

F r a m e S c r i p t

Version 4.0

U s e r ' s G u i d e

ELMSOFT INC.

7954 Helmart Drive

Laurel Maryland 20723

USA

Copyright © 1997-2005 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 interoperability 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, Sun386i, SPARC, SPARCstation / Sun Microsystems, Inc.

Scintilla and SciTE Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

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 Installing FrameScript - - - - -	5
Starting the installation - - - - -	5
Summary of Steps - - - - -	5
Completion - - - - -	6
Directory Structure - - - - -	6
Installation troubleshooting - - - - -	6
Manual registration - - - - -	7
Uninstalling FrameScript - - - - -	8
3 Using FrameScript - - - - -	9
Running Scripts - - - - -	9
Run command - - - - -	9
Installing Scripts - - - - -	10
Install Menu command - - - - -	10
Uninstalling Scripts - - - - -	12
Uninstall Script Menu Command - - - - -	12
Compiling Scripts - - - - -	13
Compile Menu command - - - - -	13
Customizing FrameScript - - - - -	16
Options Menu Command - - - - -	16
Customization using the Ini File - - - - -	19
Using the Script Window - - - - -	23
Script Window - - - - -	23

Using other editors - - - - -	23
ElmEdit- - - - -	23
Batch Processing - - - - -	26
RunEslBatch - - - - -	26
Using RunEslBatch- - - - -	26
Configuring RunEslBatch - - - - -	27
Configuring RunEslBatch for Text Editors - - - - -	27

4 Writing Scripts - - - - - 31

Elements of FrameScript- - - - -	31
Standard Scripts - - - - -	31
Event Scripts - - - - -	31
Integer Constants - - - - -	42
Real constants - - - - -	43
Metric constants - - - - -	43
String constants - - - - -	43
EArray - - - - -	50
EVector- - - - -	51
ECollection - - - - -	51
Basic commands - - - - -	52
Creating and Deleting Data - - - - -	52
Built-in Dialogs- - - - -	53
Subroutines and Functions - - - - -	53
Possible Problem - - - - -	54
Local Data Space - - - - -	55
Local Command - - - - -	55
Modules- - - - -	60
Standard Script Library- - - - -	65
Function DocIsAlreadyOpen - - - - -	66
Function ForAllDocsInBook - - - - -	66
Function GetCellXY - - - - -	67
Sub AddParaToCellXY - - - - -	67
Function DlgStringDualSelect - - - - -	68
Function DlgDB_Connection - - - - -	69
Function DBTableExists - - - - -	69
Events - - - - -	70
List of Error Messages - - - - -	72

Common Script Errors - - - - - 75

5 Frame Architecture - - - - - 77

Object Lists - - - - - 77
Session Object - - - - - 77
Book Object - - - - - 78
Document Object- - - - - 78
Body Page - - - - - 79

6 Using ElmStudio - - - - - 81

Introduction - - - - - 81
Editor Menus - - - - - 81
 File- - - - - 81
 Edit - - - - - 82
 Search- - - - - 83
 View- - - - - 83
 Exec - - - - - 84
 Debug - - - - - 84
 Options - - - - - 85
 Windows - - - - - 85
 Help - - - - - 86
Customization - - - - - 88
Abbreviations file - - - - - 88
Interactive Debugger - - - - - 88
 Setting Breakpoints - - - - - 89
 Examine Dataspace Window - - - - - 89
 Sample Debug Session - - - - - 89

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.

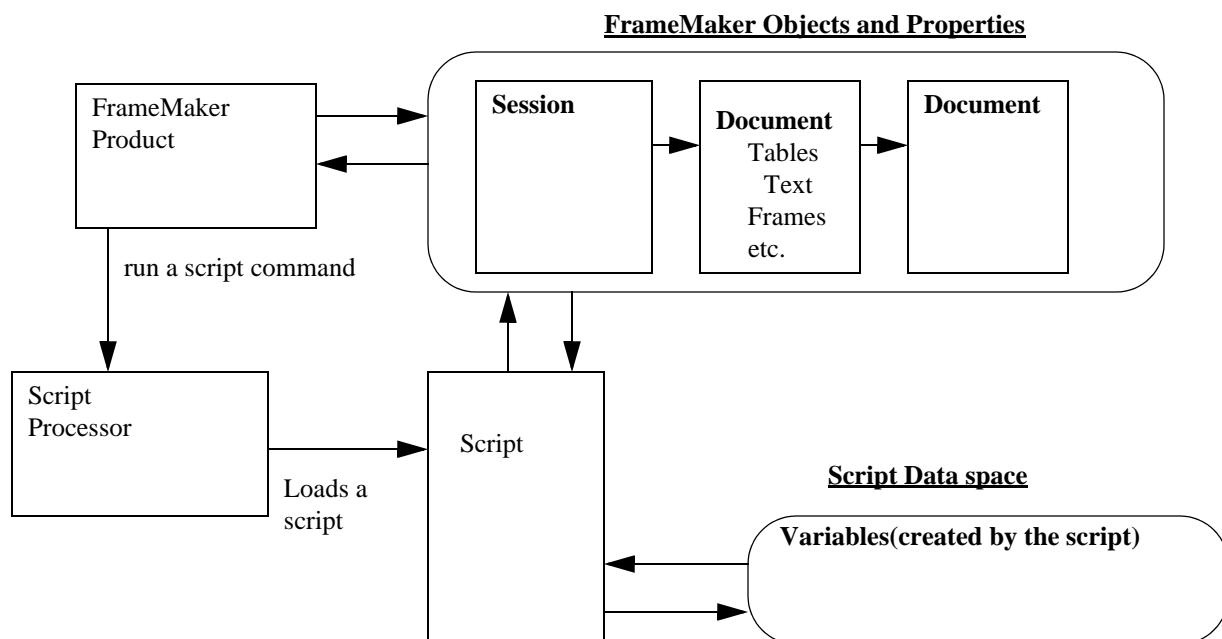
What is a script?

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

Installing FrameScript

FrameScript is delivered as a single installation file, which, when run, installs the entire product into a directory of your choosing. The form of the installation file name is **Fs14_XX.exe**, where **XX** is the target FrameMaker version. So the FrameScript install file for FrameMaker 7.0 would be **Fs14_70.exe**.

IMPORTANT: You should not install one version of FrameScript over an older version. You should first uninstall the old version. You can, however, keep the old version around if you install the new version into a different directory. Only one can be active at a time for a particular FrameMaker installation.

Starting the installation

IMPORTANT: Make sure that FrameMaker is **NOT** running when you install FrameScript. If it is running, then the registration step will not work.

IMPORTANT: If you are installing an evaluation copy of FrameScript, ignore any references to the registration number. It will not be required.

You start the installation by double-clicking on the installation file. The InstallShield process takes over from there. It will let you choose the directory to install the product. The default directory is

C:\Program Files\ElmSoft\FrameScript4_XX

where **XX** is the target version of FrameMaker. Follow the instructions on the ensuing screens to create the correct directory structure and copy the files into the right places. When this is complete, a registration screen will appear. This screen will allow you to enter your FrameScript registration number. It will verify it and store it in the correct place. It will skip this step for evaluation copies. Since FrameScript is an add-on to FrameMaker, FrameScript must be registered with FrameMaker as well. This registration screen will also locate the correct version of FrameMaker and make the necessary entry in FrameMaker's add-on list. If you have more than one FrameMaker installation, you may have to choose to which version of FrameMaker you wish to register this version of FrameScript.

Summary of Steps

1. Double click on the installation file.
The install file does not have to be in any special directory. You can keep it anywhere you keep your downloaded files.
2. Choose the directory to place the files.
The default directory will be the correct one for most installations.

3. When the registration screen appears, enter your FrameScript registration number, user name, and optionally you company name.
4. Make sure the correct version of FrameMaker is identified. If not, choose the correct version.

IMPORTANT: The registration program looks at the Windows Registry to find the correct location for the FrameMaker version corresponding to the FrameScript version you are installing. Most of the time, it will find the correct version. In some cases, however, especially if you have more than one copy of the same version of FrameMaker installed, the registration program may not be able to find it. If this happens, use the Browse button on the registration dialog, to select the correct version of FrameMaker. Select the FrameMaker executable file (**FrameMaker.exe** or **FrameMaker+SGML.exe**, depending on the version you have).

5. Press Register

Completion

When the above steps finish FrameScript should be ready to run. The next time you start FrameMaker you should see the FrameScript menu in the FrameMaker menubar. If this does not happen, look at the installation trouble shooting section.

Directory Structure

This directory structure should look like the following illustration.

MainDirectory

```

SysScripts-- Sub-Directory, Contains various system scripts
Docs      -- Sub-Directory, Contains documentation files (PDF)
Lib       -- Sub-Directory, Contains library scripts
Demos     -- Sub-Directory, Contains demonstration scripts and docs
SampleScriptsSub-Directory, Contains the sample scripts
Tutorial  -- Sub-Directory, Contains a tutorial
fsl4_70.dll File, The FrameScript Api client
WinSys.dll File, An auxillary client file
EslDlgs.dll File, Dialog Resources
fscript.ini File, The customization file
RegisterEsl.exeFile, The registration program
EslReg.ini File, contains the registration information
RelNotes.pdf File, Release Notes for this release.
Plus several .Property files.

```

Installation troubleshooting

If the FrameScript menu does not appear when you start FrameMaker, first check to make sure that the above directory structure is in place. If not, then the installation failed for some reason. You may have to try to uninstall it before trying to install it again. See the uninstall instructions.

If the product installed correctly (the above directory structure is in place), but the FrameScript menu does not appear on the FrameMaker menu bar or if you get the message that you have an invalid registration number, then the registration step may have failed. You can try running this part again by running the RegisterEsl.exe program from the

main directory. This will bring up the registration screen again. Check to make sure all the information is correct including the FrameScript registration number.

If the registration screen does not work or cannot find the correct version of FrameMaker, you can try to register the product manually. See below.

If every attempt fails and you need to contact Tech support, make sure you provide the following information:

- FrameMaker registration number
- FrameMaker version
- FrameScript registration number or evaluation
- Version of MS Windows
- A list of steps the you followed above and where in the installation it may have failed

Manual registration

IMPORTANT: Make sure that FrameMaker is **NOT** running when you manually install FrameScript. If it is running, then the registration step will not work.

To register FrameScript manually you will need to use a text editor, such as notepad.exe, located in the accessories folder (Start->Programs->Accessories->NotePad). Open the fscript.ini file located in your FrameScript directory. Under the [RegInfo] section enter you user name (and optionally company name) and your FrameScript registration number (RegNum), as follows:

```
[RegInfo]
User=My Name
Company=My Company
RegNum=04-4-5X-NNNN-AAAA-N
```

Make sure you replace the above entries with your information, especially your FrameScript registration on the RegNum line.

Save this file when you are finished. This will record your FrameScript registration information. The next step is to make FrameMaker aware of the FrameScript client.

To add FrameScript to the list of FrameMaker clients, do the following:

Use notepade again and open the file maker.ini, located in your FrameMaker directory. This file might also be fmsgml.ini if you are using the FrameMaker+SGML versions of Frame 6.0 or 5.5.6.

Locate the section called [APIClients]. Go to the end of this section and add the following line for FrameMaker versions 6.0 and below:

```
fsl=Standard,FrameScript,YourFrameScriptDirectory\Fsl4_XX.dll
```

For FrameMaker versions 7.0 or greater, add the following line:

```
fsl=Standard,FrameScript,YourFrameScriptDirectory\Fsl4_XX.dll,all
```

the last part is the name of the FrameScript client name found in the installation directory. Replace the YourFrameScriptDirectory with your actual FrameScript installation directory.

Save the file and re-start FrameMaker.

Uninstalling FrameScript

You uninstall FrameScript by using the MS Windows Add/Remove programs icon located in the MS Windows Control Panel. Bring up the control panel and double click on this icon. It will provide a list of installed software. Select the line identifying FrameScript and click uninstall.

Chapter 3

Using FrameScript

When FrameScript is installed, it will place a new menu on the FrameMaker menubar called FrameScript (though the actual text may be different due to customization options). This menu will contain six commands (and optionally a sub menu).

Using FrameScript primarily involves running and installing scripts. These scripts may have been written by you or provided by others. A script is usually a text file (commonly using the extension fsl) containing a set of FrameScript commands. In some cases, it can also be an object file (FrameScript style object file), commonly using the extension fso. These object files are generated using the Compile command. This is usually done by the script writer and is not of interest to someone just using scripts.

There are two kinds of scripts, standard scripts and event scripts. These are discussed more thoroughly in the Basics.pdf document. Standard scripts are designed to run and then return to the user. Event scripts are installed, which means they are loaded in the system and they wait for any of the system events for which they were programmed to wait. These events can be menu command events, messages or system notifications. The script developer (scripter) decides what type of script it will be when it is designed. As a script user, you will need to know whether a script is a standard script or an event script before you can use it, because the menu commands will work differently for each type.

Running a script is simple. You just need to select the **Run** menu command, from the FrameScript menu, then choose the script file (containing a standard script) from the resulting dialog. This dialog allows you to navigate to any place on your hard disk (or network disk).

Installing scripts can be more complicated because there are two kinds of scripts that may be installed. Installing standard scripts is just a way of making it easier to access them by automatically creating a menu command which allows the user to run that script. When event scripts are installed, however, they are loaded into the system where they wait for their events to occur.

Compiling scripts is something that script writers do. This takes a script in a text file and converts it into an object file.

