, see "EnabledWhen Context Descriptions" on page 81. If omitted, then the default value will be used, which means it will always be enabled.

The following table lists the values EnabledWhen can have and the corresponding contexts in which a menu item is active .

Table 31: EnabledWhen Context Descriptions (Page 1 of 2)

EnabledWhen value	Command Context Description
EnableAlwaysDisable	No context. The menu item is disabled. If a menu item is enabled and you set EnabledWhen to this value, it disables and dims the menu item.
EnableAlwaysEnable	All contexts. This is the default value. If the menu item is disabled, setting EnabledWhen to this value enables it.

Table 31: EnabledWhen Context Descriptions (Page 2 of 2)

EnabledWhen value	Command Context Description
EnableCanPaste	The Clipboard contains an object or text that can be pasted at the insertion point.
EnableCopy	Some text or an object is selected.
EnableCopyFont	The insertion point or selection is in the text of a paragraph, a math object, a table, or a text line.
EnableInCellText	The insertion point or selection is in a table cell.
EnableInFlow	A text frame is selected, or the insertion point or selection is in a paragraph.
EnableInMath	The insertion point or selection is in a math object.
EnableInParaText	The insertion point or selection is in a paragraph (but not in a math object).
EnableInTable	The insertion point or selection is in any part of a table.
EnableInTableTitle	The insertion point or selection is in the table title.
EnableInText	The insertion point or selection is in a graphic text line or a paragraph.
EnableInTextLine	The insertion point or selection is in a graphic text line.
EnableIsAFrame	The first selected object is an anchored frame.
EnableIsCell	A single cell in a table is selected.
EnableIsCells	One or more cells in a table are selected.
EnableIsGraphicInset	The first selected object is a graphic inset.
EnableIsObj	An object is selected.
EnableIsOrInFrame	The selected object is a graphic frame or is in a graphic frame that is not a page frame.
EnableIsTables	An entire table is selected.
EnableIsTextFrame	A text frame is selected.
EnableIsTextInset	The first selected object is a text inset.
EnableIsTextSel	The selection is in a paragraph.
EnableIsViewOnly	The current document is locked.
EnableNeedsBookpOnly	A book is open.
EnableNeedsDocpOnly	A document is open.
EnableObjProps	The insertion point is in text, a table, or a math object, or a graphic object is selected.
EnableNeedsDocPOrBookP	A document or a book is open.
EnableBookHasSelection	The book has a selection made.
EnableDocOrBookHasSelection	A document is in front or a book has a selection.

Example 1:
The example installs the event script called tabletest.fsl.

```
Install Script File('tabletest.fsl') Name('TableTest');
```

Example 2:

The example installs the standard script called insertvar.fsl and supplies a label for the Scripts menu with the keyboard shortcut (ESC a b). The menu will be enabled only when the insertion point is in a text object (paragraph or text line).

```
Install Script File('insertvar.fsl') Name('InsertVariable')
   Label('Insert a Variable Here') ShortCut('\!ab') When(EnableInText);
```

Uninstall Script

The **`Uninstall script`** command deregisters a script from the FrameScript system. If the script is an event script the 'terminate' event is run first then the script is removed from memory. If it is a standard script, the menu item (under the FrameScript -> Scripts menu) is removed. Note: The user may use a menu item to uninstall a script.

Format:

```
Uninstall Script Name(scriptname);
```

Table 32: Uninstall Script Options

Option Name	Option Description
`Name` *(Required)*	The internal name of the script. Every script has a name. If it wasn't given during the install command, FrameScript made up a name for it. If you plan to uninstall a script you need to give it a name so you can use it here to uninstall it.

Example:

The example uninstalls the event script called `TableTest`, which was previously installed.

```
Uninstall Script Name('TableTest');
```

Exec Compile Command

The Exec Compile command allows you to programmatically compile scripts. You can write a script to compile all your other scripts. This is useful if you have many scripts and you need to re-compile all of them.

Format:

```
Exec Compile FileName(fileNameString) OutputFileName(outFileNameString);
or
Exec Compile ScriptText(stringValue) OutputFileName(outFileNameString);
```

Table 33: Exec Compile Options

Option Name	Option Description
`FileName`	A string value containing the name of the script file.
`ScriptText`	A string value containing the text of the script. You must use one of FileName or ScriptText
`OutputFileName`	The internal name of the script. If not specified this name will be created by FrameScript. It is used to uninstall the script (if desired) and it is also used to identify the script for `message events`.

Exec Script Command

The Exec Script command allows to start and run another script. This new script runs in its own data space, so there is no sharing of variables or data between script running the command and the script that is started. This is for standard scripts only

Format:

```
Exec Script FileName(fileNameString);
or
Exec Script ScriptText(stringValue);
```

Table 34: Exec Script Options

Option Name	Option Description
`FileName`	A string value containing the name of the script file.
`ScriptText`	A string value containing the text of the script. You must use one of FileName or ScriptText

Chapter 11

Working with Text Files

New Textfile

The **New Textfile** command creates a new text data file.

Format:

```
New Textfile File(filename) NewVar(filevar) IOType(ftype);
```

Table 35: New Textfile Options

Option Name	Option Description
File	The file name to create.
IOType	WriteOnly Append.
NewVar	The name of the variable to hold the newly created text file object.

Example:

This example creates a text file called 'c:\temp\test.txt' and writes out two text lines before closing it.

```
New Textfile file('c:\temp\test.txt') NewVar(filevar) IOType(WriteOnly);
write Object(filevar) 'Write line 1 out to File';
write Object(filevar) 'Write line 2 out to File';
Close textfile Object(filevar);
```

Open Textfile

The **Open Textfile** command opens an existing text data file.

Format:

```
Open Textfile File(filename) NewVar(filevar) IOType(ftype);
```

Table 36: Open Textfile Options

Option Name	Option Description
`File`	The file name to create.
`IOType`	`ReadOnly WriteOnly Append.`
`NewVar`	The name of the variable to hold the opened text file object.

Example:
This example opens a text file called ’c:\temp\test.txt’ and writes out two text lines before closing it.

```
Open Textfile file('c:\temp\test.txt') NewVar(filevar) IOType(Append);
write Object(filevar) 'Write line 1 to the end of the File';
write Object(filevar) 'Write line 2 to the end of the File';
Close textfile Object(filevar);
```

Read command

The **`Read`** command reads a line of text from a text file.

Format

```
Read File(filevar) NewVar(stringvar) [Rewind];
```

Table 37: Read Options

Option Name	Option Description
`File`	The file variable of the file to read.
`NewVar`	A string variable where the text line will be placed.
`Rewind`	Specifies that the text file will be positioned at the beginning.

Use the errorcode variable to determine the results of the read command. If the value is zero then it worked. If it is positive, then it reached the end of the file. A negative value is (as usual) an error.

Example:
This example opens a text file called ’`C:\FrameScript\test.txt`’, reads each line of the file and writes the text to the console. Then, it rewinds the file to the beginning and does it again.

```
Set infname = 'C:\FrameScript\test.txt';
Set errorcode = 0;
Open Textfile File(infname) NewVar(tfile) IOType(ReadOnly);
If errorcode not = 0
   Display 'open error on text file ' + errormsg;
   LeaveSub;
EndIf

Read File(tfile) NewVar(tbuf);
Loop While (errorcode = 0)
   Write console tbuf;
   Read File(tfile) NewVar(tbuf);
EndLoop

Read File(tfile) Rewind;
Read File(tfile) NewVar(tbuf);
Loop While (errorcode = 0)
   write console tbuf;
   Read File(tfile) NewVar(tbuf);
EndLoop

Close TextFile Object (tfile);

Display 'Done';
```

Write Command

The Write command allows you to write a string expression to one of three output targets: the FrameMaker console, the FrameMaker display (dialog box), or a textfile.

Format:

```
Write {console display object(filevar)} Stringexpression;
```

Table 38: Write Options

Option Name	Option Description
Destination *(Required)*	The destination of the output string. This can be console to write the FrameMaker console, display to have FrameMaker display a dialog box and object(filevar) to write the string expression to a text file..
stringexpression	Any expression which can be converted into a string..

Example:

This example write two messages to the console, then creates and writes two lines to a text file called `ofile.txt`.

```
Write console 'I am going to write a string to the console';
Write console 'Then I will write to a text file';
New TextFile NewVar(filevar) File('ofile.txt');
Write object(filevar) 'Write a line to a file';
Write Object(filevar) 'Write another line to this file';
Close textfile Object(filevar);
```

Close Textfile

The **`close textfile`** command closes a textfile.

Format:

```
Close TextFile Object(fileobjectvar);
```

Table 39: Close Textfile Options

Option Name	Option Description
`Object` *(Required)*	The object variable of their textfile to close.

Example

The following code closes the active document, ignoring any unsaved changes:

```
. . .
Close Textfile Object(filevar);

. . .
```

Chapter 12

List Commands

Introduction

Certain data types are list types (StringList, IntList, MetricList, UIntList). These contain fixed types of data (strings, integers, metrics, Unsigned Integers, respectively). The members of these list types can be accessed by the indexing operator ([]), as follows:

```
New StringList NewVar(gvMySL) Value('String1') Value('String2') Value('String3');
Display 'The second string is '+gvMySL[2];
```

> **Note:** These list types are used primarily for fixed length lists. FrameScript provides several other array types that are more flexible for updating. They are Object data types and can contain different types of data. See the EslObject referemce for more information on the FrameScript array types.

You use the New command to create list variable and the Delete Var command to remove them.

The following commands are provided for creating, accessing and updating members of these list types.

New List Data types

New IntList
New UIntList
New MetricList
New StringList
Format:

```
New {IntList UIntList MetricList StringList} NewVar(varname)
   Value(expression) value(expression) ... value(expression);
```

The new list data types commands allow you to create list variables. MetricLists are used by FrameMaker for various lists of measurements, such as the widths of Table columns. StringLists are used for various name lists, including font lists and dialog scroll box lists. UIntlists are used to record f-code lists.

Table 40: New List Data Type Options

Option Name	Option Description

`Data type`	The type of data to create. Possible values are:
	`IntList`
	`UIntList`
	`MetricList`
	`StringList`

`Value`	The value of each member of the new variable. If it is not the same type as the target data, it will be converted, if possible.
`NewVar`	The name of the variable to hold the newly created text file object.