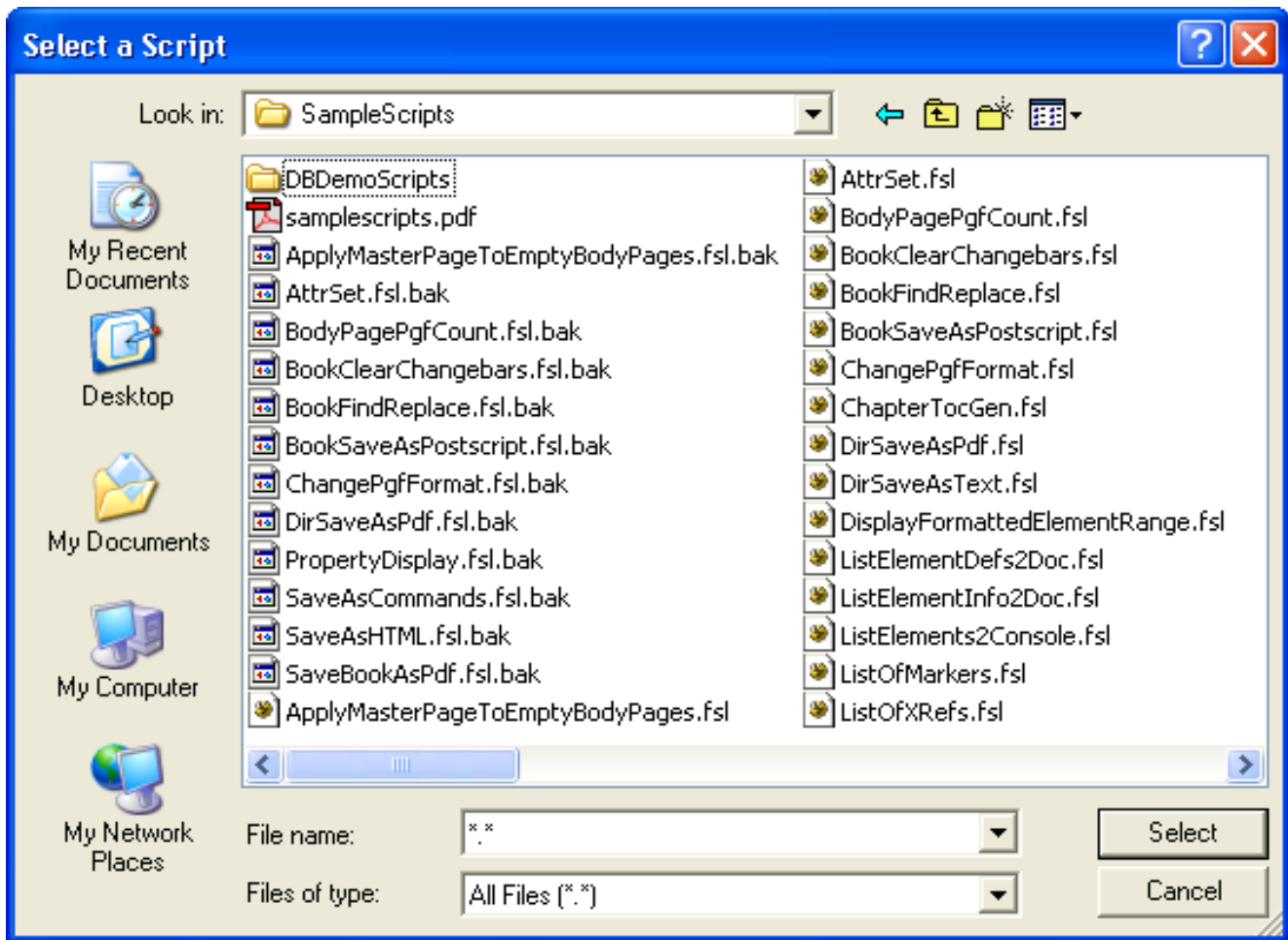
IMPORTANT: If you wish information on writing scripts, see the section on Writing Scripts.

Running Scripts

Run command

You use the Run command to start a standard script. When you select this command a dialog box appears asking you to select the script you wish. See “Select a Script to Run” on page 10. If you select an event script instead of a standard script, nothing will happen.

Figure 3-1 Select a Script to Run



When you choose your script it runs immediately.

IMPORTANT: If you wish to stop a running script after it has started, press the ESC key. This causes the script to come to a halt immediately after the next threshold command finishes. See “UserAbortThreshold” on page 20

Installing Scripts

Install Menu command

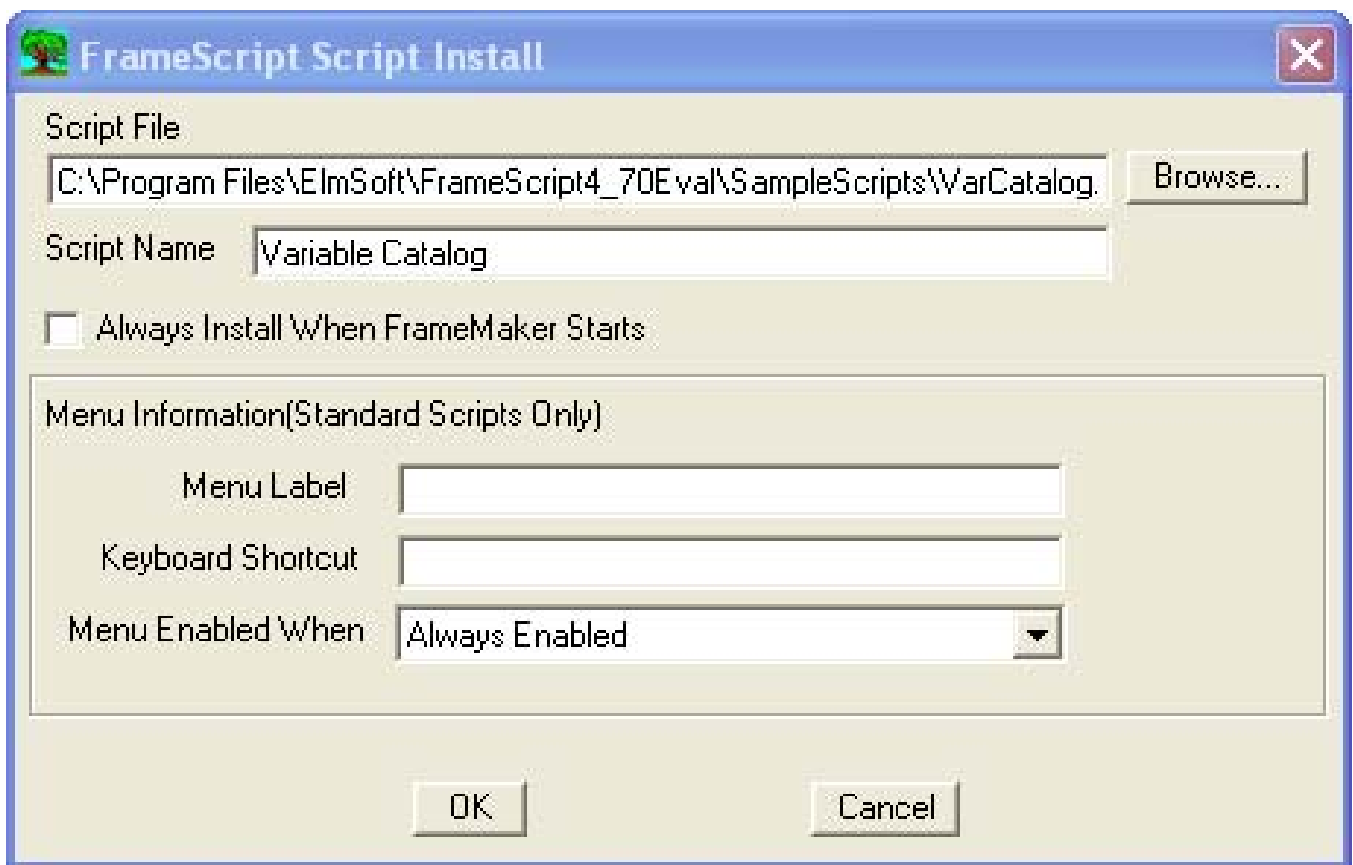
The Install menu command tells FrameScript to install a script into the FrameScript system. A standard script will be assigned a menu item for easy access. An event script will be initialized so it can handle any events it has defined. When you select this menu command, FrameScript will present the FrameScript script install dialog box (See Figure 3-3, “Install Standard Script,” on page 12). You can enter the file name of the script you wish to install (or more conveniently use the browse button to select a script file). Then enter a name for the script. This name is used to allow

you identify the script later if you wish to uninstall it. If you do not specify a name, FrameScript will make up a name (StandardScriptName n), which is not very helpful.

If you are installing an event script all you have to do is press the OK button and the script will be installed. The figure below shows the install event script dialog.

IMPORTANT: A script is installed only for the current FrameMaker session. To have a script installed each time FrameMaker starts, use the Install script command in the Initial Script. See “InitScript=c:\FrameScript\myinit.fsl” on page 17

Figure 3-2 Install Event Script

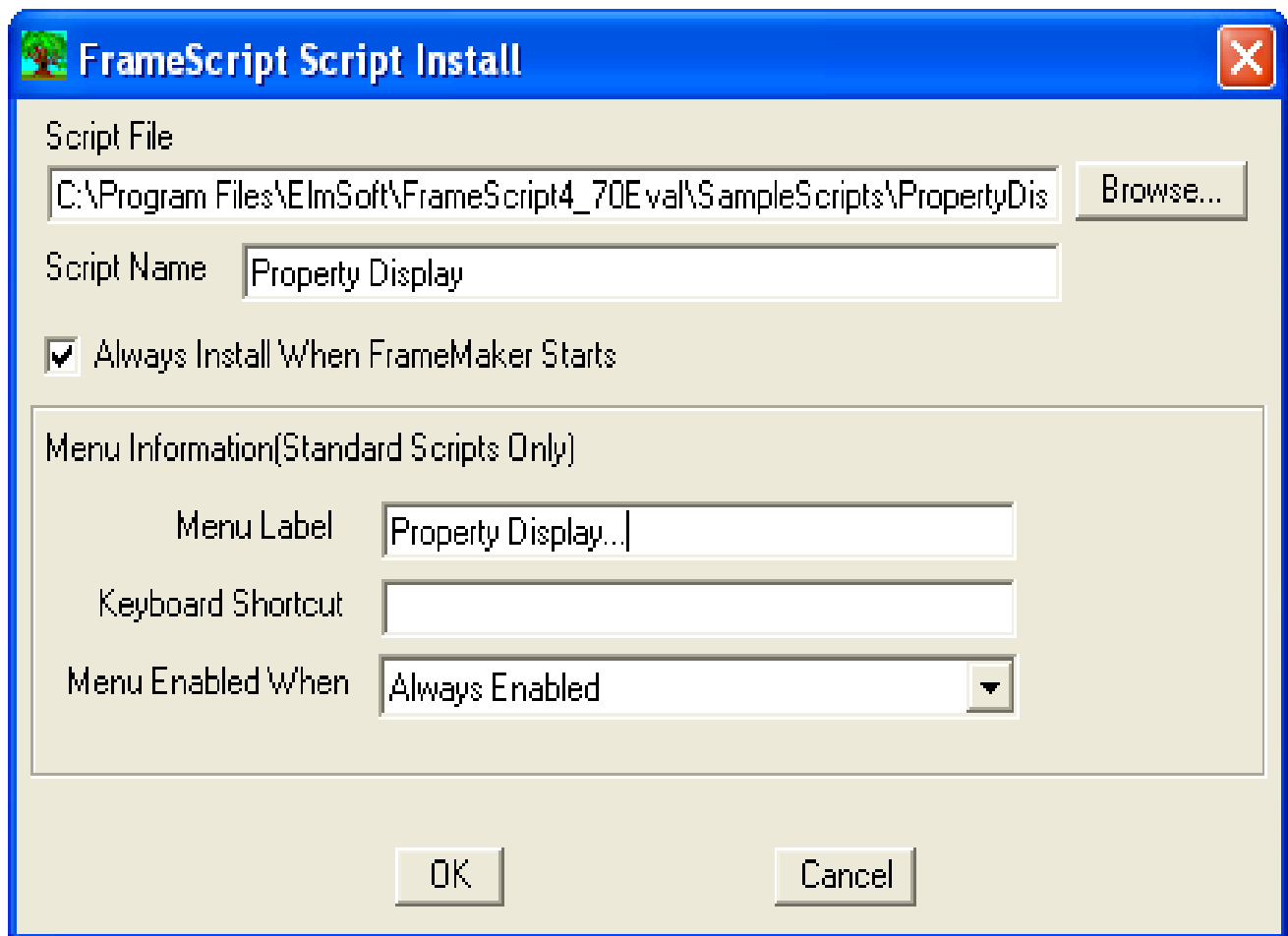


If you are installing a standard script, you should supply more information. Since installing a standard script creates a menu command for easy access, you should enter the text of that menu command. If this is left blank a default name will be chosen. You can optionally supply a keyboard shortcut and you can also specify when the menu command will be activated. The following table shows the text to use for each type of keyboard shortcut.

Keyboard key combination	Text for the keyboard shortcut field
Alt-Fn (where Fn is any function key)	~/Fn
Ctrl-n (where n is any letter)	^n
Ctrl-Fn (where Fn is any function key)	^/Fn
Shift-Fn (where Fn is any function key)	/Fn

The figure below shows the install standard script dialog.

Figure 3-3 *Install Standard Script*

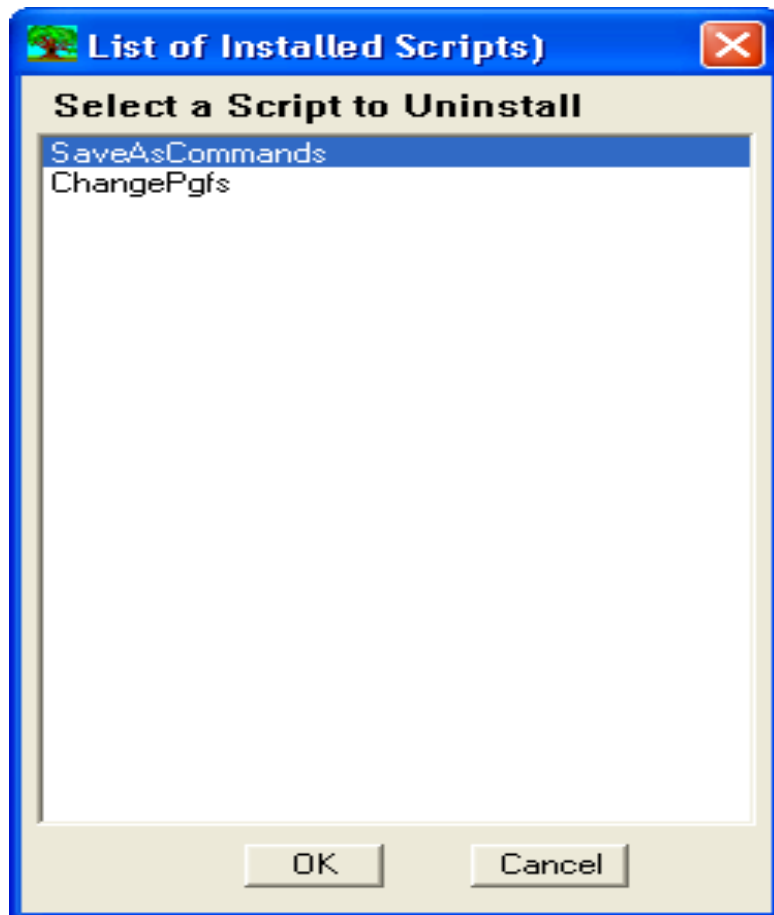


Uninstalling Scripts

Uninstall Script Menu Command

The Uninstall script command removes a previously installed script from the script space, making it unavailable to the user. When the user selects this command, a dialog box will appear with a list of all the installed scripts, see Figure 3-4. Select the one you wish, then press the OK button. The selected script will be removed.

Figure 3-4 Uninstall Script (Windows)



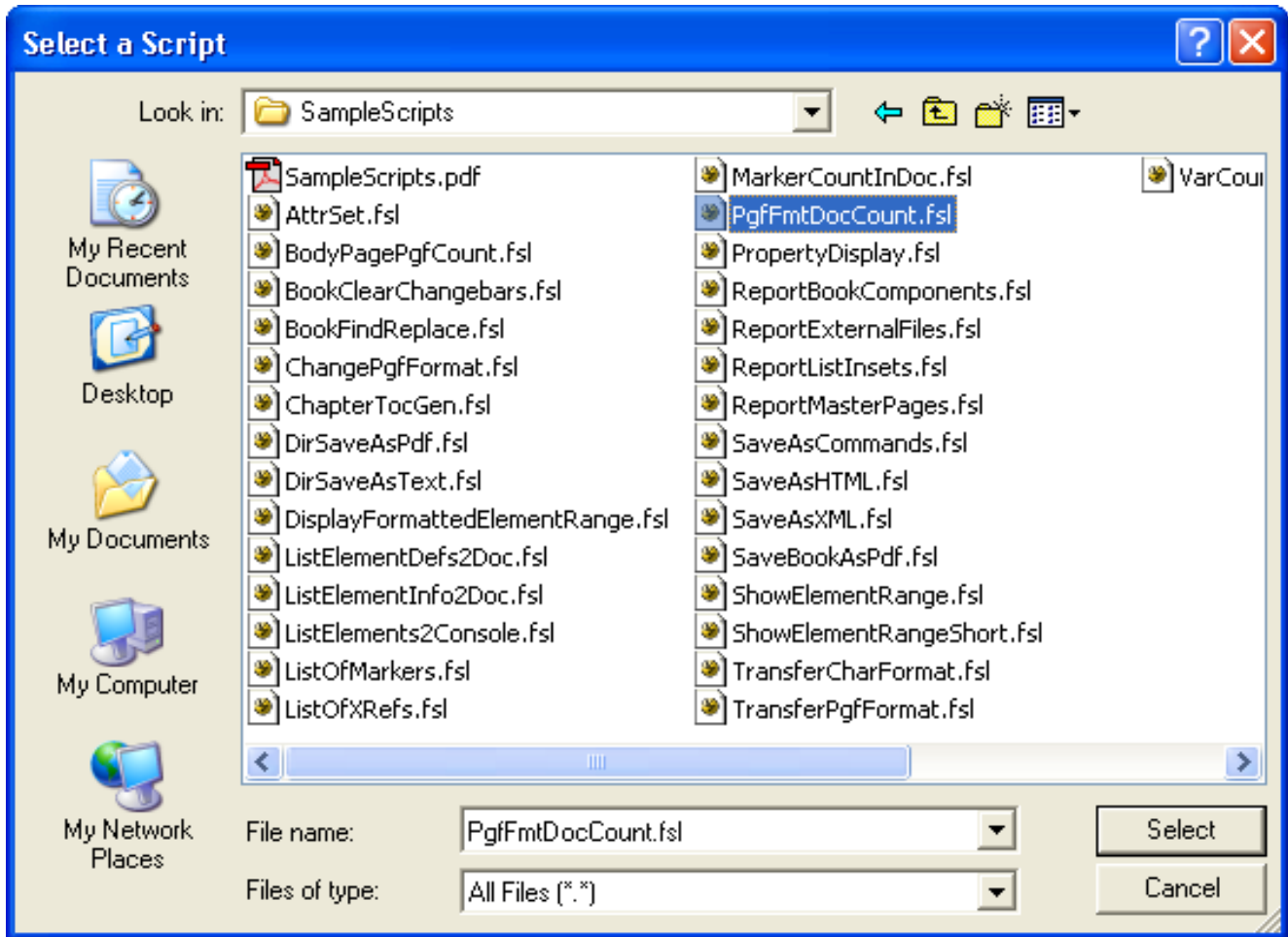
Compiling Scripts

The compile menu command allows the script writer to convert a script from the text file format to the FrameScript object format. The primary reason for doing this is to distribute a script without giving the user the ability to modify it (or even look at it). This is useful in organizations where a small number of script writers write scripts for others to use and they do not want the user modifying them. It is also useful for script developers who sell scripts to others.

Compile Menu command

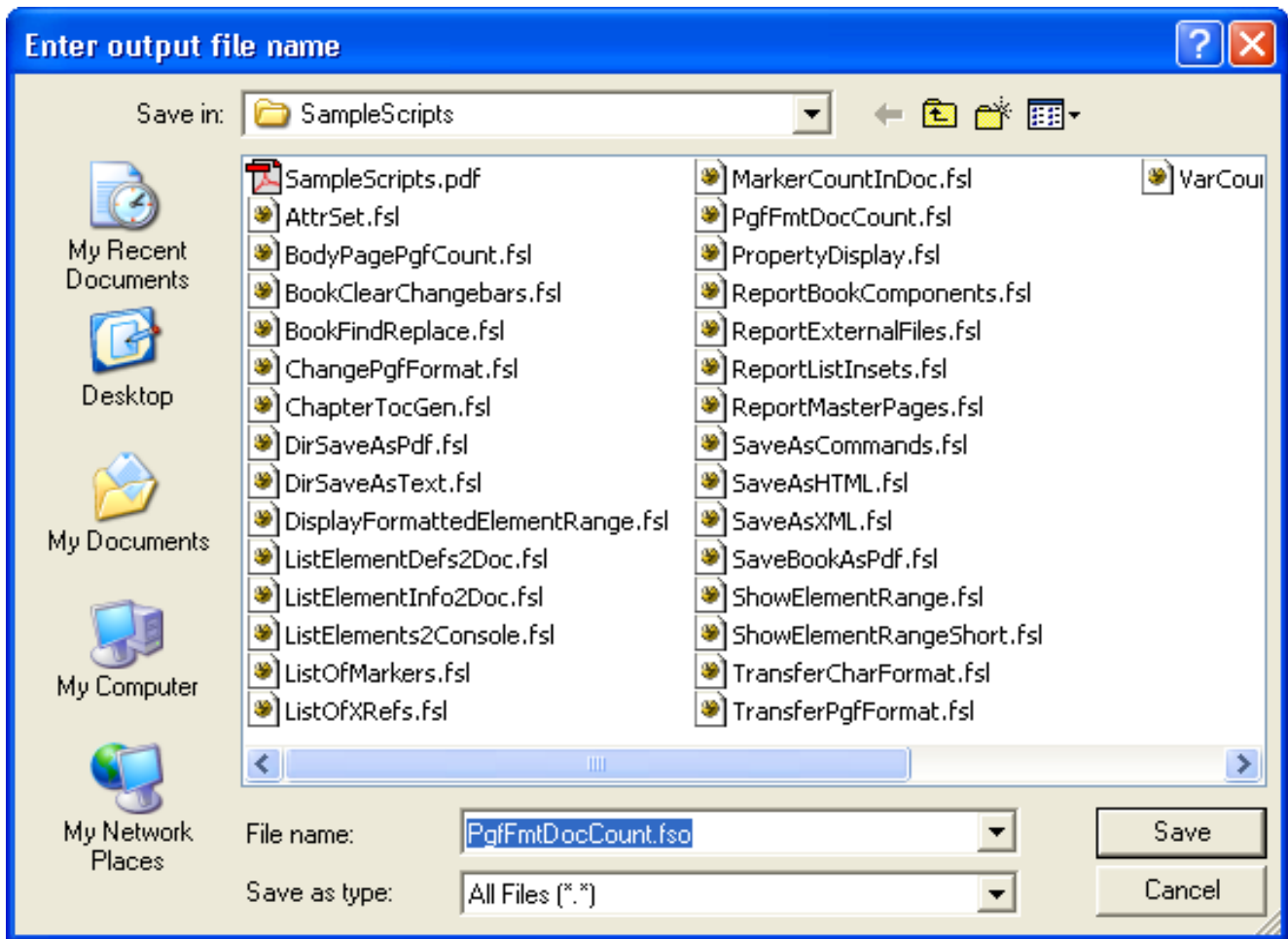
When you select the FrameScript->Compile command a dialog box appears asking you to select the script to compile. This screen is very similar to the one used to select a script to run (See “Select a Script to Compile” on page 14).

Figure 3-5 Select a Script to Compile



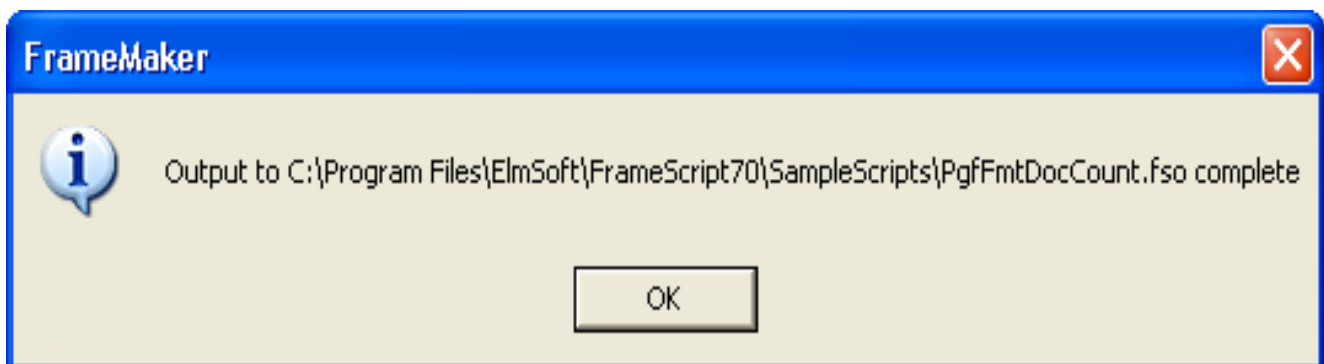
When you select a script, however, it does not run it but instead it puts up a second dialog asking for the location to put the compiled script (See “Select the output file for the compiled script” on page 15). It starts in the same directory as the source script and places a default name using the name of the source script with a fso extension. Most of the time, you will want to keep it that way. If you don’t, then you can change the name (and/or directory) to suit your needs.

Figure 3-6 Select the output file for the compiled script



A confirmation message appears when the process is complete.

Figure 3-7 Confirmation of the compiled script function



Customizing FrameScript

You can customize the way that the user interface looks, how FrameScript responds to errors, how it searches for scripts and other items. You do this with the Options menu item

Options Menu Command

You can modify the customization options by using the Options menu command. This command brings up a dialog box which has three panels selected via a drop down box. The General panel allows you to select the initial script, log file and error handling. The Search path panel defines how FrameScript searches for scripts and the Menus panel lets you change the names of the menu items and it also lets select which menu items (if any) will appear.

Use the drop down box to change to a different panel.

General Panel

The initial script field allows you to select a script to run when FrameMaker (and FrameScript) starts. The Keep initial data checkbox says to keep the global data space available when the initial script terminates. This allows you to define read-only variables for all other scripts to use. You would want to turn this off if these variables might interfere with variables defined in other scripts.

If the Signon screen checkbox is on, then a small window will appear whenever FrameScript starts. Uncheck this to turn off that behavior.

The following figure shows the general panel.