Example 1:

This sample script sets the column widths of the first table in the document to 1 inch, 2 inches and 1.5 inches respectively.

```
New MetricList NewVar(gvColumnWidths) Value(1”) Value(2”) Value(1.5”);
set tableobj = FirstTblInDoc;
set tableobj.TblColWidths = gvColumnWidths;
Delete Var(gvColumnWidths);
```

Example 2:

This sample script creates a list of names, then displays them in a scroll box.

```
New StringList NewVar(gvNameList) Value('George Washington')
    Value('John Adams') Value('Thomas Jefferson');
DialogBox Type(ScrollBox) Title('President Selection Dialog')
    Caption('Select a president')
    Init(-1) List(gvNameList)
    NewVar(gvPresidentName) Button(gvButton);
```

Add Member

The **`Add Member`** command adds an individial member to a list data item.

Format

```
Add Member(membervalue) To(listvar)
    [After(memberNumber)] [Before(memberNumber)];
```

Table 41: Add Member Options

Option Name	Option Description
`Member`	The value to add to the list. Should be the same type as the other list members
`To` *`(Required)`*	Specifies the list variable name.

Table 41: Add Member Options

Option Name	Option Description
After	Specifies the member number after which the new item will be added.
before	Specifies the member number before which the new item will be added. If neither After nor Before is specified, then the new member will be added to the end of the list.

Example

The following code adds the new member to the string list:

```
. . .
New StringList NewVar(slistvar)
   Value('Apples') Value('Oranges') Value('Peaches');
....
Add Member('Grapes') To(slistvar);
. . .
```

The list will now contain four values ’Apples’, ’Oranges’, ’Peaches’, and ’Grapes’.

Find Member

Format:

```
Find Member(MemberValue)
         InList (Listvariable)
         [Binary]
         [Indirect(indList)]
         ReturnStatus(TrueFalsevarname)
         ReturnPos(positionvarname);
```

Table 42: Find Member (InList) Options

Option Name	Option Description
Member *(Required)*	The value to search for. This should match the type of data in the list.
InList	The name of the list variable in which to search.
Binary	This tells FrameScript to perform a binary search. For long lists (over 300 items), this will be faster than a sequential search, but it requires that the list be in ascending order. See “Sort command” on page 95. The default is to perform a sequential search.
indirect	This tells FrameScript to use an IntList to indirectly indicate the order of the items to search. See “Sort command” on page 95.
ReturnStatus	Variable name. This is true if the string was found, false otherwise
ReturnPos	The position of the member in the list, if found.

Example 1:

This example searches the string list for the value 'STU'. It returns the position of the item in the list, which, in this case, is 2.

```
New StringList NewVar(slist) value('ABC') Value('STU') value('XYZ');
Find Member ('STU') InList(slist)
   ReturnStat(found) ReturnPos(posvar);
MsgBox 'Item found at position '+posvar+' in the list';
```

Example 2:

This example performs the same search as above, except it performs a binary search. The result is the same, but if the list had been very long (over 300 items) it would run faster.

```
New StringList NewVar(slist) value('ABC') Value('STU') value('XYZ');
Find Member ('STU') InList(slist) Binary
   ReturnStat(found) ReturnPos(posvar);
MsgBox 'Item found at position '+posvar+' in the list';
```

Example 3:

This example shows an indirect binary search. This gets names of all the fonts, sorts the list indirectly (it creates an integer list which points to the original list).

```
Set pList = FontFamilyNames;
Sort List(pList) case Indirect NewVar(pListSorted);
Find Member('Helvetica') InList(pList) Binary Indirect(pListSorted);
```

Get Member

The **Get Member** command gets an individial member from a list data item.

Format

```
Get Member [Number(membernumber)] From(listvar) NewVar(varname);
```

Note: This command is now obsolete. It is kept for backward compatiblilty with previous versions. You can access members easier and more efficiently using the index operator ([]).

Table 43: Get Member Options

Option Name	Option Description
Number	The integer number of the member in the list. The first member is 1.
From *(Required)*	Specifies the list variable name.
NewVar	Specifies the name of the variable to put the member.

Example 1:

The following code gets the specified member from the string list:

```
. . .
New StringList NewVar(gvStringList)
   Value('Apples') Value('Oranges') Value('Peaches');
. . .
Get Member Number(2) From(gvStringList) NewVar(gvString);
Display gvString;
. . .
```

The value of `gvString` will be ’Oranges’.

Example 2:

This is the same as above except this uses the indexing operator instead of the Get member command:

```
. . .
New StringList NewVar(gvStringList)
   Value('Apples') Value('Oranges') Value('Peaches');
....
Display gvStringList[2];
. . .
```

The value displayed will be ’Oranges’.

Remove Member

The **`Remove Member`** command removes an individial member from a list data item.

Format

```
Remove Member[(membervalue)] [Number(membernumber)] From(listvar);
```

Table 44: Remove Member Options

Option Name	Option Description
`Member`	Value of the member to remove.
`Number`	The integer number of the member in the list. The first member is 1.
`From` *(Required)*	Specifies the list variable name.