Figure 3-8 Options Dialog (General Panel)

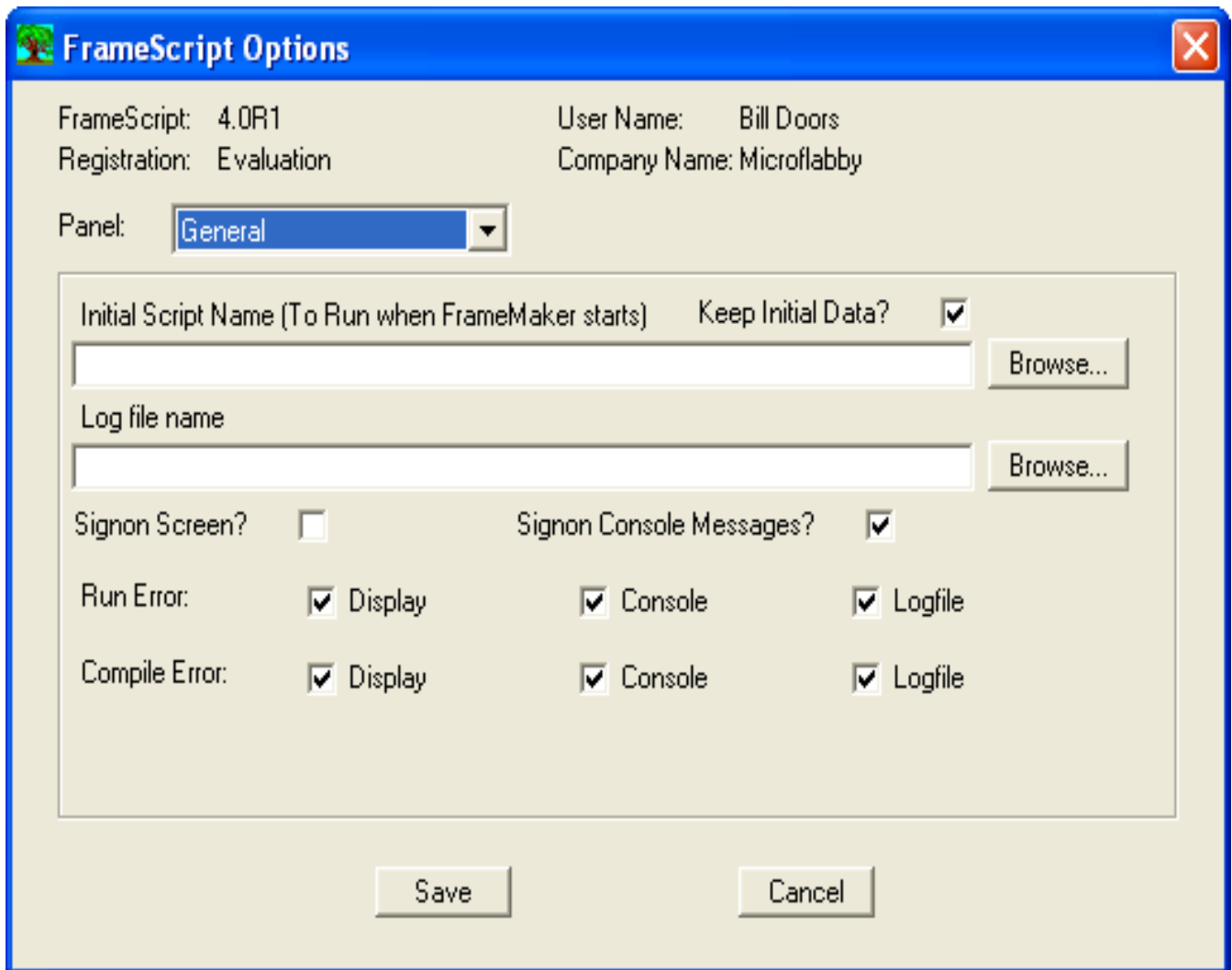
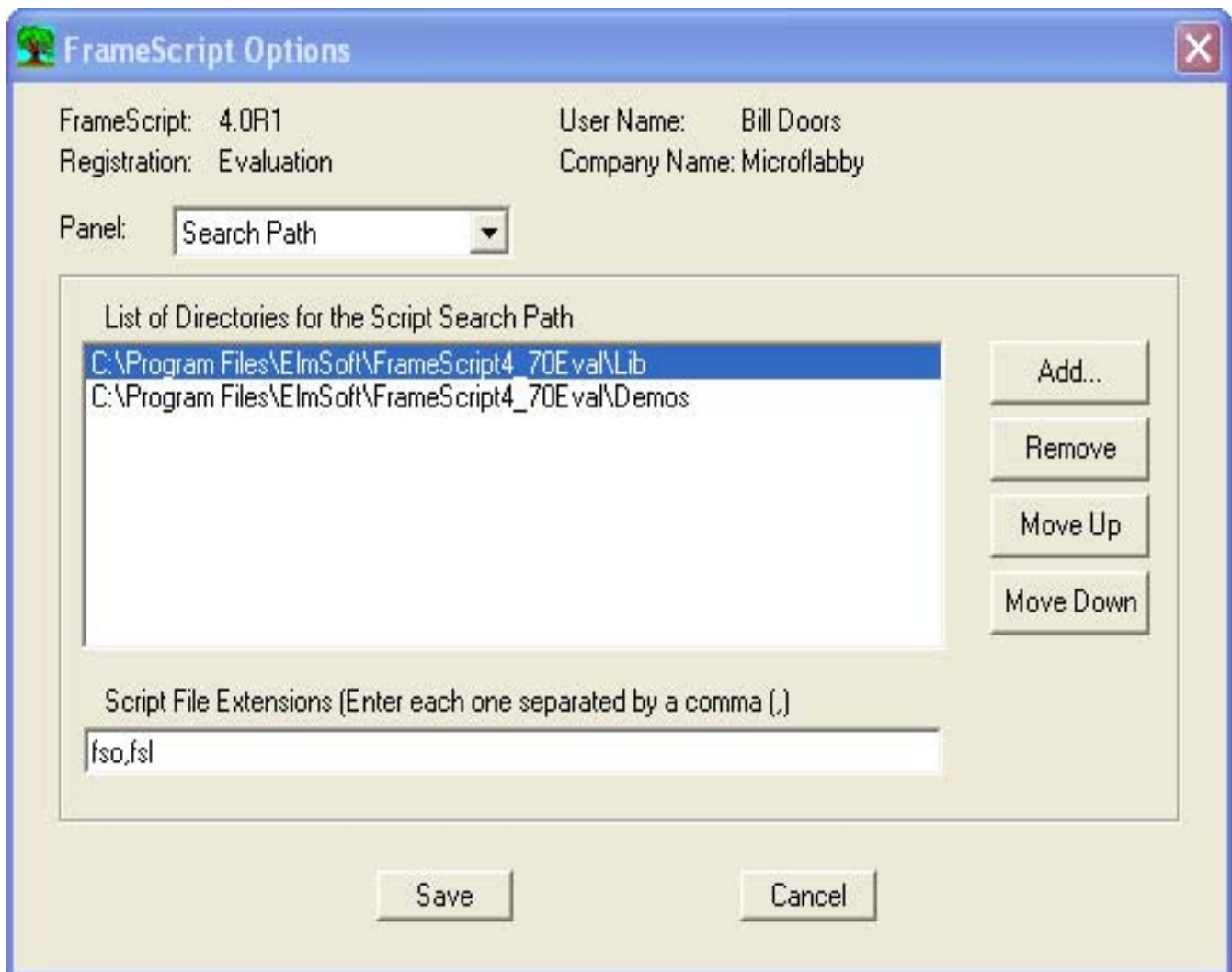


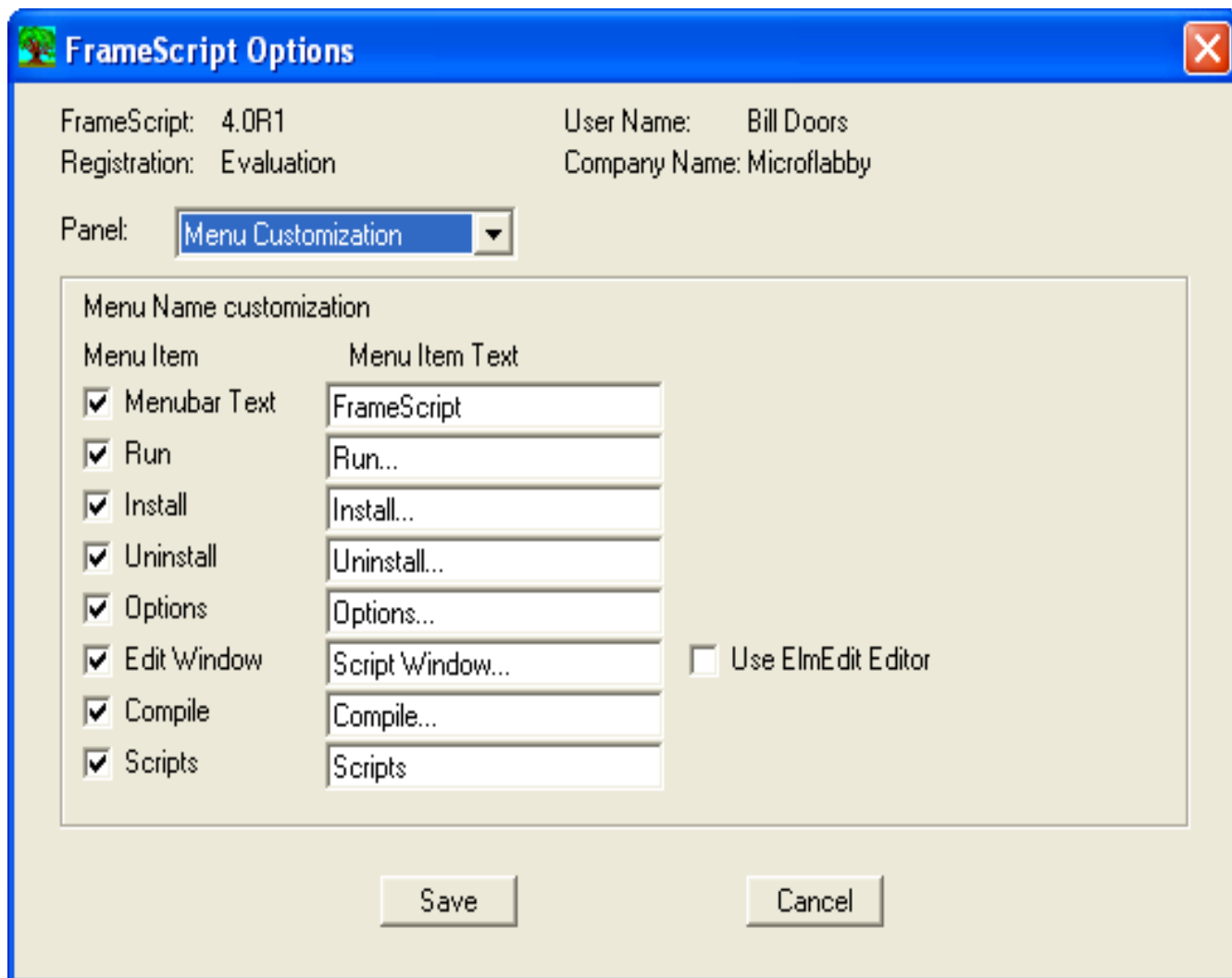
Figure 3-9 Options Dialog (Search Path Panel)



Search Path Panel

If a FrameScript command tries to access another script (the install script command for example), and a full pathname is not specified, FrameScript will search the directories in the search path for that file. If the file extension is not given, then it will use the set of file extensions specified here. Use the buttons to add new entries or change the order of the directories.

Figure 3-10 Options Dialog (Menus Panel)



Menus Panel

This panel lets you customize the user interface. A check in the checkbox indicates that the corresponding item will be available to the user. You may also change the text of the menu item. If you turn off the Menubar text, then the FrameScript menu will not appear at all.

Customization using the Ini File

FrameScript provides a customization file which allows you to set various options. Ordinarily, you will use the Options dialog box to modify these items, but you can use a text editor to modify them manually. Make sure the FrameMaker is not running when you do this. The name of the customization file is `fscript.ini`. It must be located in the same directory as the FrameScript client program (`fs14_xx.dll`, where `xx` is the target FrameMaker version).

The following table shows the section names, options, a description and the default values that will be used if it is not specified in this file.

Table 1: FScript.ini File customization options

Section	Option	Description
[GENERAL]	SignonScreen	This option specifies whether you wish to have the FrameScript signon window appear when you start a Frame session. e.g. SignonScreen=Yes If not specified, the signon screen <i>will</i> appear. A value of No means it will not appear.
	SignonMessage	This option specifies whether you wish to have the FrameScript signon console messages appear when FrameMaker starts. e.g. SignonMessage=Yes If not specified, the signon console messages <i>will</i> appear. A value of No means it will not appear.
	GlobalDataSpaceForInitialScript	This option specifies whether you wish to have the global data space from the initial script (if any) saved as read-only global variables. e.g. GlobalDataSpaceForInitialScript=Yes If not specified, the global variables <i>will</i> be saved. A value of No means they will not be saved.
[FILES]	LogFile	This option allows you to specify the name and location of the FrameScript log file. This log file is where FrameScript writes comments and error messages. e.g. LogFile=c:\fscript.log If not specified then no logfile will be generated.
	InitScript	This option allows you to specify the name of a script to run at the start of a FrameScript session. You must specify the complete pathname of the initial script to run. e.g. InitScript=c:\FrameScript\myinit.fsl If not specified, then no script is run when FrameScript starts.
[Defaults]	ScriptExtensions	This option specifies the default file extensions that FrameScript will search for when not specified. e.g. ScriptExtensions=fso,fsl
	UserAbortThreshold	This option specifies how often the user abort is checked. A higher number improves performance. e.g. UserAbortThreshold=50 The default value is usually adequate.
[ScriptSearchPath]	n (A sequence number)	This option specifies a set of directories that FrameScript uses to search when a complete path is not specified. e.g. 1=c:\Program Files\ElmSoft\FrameScript4\Lib 2=c:\Program Files\ElmSoft\FrameScript4\SampleScripts

Table 1: FScript.ini File customization options

Section	Option	Description
[Directories]	SearchScript	This option specifies the initial directory to look for for scripts when the user selects the Run... menu command. e.g. SearchScript=c:\FrameScript\SampleScripts If not specified, then FrameScript will start looking in the same directory as the FrameScript client program.
[MENUS]	RunMenuItem	This option specifies whether you wish to have the Run... menu item on the FrameScript menu or not. e.g. RunMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	InstallMenuItem	This option specifies whether you wish to have the Install... menu item on the FrameScript menu or not. For Example: InstallMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	UninstallMenuItem	This option specifies whether you wish to have the Uninstall... menu item on the FrameScript menu or not. For Example: UninstallMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	OptionsMenuItem	This option specifies whether you wish to have the Options... menu item on the FrameScript menu or not. For Example: OptionsMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	ScriptWindowMenuItem	This option specifies whether you wish to have the Script Window... menu item on the FrameScript menu or not. For Example: ScriptWindowMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	CompileMenuItem	This option specifies whether you wish to have the Compile... menu item on the FrameScript menu or not. For Example: CompileMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	ScriptsMenuItem	This option specifies whether you wish to have the Scripts sub menu on the FrameScript menu or not. For Example: ScriptsMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	MainMenuItem	This option specifies whether you wish to have the FrameScript menu appearing or not. If this is not on, then the other menus items will not appear For Example: MainMenuItem=Yes If not specified, the menu item <i>will</i> appear. A value of No means it will not appear.
	UseElmEdit	This option specifies whether you wish to use the ElmEdit editor or the standard script editor. For Example: UseElmEdit=Yes No is the default value, which means that the standard editor will be used..

Table 1: FScript.ini File customization options

Section	Option	Description
[MENUNAMES]	RunMenuName	This option specifies the label of the Run... menu item. Default: RunMenuName=Run...
	InstallMenuName	This option specifies the label of the Install menu item. Default: InstallMenuName=Install...
	UninstallMenuName	This option the label of the Uninstall... menu item. Default: UninstallMenuName=Uninstall...
	OptionsMenuName	This option specifies the label of the Options... menu item. Default: OptionsMenuName=Options...
	ScriptWindowMenuName	This option specifies the label of the Script Window... menu item. Default: ScriptWindowMenuName=Script Window...
	CompileMenuName	This option specifies the label of the Compile... menu item. Default: CompileMenuName=Compile...
	ScriptsMenuName	This option specifies the label of the Scripts sub menu. Default: ScriptsMenuName=Scripts
	MainMenuName	This option specifies the name of the FrameScript menu. Default: MainMenuName=FrameScript
[ErrorHandling]	RunErrorDisplay	This option tells if FrameScript will report a FrameScript command run error to the display (dialog box). Default: RunErrorDisplay=Yes If not specified, no reporting is done.
	RunErrorConsole	This option tells if FrameScript will report a FrameScript command run error to the console. e.g. RunErrorConsole=Yes If not specified, no reporting is done.
	RunErrorLogFile	This option tells if FrameScript will report a FrameScript command run error to the logfile. e.g. RunErrorLogFile=Yes If not specified, no reporting is done.
	CompileErrorDisplay	This option tells if FrameScript will report a FrameScript command option error to the display (dialog box). e.g. CompileErrorDisplay=Yes If not specified, no reporting is done.

Table 1: FScript.ini File customization options

Section	Option	Description
	CompileErrorConsole	This option tells if FrameScript will report a FrameScript command option error to the console. e.g. CompileErrorConsole=Yes If not specified, no reporting is done.
	CompileErrorLogFile	This option tells if FrameScript will report a FrameScript command option error to the logfile. e.g. CompileErrorLogFile=Yes If not specified, no reporting is done.

Using the Script Window

Script Window

The Script window is now part of the ElmStudio system. For complete documentation, see Chapter 6, “Using ElmStudio.”

Using other editors

The script editor is provided as a convenience for script writers. But you do not have to use this editor. Since script files are standard text files, any standard text editor will work for creating and modifying scripts. There are many of these available, including the notepad.exe program that comes with MS Windows. Many customers have used UltraEdit from IDM Computer Solutions (www.ultraedit.com) and TextPad from Helio Software Solutions (www.textpad.com). These two text editors have programmable tools that allow direct connection to FrameScript via the RunEslBatch.exe program, described in the next chapter. Other editors may also have this feature.

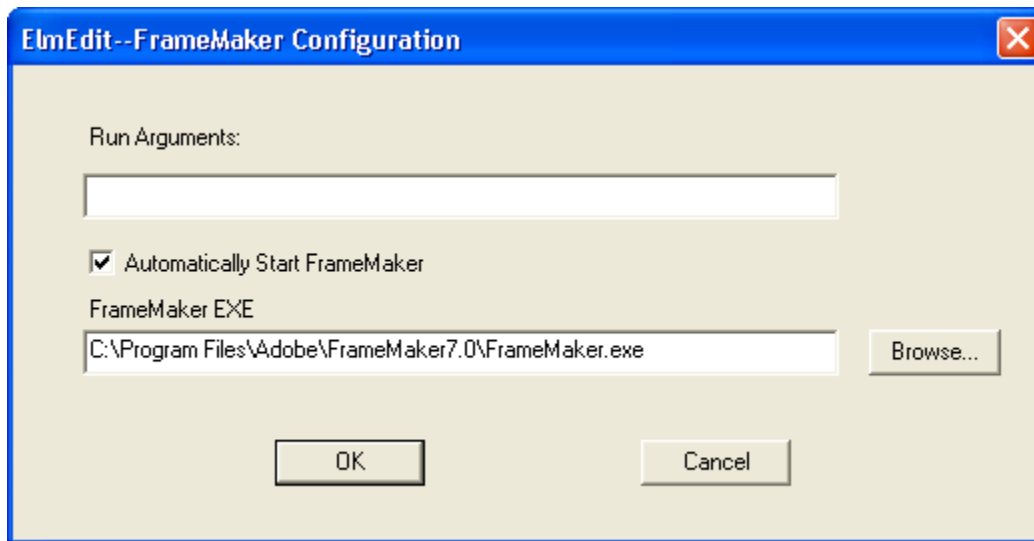
ElmEdit

ElmEdit.exe is a stand-alone, multi-file text editor that you may use to create and modify script files. This editor does not have all the bells and whistles of some of commercial text editors but it has an adequate set of features for standard text editing. Unlike the script window above, this editor is used outside of FrameMaker. You can save your script files and run them using the Run menu command. You can also run scripts directly from ElmEdit. The ElmEdit Run Menu command (or Run button) will send the contents of the current script to the FrameMaker program, where it will be run. If FrameMaker is not currently running, ElmEdit will attempt to start FrameMaker, unless configured not to do so.

Configuring ElmEdit for FrameMaker

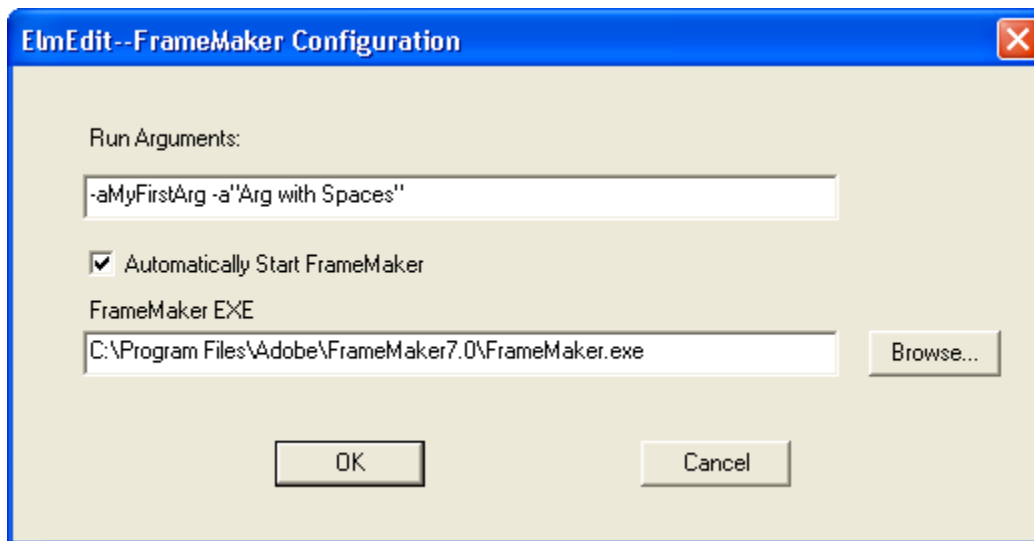
ElmEdit comes preconfigured to connect to FrameMaker without any configuration necessary. If FrameMaker is already running, it will receive the script. If FrameMaker is not running, ElmEdit will use the registry information to attempt to start FrameMaker. If there is some problem, or if you have more than one version of FrameMaker installed, then you may need to configure ElmEdit to launch the correct version of FrameMaker.

The Config->FrameMaker menu command will bring up the configuration window, as follows:

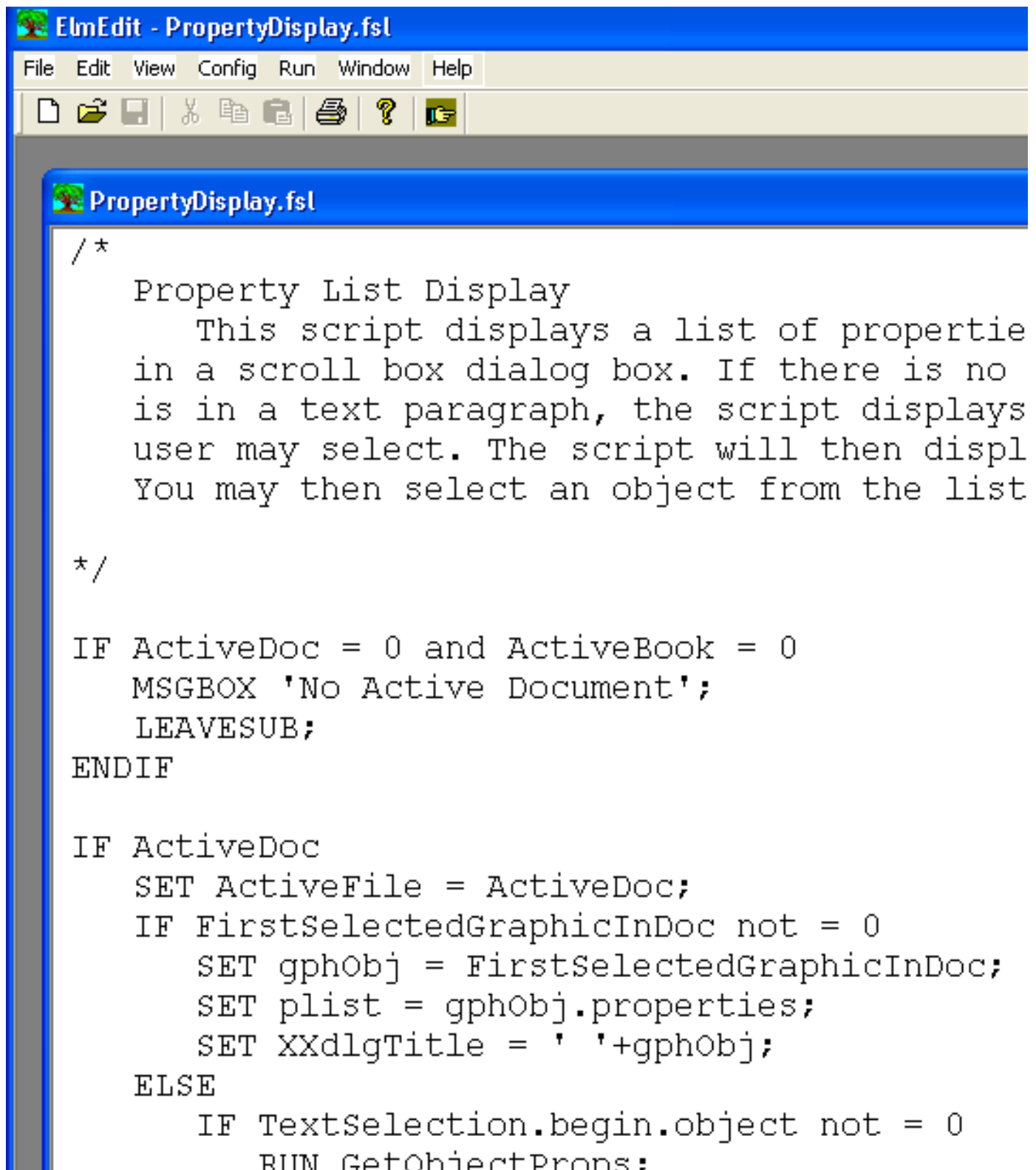


Use the Browse button to select the correct copy of FrameMaker. You can also elect not to start FrameMaker automatically, by unchecking the check box. If this is done, then ElmEdit will only run the script, if FrameMaker is already running.

The Run Arguments edit box is used to pass arguments to the script that you run. This is useful if you are testing a script that will run in batch mode (See Chapter 5, “Batch Processing.”). below is an example of setting script arguments.



Two arguments will be passed to the script will you run it from ElmEdit. These arguments are accessed by the **Args** array from the main script.

Figure 3-11 *ElmEdit*

Batch Processing

RunEslBatch

RunEslBatch.exe is a batch oriented windows program that sends messages to FrameScript telling it to run scripts. RunEslBatch is also capable (if properly configured) to start FrameMaker if it is not already running, before attempting to send the message.

Using RunEslBatch

The format for running RunEslBatch is as follows:

```
RunEslBatch -f"ScriptFilename" [-aValue] ... [-aValue]
or
RunEslBatch -s"ScriptText" [-aValue] ... [-aValue]
```

This is how it might appear in a DOS batch file.

Using the -f option, the ScriptFilename is the full path name of the file containing the script that you wish to run or a partial name of a script that is in the SearchPath. Using the -s option, the ScriptText is the actual text of a script to run. The -a option(s) are used to pass string values to a script. Use double quotes if the values contains spaces or special characters. These arguments are passed to the script like subroutine/function arguments. You can access them in a script using the Args array. Args.Count will give you the number of arguments passed. Args[1] will be the first argument, Args[2] the second, and so on. Also the name of each argument is of the form ArgN, where N is the order number (starting at 1) of the arguments.

For example, you could have RunEslBatch send the included script as follows:

```
RunEslBatch -s" Quit Session;"
```

This single line script will cause FrameMaker to quit running.

IMPORTANT: Due to limitations with the DOS/Windows command line processing, the -s option is useful only for a short scripts without special characters.

Here is a sample batch file using RunEslBatch. You could also include other programs in the batch file.

```
RunEslBatch -f"MyScript1.fsl" -a"My String Value" -a"My other string value"
RunEslBatch -f"c:\MyScripts\MyOtherScript.fsl"
RunEslBatch -s" Quit Session;"
```

IMPORTANT: When running in batch mode, be sure that the scripts do not require any user interaction, because the script will wait for it. If you wish to run a long operation over night, this could be a problem.