Example 1

The following code removes member number 2 in the string list:

```
. . .
New StringList NewVar(slistvar)
   Value('Apples') Value('Oranges') Value('Peaches');
. . .
Remove Member Number(2) From(slistvar);
. . .
```

The list will now contain two members, ’Apples’ and ’Peaches’.

Example 2

The following code removes the member ’Peaches’ from the string list:

```
. . .
New StringList NewVar(slistvar)
   Value('Apples') Value('Oranges') Value('Peaches');
. . .
Remove Member('Peaches') From(slistvar);
. . .
```

The list will now contain two members, ’Apples’, and ’Oranges’.

Replace Member

The **Replace Member** command replaces an individial member from a list data item.

Format

```
Replace Member[(membervalue)] [Number(membernumber)] In(listvar) with(newmembervalue);
```

Note: This command is now obsolete. It is kept for backward compatiblilty with previous versions. You can replace members easier and more efficiently using the index operator ([]).

Table 45: Replace Member Options

Option Name	Option Description
Member	Value of the member to replace.
Number	The integer number of the member in the list. The first member is 1.
In *(Required)*	Specifies the list variable name.
With	Value of replacement member.

Example 1:

The following code replaces member number 2 in the string list:

```
. . .
New StringList NewVar(gvStringList)
   Value('Apples') Value('Oranges') Value('Peaches');
. . .
Replace Member Number(2) In(gvStringList) With('Grapes');
. . .
```

The list will now contain three members, ’Apples’, ’Grapes’, and ’Peaches’.

Example 2:
The following code replaces member number 2 in the string list. This is the same as the above except that it uses the indexing operator instead of the Replace Member command

```
. . .
New StringList NewVar(gvStringList)
   Value('Apples') Value('Oranges') Value('Peaches');
. . .
Set gvStringList[2] = 'Grapes';
. . .
```

Example 3:
The following code replaces member ’Peaches’ in the string list with ’Grapes’:

```
. . .
New StringList NewVar(gvStringList)
   Value('Apples') Value('Oranges') Value('Peaches');
....
Replace Member('Peaches') In(gvStringList) With('Grapes');
. . .
```

The list will now contain three members, ’Apples’, ’`Oranges`’, and ’Grapes’.

Sort command

The **`Sort`** command allows you to sort the items in a StringList or IntList variable.

Format:

```
Sort List(listVar) [NewVar(sortedList)]
   [Ascending | Descending] [Nocase] [Indirect]
```

Table 46: Sort Options

Option Name	Option Description
`List` *(Required)*	The name of an existing StringList or IntList variable. This identifies the list to be sorted.
`Ascending`	Sort the list in normal character set order. This is the default value.
`Descending`	Sort the list in reverse character set order.
`Nocase`	Do a case insensitive sort (for StringList type only).
`Indirect`	This tells the sort routine to do an indrect sort. This means that instead of producing a sorted list, it produces an IntList which contains the relative numbers of the items in sorted order. This is useful, if you don’t want to change the order of original order.
`NewVar`	The name of the variable to put the result of the sort. If not specified, the list is sorted and the result placed back into the same variable.

Example:
The following script creates and sorts the values in a string list.

```
. . .
New StringList NewVar(slist) value ('CCC') value('AAA') Value('BBB');
Sort List(slist);
. . .
```

Chapter 13

Miscellaeous Commands

Find String

The **Find String** command searches for a string in another string.

Format:

```
Find String(SearchString)
          InString (StringToSearch)
          [Start(integerexpression)]
          [NoCase WholeWord Backward Prefix Suffix]
          [ReturnPos(posvarname)]
          [ReturnStatus(TrueFalsevarname)]
          [ReturnString(strvarname)];
```

Table 47: Find String Options

Option Name	Option Description
String *(Required)*	The string to search for. This could be a string variable or a string constant or an expression combining the two.
InString *(Required)*	The string in which to search. This could be a string variable or a string constant or an expression combining the two.
Start	Character position in the string to start searching. The first character is 1. Default: start at beginning of string.
NoCase	Case insensitive (default-case sensitive)
WholeWord	Find whole word only
Backward	Search the string backward.
Prefix	Find the string only if it is the prefix of the InString string
Suffix	Find the string only if it is the suffix of the InString string
ReturnPos	Variable name. This will contain the starting position of the found string. (zero if not found)
ReturnStatus	Variable name. This is true if the string was found, false otherwise
ReturnString	This is the value of the string found. This might be different from the search string, if NoCase was specified.

Example:

This example attempts to find the string ’The’ in the string ’Now is the Time’.

```
Find String ('The') InString('Now is the Time') NoCase
   ReturnPos(posvar) ReturnStat(found) ReturnString(strvar);
```

After this command runs it, the ’posvar’ variable contains the value 8, the ’found’ variable contains the value True, and the strvar variable contains the value ’the’.

Get String

The **Get String** command retrieves a sub string from another string. The string can be modified as it is retrieved (e.g. uppercase). You can use this command to perform many string operations.