Configuring RunEslBatch

RunEslBatch will run without being configured if FrameMaker is already running when the program is started. If FrameMaker is not running, RunEslBatch will check the registry for the current version of FrameMaker. It will automatically start this version of FrameMaker. If you have more than one version of FrameMaker installed and RunEslBatch starts the wrong version, you can configure it manually using the configuration file. The configuration file is called EslBatch.ini and it is located in the Windows directory (WinNT for Windows 2000 systems). It has the following form:

```
[FM]
ExeFile=C:\Program Files\Adobe\Framemaker7.0\Framemaker.exe
```

The ExeFile keyword identifies the location of the FrameMaker exe file. The above example shows the usual location for a FrameMaker 7.0 installation. You will need to replace this with the complete path of your own FrameMaker installation. You will need a text editor to do this.

The RunEslBatch.exe and EslBatch.ini files are installed into the Windows directory by default. They can be located elsewhere as long as they are together.

IMPORTANT: If the RunEslBatch program is not in a directory which is part of the PATH, then you will need to include a full path to run it.

Configuring RunEslBatch for Text Editors

To create and modify scripts, you can use the built-in Script Window editor, but, since scripts are standard text files, you can also use any other standard text editor to perform the same function. Notepad.exe comes as part of MS Windows, so it is always available. It is a simple, single file at a time text editor. You can use this to write and modify scripts, then test them using the FrameScript Run command. Another text editor, WordPad.exe, also comes with MS Windows. It is a more advanced editor, but you have to remember to save the files as Text-Only files. The default is to save as RTF. RTF files will not work with FrameScript. There are also 3rd party solutions, which can be purchased separately from their respective software vendors. They are generally inexpensive and have evaluation versions available.

Our customers have recommended two 3rd party Text Editors, UltraEdit and TextPad. These editors are multifile text editors that have many compelling features, including syntax highlighting. Syntax highlighting lets you configure how the text file looks on the screen. You can have various types of words color coded to make it easier to read and modify the scripts. Many users have developed a syntax lists that are freely available. Also, both these editors allow you to run other programs from inside them. This means you can use RunEslBatch to run scripts directly from the editor windows. To do this you must first configure the editor to run this program correctly.

Note: UltraEdit is available from IDM Computer Solutions (www.ultraedit.com).

Note: TextPad is available from Helio Software Solutions (www.textpad.com)

Configuring for UltraEdit

To configure UltraEdit to run scripts with RunEslBatch, do the following:

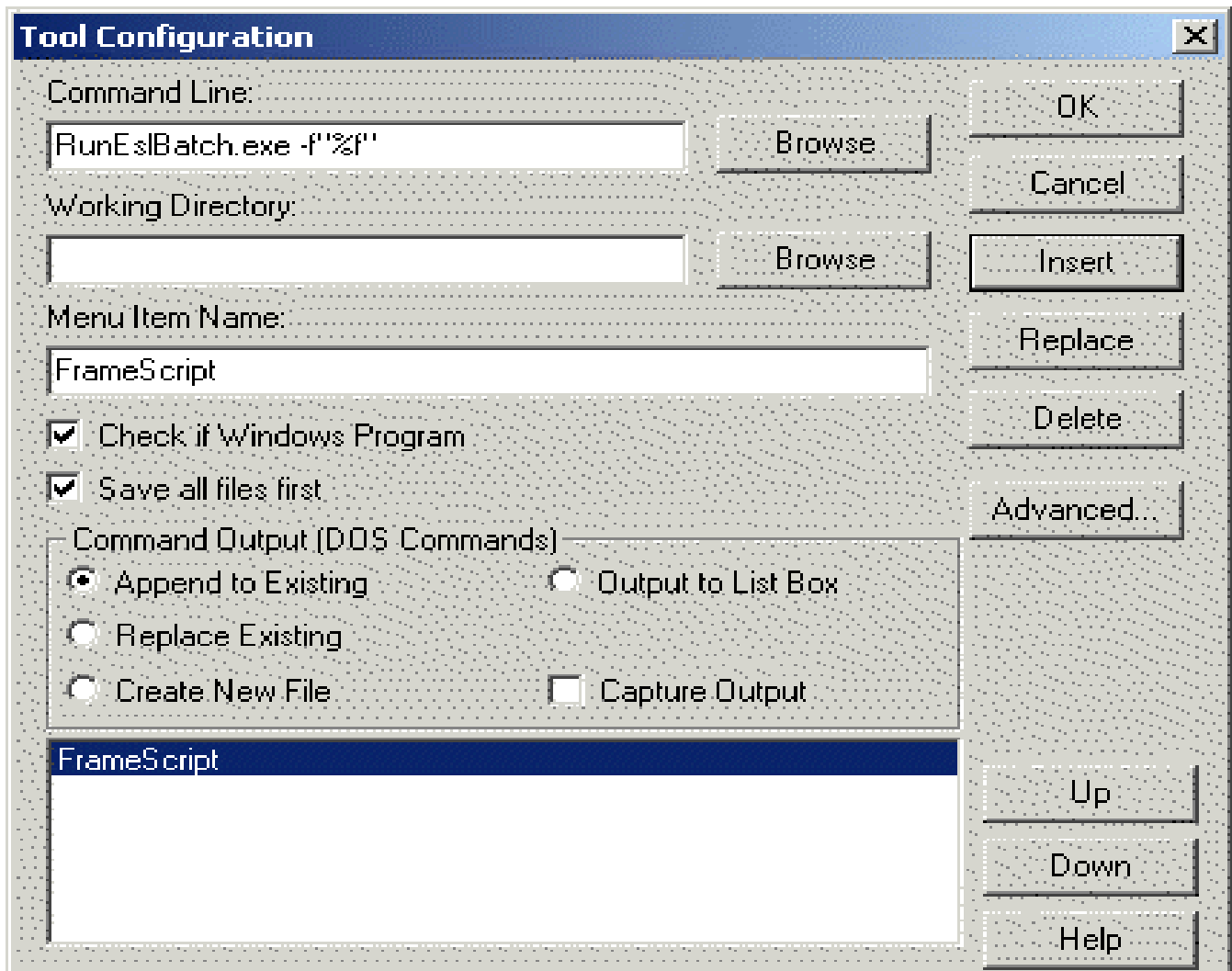
- From within UltraEdit, on the Advanced menu, select the Tool Configuration menu item (Advanced->Tool Configuration...).

- A dialog box appears that lets you define a tool. On the Command Line field, enter the following:

```
RunEslBatch.exe -f"%f"
```

- On the Menu Item Name field, enter some text of your choosing. This text will appear as a label of a menu item in UltraEdit.
- Check the box labeled 'Check if Windows Program'.
- Press the Insert button to register the new command to UltraEdit.
- Press OK to finish.

Figure 3-12 UltraEdit Tool Configuration dialog



A new menu item should now appear at the bottom of the Tools menu in UltraEdit. Whenever you select this menu item, RunEslBatch will run using the name of the currently active text file as its parameter. This means you can run the current script file whenever you wish just by selecting the menu item instead of switching to FrameMaker and using the Run command.

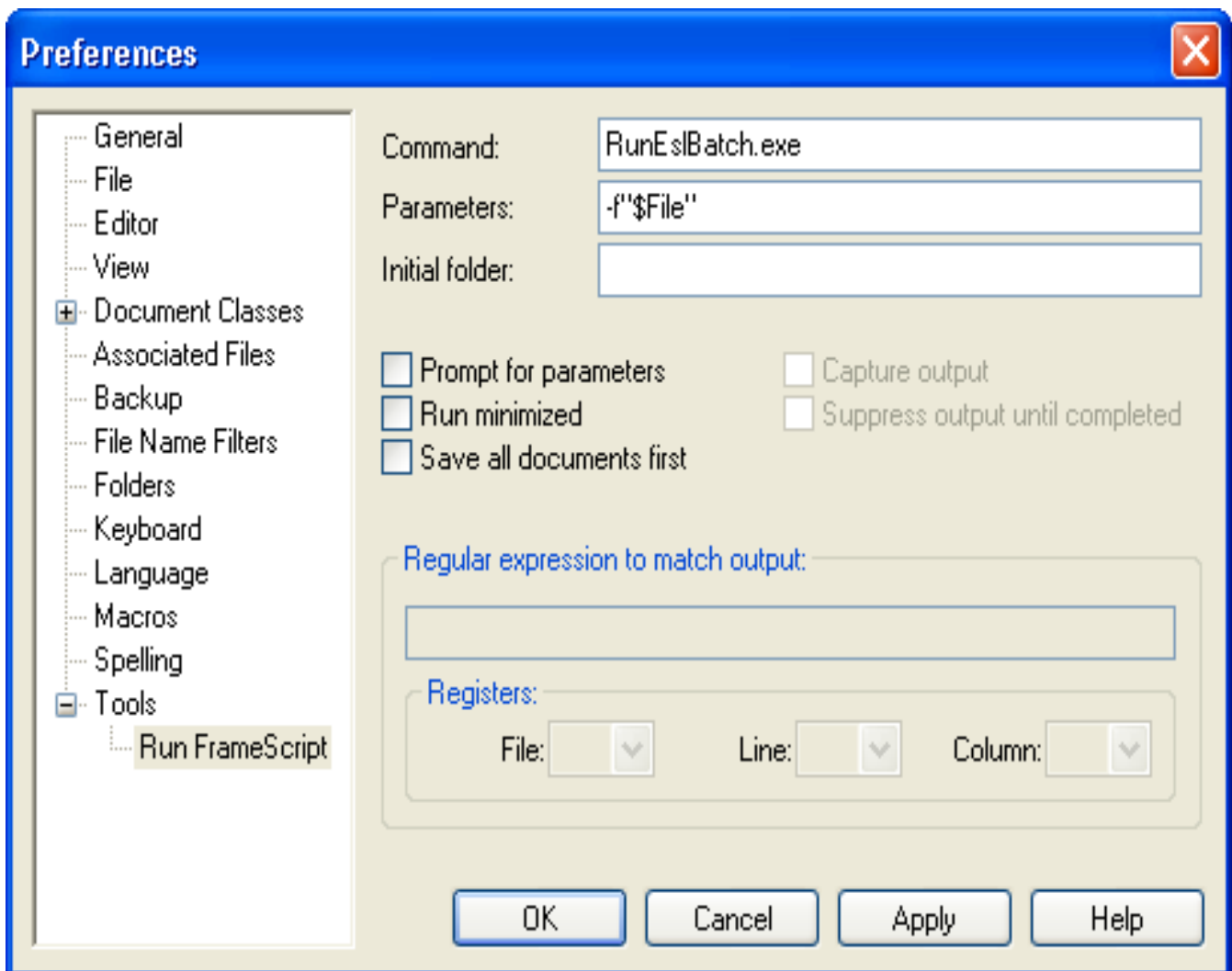
IMPORTANT: Make sure that the current file is saved before trying to run it.

Configuring for TextPad

To configure TextPad to run scripts with RunEslBatch, do the following:

- From within TextPad, on the Configure menu, select the Preferences menu item (Configure->Preferences...).
A dialog box appears.
- In the left panel, select the Tools item, then use the Add button (in the right panel) to add a new Program (Add->Program...).
This will take you to a file selection dialog.
- Navigate to the Windows directory and select the RunEslBatch program.
A new entry will appear in the list box.
- Rename this to something of your own choosing by selecting it, then making your changes.
- Press the Apply button to record the entry.
- In the left panel, expand the Tools item and select the new command.
A dialog appears on the right panel. See “TextPad Preferences for Tools dialog” on page 30
The Command line should be correct.
- In the Parameters text box, enter the following:
`-f"$File"`
- Press OK to finish

Figure 3-13 TextPad Preferences for Tools dialog



A new menu item should now appear at the bottom of the Tools menu in Textpad. Whenever you select this menu item, RunEslBatch will run using the name of the currently active text file as its parameter. This means you can run the current script file whenever you wish just by selecting the menu item instead of switching to FrameMaker and using the Run command.

IMPORTANT: Make sure that the current file is saved before trying to run it.

Chapter 4

Writing Scripts

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 it 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 it 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 it 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 48.).

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 occurrence 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 as `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 unspecified 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
   paragraph 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 2: 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 (they represent 23.5 and 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. See “Character sets” on page 41.
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.

Table 2: FrameScript Data Types

Data type	Description
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	<p>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</p> <p>tloc.Object is the object part</p> <p>tloc.Offset is the offset part.</p> <p>tloc.TextRange gives a text range from the text loc (begin and end)</p> <p>tloc.TextRange1 gives the text range for one text position ahead.</p> <p>You may also get all the text properties at the location represented by the TextLoc variable by using the .Properties property, as follow:</p> <pre>SET propList = tloc.Properties;</pre> <p>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:</p> <pre>SET colorVar = tloc.Color;</pre> <p>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.</p>

Table 2: FrameScript Data Types

Data type	Description
TextRange	<p>A TextRange identifier represents a range of text in a FrameMaker text object. A text range is just two TextLocs together. If <code>trange</code> is a TextRange identifier then:</p> <p><code>trange.Text</code> is a string representing the text within the text range.</p> <p><code>trange.Begin</code> is a TextLoc representing the starting text location.</p> <p><code>trange.End</code> is a TextLoc representing the ending text location.</p> <p>You may also access the TextLoc fields directly by:</p> <p><code>trange.Begin.Object</code> is the object of the beginning text loc.</p> <p><code>trange.Begin.Offset</code> is the offset of the beginning text loc.</p> <p><code>trange.End.Object</code> is the object of the ending text loc.</p> <p><code>trange.End.Offset</code> is the offset of the ending text loc.</p> <p><code>trange.Properties</code> allows you to assign properties (text properties) to the range of text 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 <code>PropList</code>.</p> <pre>SET trange.properties = PropList;</pre> <p>The <code>Text</code> 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 <code>TextSelection</code> 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.</p>
Point	<p>A Point identifier represents a point value for a FrameMaker graphic. If <code>pPoint</code> is a point variable then:</p> <p><code>pPoint.X</code> is the X offset value.</p> <p><code>pPoint.Y</code> is the Yoffset value.</p> <p>This is a FrameMaker specific data type.</p>

Table 2: FrameScript Data Types

Data type	Description
Tab	<p>A Tab identifier represents a tab value for a FrameMaker paragraph. If <code>tTab</code> is a tab variable then:</p> <p><code>tTab.X</code> is the offset value.</p> <p><code>tTab.Type</code> is the location type. This value can be one of the following:</p> <ul style="list-style-type: none"> <code>TabLeft</code> - Left Tab <code>TabRight</code> - Right Tab <code>TabCenter</code> - Center Tab <code>TabDecimal</code> - Decimal Tab <code>TabRelativeLeft</code> - Relative Left Tab (Format Change List Only) <code>TabRelativeRight</code> - Relative Right Tab (Format Change List Only) <code>TabRelativeCenter</code> - Relative Center Tab (Format Change List Only) <code>TabRelativeDecimal</code> - Relative Decimal Tab (Format Change List Only) <p><code>tTab.Decimal</code> is the Decimal Tab character</p> <p><code>tTab.Leader</code> is a string giving the characters before the tab.</p> <p>This is a FrameMaker specific data type.</p>
ElementLoc	<p>An <code>ElementLoc</code> identifier represents a location in a FrameMaker document or book. A <code>ElementLoc</code> 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 <code>ElementLoc</code> by specifying a modifier on the identifier as follows: if <code>eLoc</code> is an <code>ElementLoc</code> identifier, then</p> <ul style="list-style-type: none"> <code>eLoc.Parent</code> is the parent element. <code>eLoc.Child</code> is the child element. <code>eLoc.Offset</code> is the offset. <p>This is a FrameMaker specific data type.</p>

Table 2: FrameScript Data Types

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

Table 2: FrameScript Data Types

Data type	Description
AttributeDef	<p>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:</p> <p>attrDef.AttributeDefName is the name of the attribute definition.</p> <p>attrDef.AttributeDefChoices is this list of possible choices for the attribute definition. This is a StringList type.</p> <p>attrDef.AttributeDefDefaults is this list of default values for the attribute definition. This is a StringList type.</p> <p>attrDef.AttributeDefRequired is a boolean, value True-if it is required, False-if not.</p> <p>attrDef.AttributeDefFlags is a bitwise value: Possible values are:</p> <ul style="list-style-type: none"> AfReadOnly The attribute value is read-only AfHidden The attribute value is hidden. <p>attrDef.AttributeDefType is the type of the attribute definition: Possible values are:</p> <ul style="list-style-type: none"> 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 AtUniqueIDREFS One or more references to UniqueID attributes. <p>attrDef.AttributeDefRangeMin is the minimum value for a range test (if present):</p> <p>attrDef.AttributeDefRangeMax is the maximum value for a range test (if present):</p> <p>This is a FrameMaker specific data type.</p>
File	<p>A file identifier represents a text file on your computer system. This is a special type that is used only with the file commands.</p>
SubVar	<p>A subroutine identifier represents a subroutine in your script. You can use this as an easy way to access (RUN) a subroutine.</p>
LibVar	<p>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:</p> <ul style="list-style-type: none"> LibPath The name of the directory used as a LibVar

Table 2: 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 Reference Manual 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 individual 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," of the <i>FDK Programmer's Reference</i> . 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 2: FrameScript Data Types

Data type	Description
PropertyList	A list of properties for an object. This is retrieved with the <code>Get TextProperties</code> command or the <code>object.properties</code> 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 definitions for an element definition. This is a FrameMaker specific data type.

Character sets

FrameMaker uses a different character set for its string data than the standard MS Windows character set (Ansi). FrameMaker originated on the Unix platform and the character set it uses more closely resembles that of Unix. Most of the common characters in the FrameMaker character set and the Ansi character set, however, are identical so it does not matter to the majority of FrameMaker users. The differences lie in the upper ascii area. This area contains characters (such as Ä, È, ®) that contain accents or special symbols. The FrameMaker character set issue affects mostly European and South American customers (Asian customers use the MBCS (multi-byte character) supported by Adobe).

If you do not use these special characters in your documents and/or file names, then you do not have to worry about this distinction. If you do use these characters, then you should continue reading this section.

Problems arise sometimes with file names that use special characters. It also might occur if you insert text typed directly into a script (or read from a text file) into a FrameMaker document. To ameliorate this problem, we have added the PlatformEncoding option to this release. This is a session variable that applies to each script individually. If you set this value to True, the FrameScript will automatically convert any string value coming from FrameMaker from the Frame Character set to the Ansi (Windows) character set and convert any string value going to FrameMaker from the Ansi character set to the Frame character set. This alleviates you from having to be concerned about it. We recommend that any new script written, which might use special characters, put the following line at the beginning of the script.

```
Set PlatformEncoding = True;
```

Of course, this is not backward compatible with previous versions of FrameScript.

See “PlatformEncodingMode” on page 47.

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 3: Standard Object Properties

Property Name	Description
<code>dataname.Objectname</code>	Returns a string value containing the name of the data type. This applies to any variable or property. Possible values are 'String' 'Integer' 'Doc'
<code>dataname.Size</code>	Returns the physical size of the data item (length of the string for string variables or properties).
<code>dataname.Count</code>	Returns the number of items in a list object. This value is 1 for single data items.
<code>dataname.Properties</code>	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.
<code>dataname.Valid</code>	True or False, depending on whether the variable contains a valid object. The dataname must be a FrameMaker object variable or property.
<code>dataname.Text</code>	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.
<code>dataname.IsObject</code>	True or False, depending on whether the variable is a FrameMaker object.
<code>dataname.Doc</code>	The document part of a Frame Object variable.
<code>dataname.Menu</code>	The Menu part of a Command, Menu or MenuItemSeparator Object variable.
<code>dataname.Page</code>	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.
<code>dataname.Pgf</code>	The Paragraph object that contains the Frame Object. If the dataname is not an object or if it does not reside in a pgf (e.g. paragraph format), then this value is zero. Tables, Table Rows and Table Cells return the paragraph of the table anchor. Objects inside an anchored frame return the paragraph of the frame anchor. Paragraphs return themselves.
<code>dataname.IsParm</code>	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 hexadecimal value by using the H suffix and a binary value by using the B suffix, as follows:

```
10H - indicates a hexadecimal 10 which is 16 decimal.
10B - indicates 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 enclosed 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 constant values identified by reserved names. You may use these identifiers in place of the values they represent.

Table 4: 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.
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. <code>e.g. set str = 'Can' + QUOTE + 't do it';</code> 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 with 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 <code>True</code> . Otherwise it is <code>False</code> .
>	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 <code>True</code> . Otherwise it evaluates to <code>False</code> .
<	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 <code>True</code> . Otherwise it evaluates to <code>False</code> .
>=	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 <code>True</code> . Otherwise it evaluates to <code>False</code> .
<=	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 <code>True</code> . Otherwise it evaluates to <code>False</code> .
not	Not operator: This operator reverses the effect of the following operator. E.g., <code>not =</code> means not equal
and	<p>And operator: This operator allows you to combine several comparative expressions into the same expression. Both expressions must be <code>True</code> for the entire expression to be <code>True</code>. E.g.</p> <p style="text-align: center;">a = b and c = d</p> <p>This expression evaluates to <code>True</code> if both a is equal to b AND c is equal to d.</p>
or	<p>Or operator: This operator allows you to combine several comparative expressions into the same expression. If either expression is <code>True</code> the entire expression is <code>True</code>. E.g.</p> <p style="text-align: center;">a = b or c = d</p> <p>This expression evaluates to <code>True</code> if either a is equal to b OR c is equal to d.</p>

- .** (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 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 5: List of Fixed FrameScript 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.
PlatformEncodingMode	An integer variable indicating whether platform encoding mode is enabled. If this variable is False (default), then there is no automatic translation from the FrameMaker character set and the platform character set. If this value is True , then FrameScript will automatically convert any strings coming from FrameMaker to the platform character set (Ansi for Windows) and it will convert any strings going to FrameMaker to the FrameMaker character set. This is especially useful if you use characters in the upper ascii range (above 127).

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:

```
FirstPgFmtInDoc, FirstPgInDoc, 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);
Run gvVector.PushBack Value(100);
Run gvVector.PushBack Value('String Data');
Loop InitVal(3) Incr(1) LoopVar(gvIdx) While(gvIdx <=10)
    Run gvVector.PushBack Value(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 utility 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);
Run gvCollVar.PushBack Value('First Memeber') Value('Second Value'); // Add two values
Set gvMember = gvCollVar.FirstMember;
Loop While (gvMember)
    write console 'Member='+gvMember.Value;
    Set gvMember = gvMember.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'].Value; // 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.

<code>ActiveDoc</code>	A property name representing the currently active document.
<code>4.33"</code>	A metric constant indicating a value of 4.33 inches.
<code>a * b + c</code>	An arithmetic expression (assuming that a, b and c are variable), where a and b are multiplied together then the value of c is added to the result.
<code>a * (b + c)</code>	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.
<code>'This is' + ' an expression'</code>	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'
<code>ActiveDoc.SnapGridUnits + 100pts</code>	This is an expression which adds the value of the SnapGridUnits property of the currently active document and 100 points together.
<code>ActiveDoc.Name+' is the filename'</code>	This expression concatenates the file name of the currently active document with a string.

Basic commands

The most basic commands are the following: Set, If, and Loop. The Set command assigns a value to a script variable or to a property of an object. The If command lets you do conditional commands and the Loop command allows you to repeat a series of commands until (or while) some condition is true.

Creating and Deleting Data

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. This is the general purpose command for creating objects, but it can also be used to create simple data items.

Built-in Dialogs

Although you can create custom forms (see the `EsObjects.pdf` document), it is sometimes easier to use the common dialogs. Common dialogs (`DialogBox` command) are available to select a file (`ChooseFile`), select an item from a list (`ScrollBox`) or enter a few lines of data (`MEdit`). There are also commands for displaying messages to the user, such as `MsgBox` and `Display`.

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 consist 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 to 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 subroutines. 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.

```
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 sbChangePgFormat;

Sub sbChangePgFormat;

  Get Object Type(pgFmt) 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 53), 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 sbChangePgFormat 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 minimize 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 variable and 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. If you do not include an initial 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 sbChangePgFormat pvFromFmtName('Heading1') pvToFmtName('Heading2');

Sub sbChangePgFormat using pvFromFmtName pvToFmtName;
  Local lvToFmt lvPgFormat;

  Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
  Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgFormat)
    If lvPgFormat.Name = pvFromFmtName
      Set lvPgFormat.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 sbChangePgFormat pvFromFmtName('Heading1') pvToFmtName('Heading2')
    returns pvPgChgCount(gvCount);
Display 'The number of paragraphs changed was '+gvCount;

Sub sbChangePgFormat using pvFromFmtName pvToFmtName pvPgChgCount;
    Local lvToFmt lvPg lvCount(0);

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

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

Function fnChangePgffFormat using pvFromFmtName pvToFmtName;
  Local lvToFmt lvPgff lvCount(0);

  Get Object Type(pgffFmt) Name(pvToFmtName) NewVar(lvToFmt);
  Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgff)
    If lvPgff.Name = pvFromFmtName
      Set lvPgff.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 in any order we wished on the **Run** command. With functions the order matters, as follows:

```

Run sbChangePgffFormat pvFromFmtName('Heading1') pvToFmtName('Heading2')
  returns pvPgffChgCount(gvCount);

```

is the same as

```

Run sbChangePgffFormat pvToFmtName('Heading2') pvFromFmtName('Heading1')
  returns pvPgffChgCount(gvCount);

```

On the other hand:

```

Set gvCount = fnChangePgffFormat{'Heading1','Heading2'};

```

is very different than

```

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

Function fnChangePgFormat using pvFromFmtName pvToFmtName ByRef pvTotal;
  Local lvToFmt lvPgfs lvCount(0) lvTotalPgfs(0);

  Get Object Type(pgfFmt) Name(pvToFmtName) NewVar(lvToFmt);
  Loop ForEach(Pgf) In(ActiveDoc) LoopVar(lvPgfs)
    Set lvTotalPgfs = lvTotalPgfs + 1;
    If lvPgfs.Name = pvFromFmtName
      Set lvPgfs.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.

Calling a Function

The following shows the format of calling a Function.

Format:

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

Table 6: 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 subroutine 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.

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 referred 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 EsObjects, but that is described in the EsObject 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. 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. 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. 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 or 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 subtype Sub1 located in the identified 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 preferred 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

```

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 somewhere 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 utility 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 utility 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 utility 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) pvPgFmt(pgFmtVar);
```

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 **pgFmtVar** is a paragraph format object that specifies the paragraph format for this paragraph. The **pvPgFmt** parameter is optional.

Example:

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

```

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 somewhere 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 lvPgffSourceList(DocPgffFmtNameList);
Local lvPgffList;
Set lvPgffList = eDlgScript.DlgStringDualSelect{lvPgffSourceList,,
'Select Paragraph Formats',
'Paragraph Formats',
'Selected Formats'};
...
```

In this example, a list of paragraph 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 somewhere 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
```

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.

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 Reference Manual 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.

<code>FrameDoc</code>	The document object of the document containing the hypertext marker.
<code>FrameObject</code>	The object variable of the marker causing the hypertext event.
<code>Message</code>	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
```

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 7: 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
-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

Table 7: List of Error Messages

Errorcode value	Error Message Description
-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
-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

Table 7: List of Error Messages

Errorcode value	Error Message Description
-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
-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

Table 7: List of Error Messages

Errorcode value	Error Message Description
-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

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 using them on live documents.

Reserved Words

One common error is using reserved words as 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 5

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 paragraphs in order, you have to go through the flow structures.