Format:

```
Get String FromString(TargetString)
        [StartPos (startcharacterposition)]
        [EndPos (lastcharacterposition)]
        [RemoveLeading('LeadingCharactersToRemove')]
        [RemoveTrailing('TrailingCharactersToRemove')]
        [RemoveChars('CharactersToRemove')]
        [UpperCase]
        [LowerCase]
        [ReplaceFirst(Str1) With(Str2)]
        [ReplaceLast(Str1) With(Str2)]
        [ReplaceAll(Str1) With(Str2)]
        [Reverse]
        NewVar(varname)
```

Table 48: Get String Options

Option Name	Option Description
FromString *(Required)*	The string to search. This could be a string variable or a string constant or an expression combining the two.
StartPos	The starting character position.
EndPos	The ending character position
RemoveLeading	A string containing a list of characters to remove from the beginning of the retrieved string.
RemoveTrailing	A string containing a list of characters to remove from the end of the retrieved string.
RemoveChars	A string containing a list of characters to remove from the retrieved string.
UpperCase	Converts the characters to upper case.
LowerCase	Converts the characters to lower case.
ReplaceFirst	The first occurence of this string is located and replaced with the corresponding With string. You may have zero or more of these sets of options.
ReplaceLast	The last occurence of this string is located and replaced with the corresponding With string. You may have zero or more of these sets of options.
ReplaceAll	All occurences of this string are located and replaced with the corresponding With string. You may have zero or more of these sets of options.
With	This identifies a replacement string for a preceding ReplaceFirst, ReplaceLast or ReplaceAll option.
Reverse	Reverses all the characters in the string
NewVar *(Required)*	This is the returned string.

The order of the string operations is the same order as they are listed above, if they are specified. The StartPos and EndPos create a substring first (if specified), then the RemoveLeading, RemoveTrailing and RemoveChars occur next, followed by the Upper and/or Lower casing. The ReplaceFirst, ReplaceLast, ReplaceAll groups go next, followed by the Reverse option. If any of these options are not specified, then that step is skipped.

Example 1

The following code pulls a substring (characters 6 through 9) from the original string and changes the characters to uppercase.

```
. . .
Set TestString = 'This is a test string';
Get String FromString(TestString) NewVar(extractstr)
   StartPos(6) EndPos(9) Uppercase;
```

The value of 'extractstr' would be `'IS A'`.

Example 2

The following code pulls a substring (starting with character 3) from the original string, and removes the spaces from the string.

```
. . .
Set TestString = 'This is a test string';
Get String FromString(TestString) NewVar(extractstr)
   StartPos(3) RemoveChars(' ');
```

The value of 'extractstr' would be `'isisateststring'`.

Example 3

The following code pulls a substring (starting with character 3) from the original string, and removes the spaces from the string.

```
. . .
Set TestString = '    my name    ';
Get String FromString(TestString) NewVar(extractstr)
    RemoveLeading(' ') RemoveTrailing(' ') Uppercase;
```

The value of 'extractstr' would be `'MY NAME'`.

Example 4

The following code replaces the first AA string with BB, replaces all periods(.) with commas(,) and the QAZZAQ with ??.

```
. . .
Set TestString = 'AAAA1234.45QAZZAQ';
Get String FromString(TestString) NewVar(extractstr)
   ReplaceFirst('AA') With('BB')
   ReplaceAll('.') With(',')
   ReplaceAll('QAZZAQ' With('??');
```

The value of 'extractstr' would be `'BBAA1234,45??'`.

Exec Wait Command

The Exec Wait command allows you to stop running for a specified amount of time. This allows you to stop execution of the current script so that some other program can run and, perhaps, produce some data that you need.

Format:

```
Exec Wait Seconds(NumberOfSecondsToWait);
or
Exec Wait MicroSeconds(NumberOfMicroSecondsToWait);
```

Table 49: Exec Wait Options

Option Name	Option Description
`Seconds`	An integer value specifying the number of seconds to wait.
`MicroSeconds`	An integer value specifying the number of micro-seconds to wait

Chapter 14

List of Error Messages

The following table list the possible error codes and message resulting from FrameScript commands. A zero value indicates that there was no error. All errors have negative values. Error codes with values between -1 and -999 are FrameMaker generated error codes. Those from -1000 through -3000 are FrameScript error codes.

IMPORTANT: Some errors are unresolvable such as memory errors or internal errors. Many errors, however, are just codes that inform you of the result of the command. You may want to do a **`Get Object`** command just to see if a particular object name exists. If it returns an negative error code, then that's just a normal part of the script's function. You continue on as normal, perhaps by adding the name with the New command. The point is that all errors are not bad or cause problems. They just tell you what heppened. Some can safely be ignored if you understand the function ahead of time.

.

Table 50: List of Error Messages

Errorcode value	Error Message Description
0	No error
-1	Communications between FrameMaker and its clients is failing. This is an internal FrameMaker error.
-2	Invalid Document or Book object specified
-3	Invalid FrameMaker Object specified
-4	Current object doesn't have this property
-5	Property's type different than requested
-6	Can't write into this property
-7	Value not in legal range for property
-8	Closing modified doc without the IgnoreMods option
-9	Can't select/deselect object in group
-10	Must implicitly move between frames first
-11	Value must be an Object of a Graphic object
-12	Value must be an Object of a Frame object
-13	Value must be an Object of a Group object
-14	Can't move given object to this Frame
-15	Can't move given object to this Group
-16	Can't make this prev/next connection

Table 50: List of Error Messages

Errorcode value	Error Message Description
-17	Can’t delete this kind of object
-18	Can’t delete this page
-19	Wrong type for Get Object command
-20	Bad name for Get Object command
-21	Can’t find requested offset
-22	Some XRefs or Text Insets were unresolved
-23	Bad New object command
-24	Expecting Object of a Body Page object
-25	Expecting Object of a Pgf object
-26	Expecting Object of a Book Component object
-27	A general error for any bad command
-32	A same type item of this name exists
-33	Trying to give an object an illegal name
-34	Can only compare book to book or doc to doc
-35	Compare operation failed
-36	Two ends of range not in same flow or hidden
-37	PageFrames can’t be moved or selected
-38	Can’t smooth/unsmooth this object
-39	Value must be an Object of a TextFrame object
-40	Value must be an Object of a non hidden page
-41	Expecting Object of a Pgf, TextLine, Flow, Cell, TextFrame, SubCol, Fn, XRef, Var, TiFlow, TiText, TiTextTable, TiApiClient
-42	Unable to open the document due to system error.
-43	Parameter passed to an command was invalid.
-44	User canceled operation. The command required user intervention and the user canceled it
-45	Document was in an inconsistent state.
-50	Invalid file name on a save command
-58	String value is invalid for this operation
-59	Text Selection in docuement is not valid for operation
-60	Can’t access this object type
-65	Bad insertion position
-66	Bad book Object specified
-67	Book is unstructured
-68	Bad book component path specified

Table 50: List of Error Messages

Errorcode value	Error Message Description
`-70`	File was closed by an apiclient when it processed a notification.
`-71`	Expecting Object of a Pgf or Flow
`-72`	Expecting Object of a Menu
`-73`	Expecting Object of a Command
`-74`	Expecting Object of a Command defined by an api client
`-75`	Menu item (Command or Menu) is not in menu
`-76`	Expecting a valid keyboard shortcut
`-77`	Expecting a menu to contain menus only
`-81`	Importing document would cause a circular reference.
`-82`	Requested flow did not exist in the source document.
`-83`	The type of the file on disk was not the type of file the import operation expected. Or the type that the **`Update TextInset`** command expected based on the inset was invalid
`-84`	The file no longer exists on disk
`-85`	The Inset is a Mac Edition, but we aren’t running on a Mac.
`-86`	An API Client or a script canceled the operation
`-87`	Object has no text in it
`-88`	FM not in safe state for asynchronous invocation. This is an internal FrameMaker error.
`-89`	A filter that was filtered (input or output) failed
`-90`	Asian capable system required
`-91`	Can’t change tinted color this way
`-92`	Can’t Set Ink Name without Color Family
`-93`	String exceeds max length for property, truncated
`-94`	Internal code to move Graphic Inset data from current document to a file has failed, leaving user with a file with missing data. This is an incomplete, unsuccessful Save.
`-2003`	Parameter Error
`-2201`	Missing Script
`-2202`	Missing Script File
`-2203`	Command Error
`-2205`	Compile Error
`-2402`	Bad I/O
`-2403`	File Seek Error
`-2404`	Missing File
`-2405`	Missing Subroutine
`-2406`	Missing Script
`-2407`	Missing Required Parameter

Table 50: List of Error Messages

Errorcode value	Error Message Description
-2408	Parameter is invalid for this command
-2409	Error during a file operation
-2502	Variable Not Found
-2504	No Variable
-2505	Wrong Data Type
-2506	Invalid Property
-2509	Expression Error
-2510	Invalid Operation
-2511	Invalid Data Type
-2521	Invalid Object for this command
-2522	Missing Object
-2523	Read Only Variable
-2525	Cannot make new variable
-2526	Qualifiers Present
-2527	Invalid Property
-2528	Value out of range
-2529	Item not found
-2801	FrameScript Memory Failure

Chapter 15

Common Script Errors

It's impossible to list all of the things that can go wrong in writing any script in any script language. The following list gives some common errors types to look for when things go wrong.

> **IMPORTANT:** When writing and testing scripts, it is imperative that you use copies of your documents to test the scripts before putting them into production with real data. It is very easy to make mistakes during script development. All measures should be taken to assure that a script is working correctly before usnig them on live documents.

Reserved Words

One common error is using reserved words are variable names. Any occurrence of a FrameScript command name automatically stops the current command and starts a new one. Using a command name as a variable name will cause all sorts of compiling problems.

EndIf, EndLoop, EndSub, EndEvent

Each If command, Loop command, Sub command and Event command starts a block of FrameScript commands. These blocks must be terminated by the appropriate EndXXXX command. For example, a common error is forgetting to put in the EndIf command at the end of a list of commands under an If command.

It is useful (but not necessary) to indent the commands under any of these block start type of commands. This makes reading the script easier.

Logic Errors

The most common error is simple logic errors. Commands are executed one at a time, one after the other until the end of a list of commands is reached. Following the logic step by step is usually enough to solve these types of errors.

Check Error Codes

Another common error is not checking an error code after a command. You cannot always assume that an open document command will work (perhaps the file no longer exists, or it has a warning and the user elected to cancel the operation) An invalid text insertion point is a common mistake. For a list of error codes see FrameMaker Reference.

Read-only variables

Some variables (global) and properties are marked as read-only. This means that you may use these variables in commands and in computations but you may not try to change the value.

Run Away Scripts

If you are running a script that is taking a long time, it may be in a run away condition (an infinite loop using programmer terminology). This means that due to some logic error the script will never stop without stopping FrameMaker or re-booting the computer. If this happens, you can press the ESC key to interrupt the script. Of course, you can also do this if the script is just taking too long.

Chapter 16

Frame Architecture

Object Lists

The FrameMaker product keeps many lists of objects. Some objects have properties which act as the start of a set of other objects. This means that they have a property which contains an object variable representing another FrameMaker object. This other FrameMaker object variable has a property which indicates the next object in the list. These object lists have the following general form: `FirstXXXXInYYYY` and `NextXXXXInYYYY`, where `XXXX` is the name of the object in the list and `YYYY` is the name of the head object. For example, a document has a property called `FirstPgfInDoc`, which contains an object variable representing the first paragraph object in the document. The paragraph itself has a property called `NextPgfInDoc`, which contains an object variable representing the next paragraph in the document. FrameScript provides an easy way to navigate through these lists using the `ForEach` option of the **`Loop`** command. Note that the paragraph list illustrated above does not give you all the paragraphs in order in the document. It is simply a list of all the paragraphs in the document (the order appears to be the order that the paragraphs were entered). If you want a list of pargraphs in order, you have to go through the flow structures.

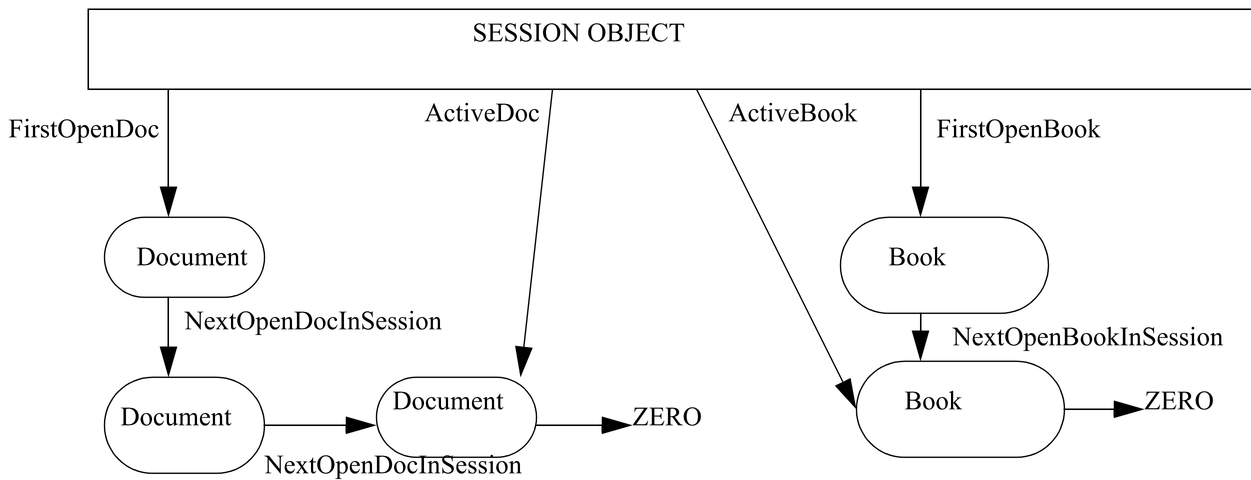
Session Object

When the FrameMaker product starts, it creates one and only one session object. This object provides information (in its properties) that are global to FrameMaker. Some of these properties are `AutoBackup` flag, `AutoSave` flag, `FontAngleNames`, etc. For a complete list of properties, see Scriptwriter's Reference. The session maintains two lists: a list of open documents and a list of open books. For the architecture the most important properties are in the following list:

Table 51: List of Session Architecture Properties

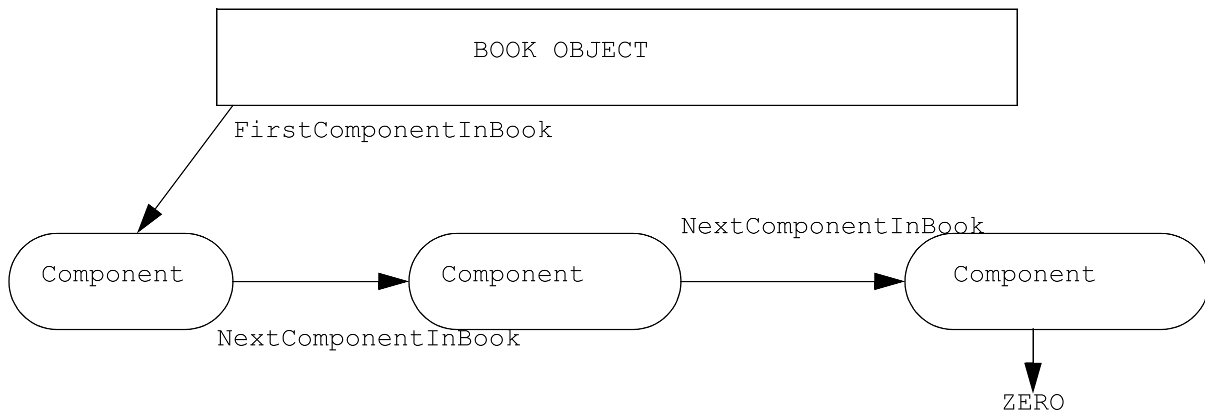
Property Name	Description
`ActiveDoc`	The object of the currently active document.
`ActiveBook`	The object of the currently active book.
`FirstOpenDoc`	The first in a list of open documents
`FirstOpenBook`	The first in a list of open books.

The following illustrates the relationship between the session and the documents and books.



Book Object

A book object contains a list of book component objects. These book components describe the global properties of the documents of a book file. See the Scriptwriter's Reference for more information about book components and for more information about books. The `FirstComponentInBook` property of the Book object gives the object for the first book component in the list of components. In the book component object, the `NextComponentInBook` property gives the next book component object. The following diagram illustrates the relationship between books and components.

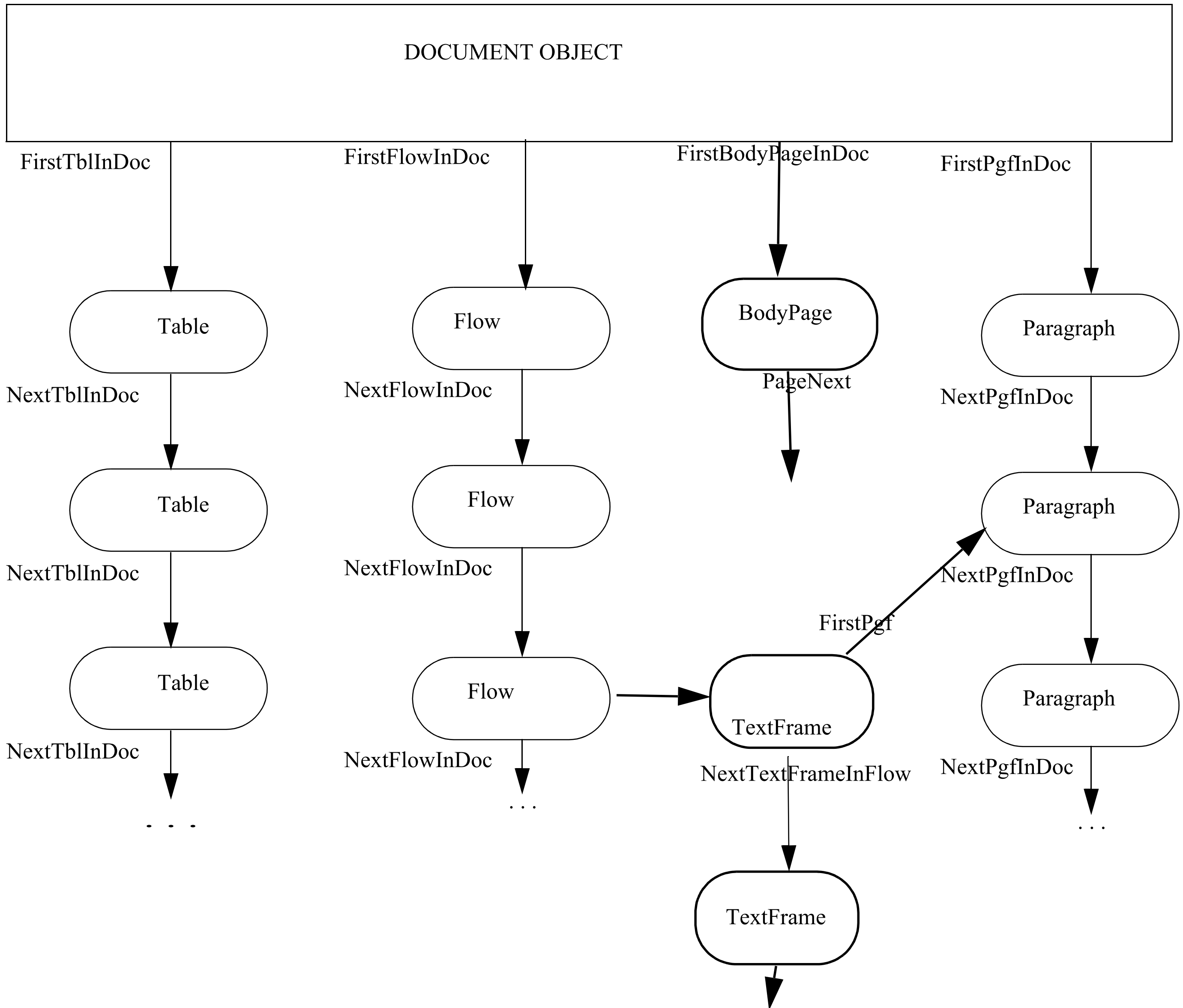


Document Object

The document object is the fundamental object in the FrameMaker system. It is the source of most of the other FrameMaker objects. A document object contains the start of many lists, which gives you access to these other objects.

These lists include a list of all the marker objects in a document, all the body pages, all the character formats, all the paragraph formats, all the flows, all the tables, and so on. For a complete list of object lists for a document see the Scriptwriter's Reference.

The following illustrates three of the lists and structures.



Body Page

The following illustrates the Body Page Object.