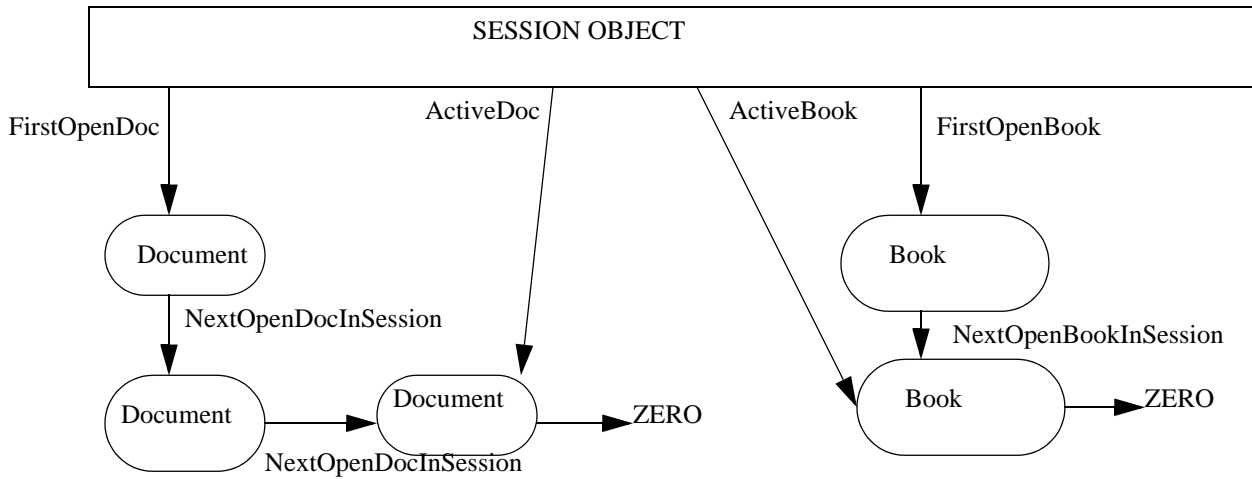
Session Object

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 8: List of Session Architecture Properties

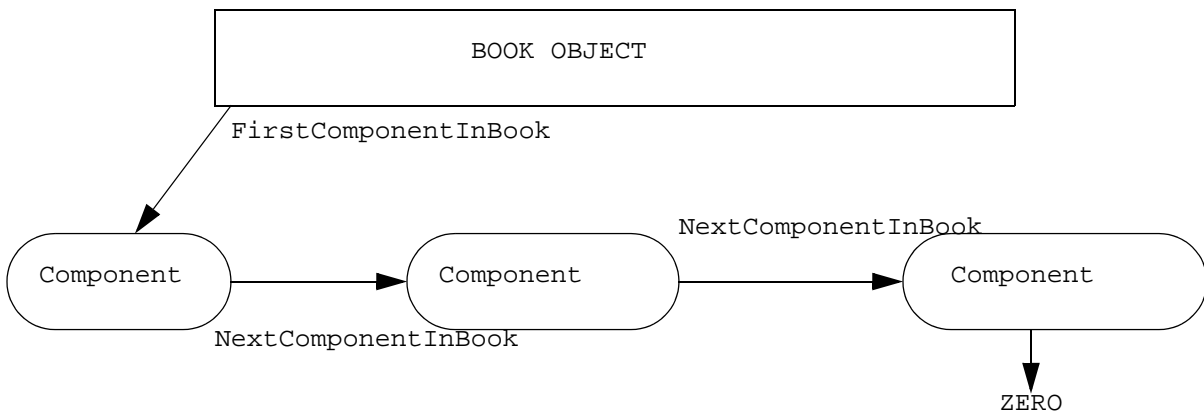
Property Name	Description
<code>ActiveDoc</code>	The object of the currently active document.
<code>ActiveBook</code>	The object of the currently active book.
<code>FirstOpenDoc</code>	The first in a list of open documents
<code>FirstOpenBook</code>	The first in a list of open books.

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.

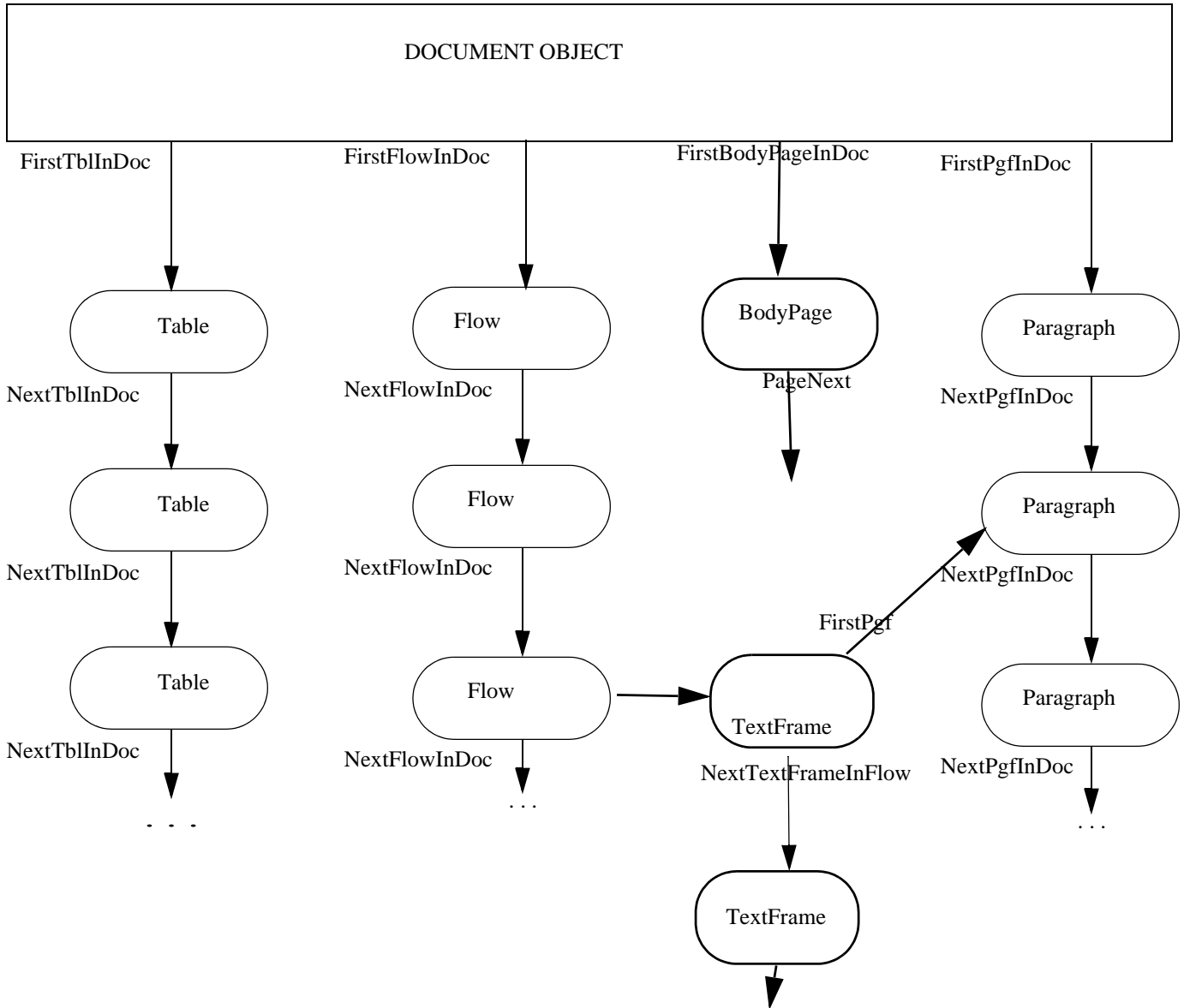


Document Object

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.

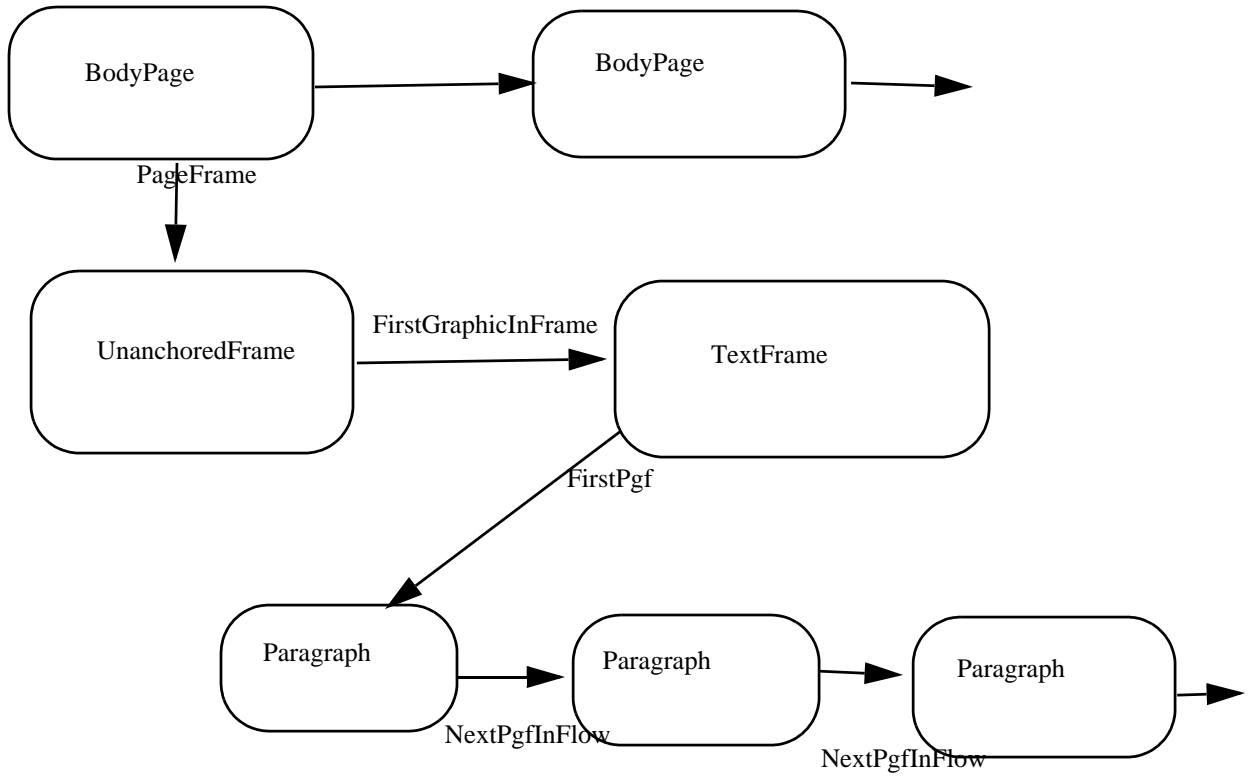
These lists include a list of all the marker objects in a document, all the body pages, all the character formats, all the paragraph formats, all the flows, all the tables, and so on. For a complete list of object lists for a document see the Scriptwriter's Reference.

The following illustrates three of the lists and structures.



Body Page

The following illustrates the Body Page Object.



Chapter 6

Using ElmStudio

Introduction

ElmStudio is a text editor and source code debugger for FrameScript scripts. The editor allows you to create, open, modify, save and run scripts without leaving the FrameMaker environment. Although you can edit any reasonably sized text file, there are special features available for developing FrameScript scripts. The editor follows the standard conventions of most text editors (such as NotePad, WordPad, and other stand-alone commercial text editors). There is a File menu with standard file operations (New, Open, Save, etc.), an Edit menu with cut, copy, paste, bookmarks, etc., a search menu with find text commands and so forth. In addition to these standard operations, it has syntax coloring (for any script with an .fsl extension), folding, word completion and running scripts.

IMPORTANT: The ElmStudio editor is based on the public domain SciTE editor. For complete documentation of this stand-alone editor is located at <http://scintilla.sourceforge.net/SciTEDoc.html>. ElmStudio is a subset of this editor geared toward writing scripts. It does not have all the features of the SciTE editor, which is designed as a stand-alone IDE for many languages.

Editor Menus

File

The File menu is primarily concerned with operations on the entire text file, such as creating, opening, saving, etc.

Table 9: File Menu Items

Data type	Description
New	Creates a new empty text file. Note: There is a current limit of 10 open documents at a time.
Open	Opens an existing text file. If the file extension is .fsl, then syntax highlighting will be invoked. Note: There is a current limit of 10 open documents at a time.
Open Selected Filename	If you select some text and that text is a file name, this command will open the file. Note: There is a current limit of 10 open documents at a time.
Revert	The command will reload the last saved version of the file, removing any unsaved changes.
Close	Closes the current text file. You will be prompted to save the file if any changes have been made but not saved.
Save	Saves the current text file, replacing the currently saved file. No prompting will be done.
Save As...	Saves the current text file after prompting you for a new name. The file name in the text editor will change to the new file name.

Table 9: File Menu Items

Data type	Description
Save A Copy...	Saves the current text file after prompting you for a new name. The file name in the text editor will remain the same.
Encoding	Allows you to change the current character encoding. Note that this is primarily for non-script text files. FrameScript only recognizes standard 8-bit encoding. Changing to another encoding value might create an invalid script file.
Export	Allows you to save the text file into Html, Xml, Rtf, Latex or PDF format.
Page Setup	Displays a dialog allowing you to change the page setup (Margins, printer, etc) for printing scripts . This is <i>not</i> the same as the Print setup for FrameMaker, which is for FrameMaker documents.
Print	Prints the current document.
Exit	Quits the editor.
At the bottom of this menu is a list of the most recently opened text files.	

Edit

The Edit menu has commands for editing the characters in a text file, such inserting characters, cut/copy/paste, etc.

Table 10: Edit Menu Items

Data type	Description
Undo	Undo last command or text operation.
Redo	Redo last command.
Cut	Delete the current selection and send to the clipboard.
Copy	Copy the current selection to the clipboard.
Paste	Paste the text contents of the clipboard to the current insertion point.
Delete	Delete the current selection.
Select All	Select all the text in the current document.
Copy as RTF	Copies the current selection to the clipboard as rtf text.
Complete Word	The editor scans the other words in the document and presents a list that matches the current word prefix.
Expand Abbreviation	The editor scans the abbrev.properties file for a match for the text at the cursor. See “Abbreviations file” on page 88 for information about the abbrev.properties file.
Insert Abbreviation...	A dialog appears allowing you to select an abbreviation (from the abbrev.properties file) to expand. See “Abbreviations file” on page 88 for information about the abbrev.properties file.
Block Comment or Uncomment	Comments each line of the selection with a //~ set of characters. If the lines have been previously commented using this command, they will be uncommented. Note: The double slashes produce the actual commenting characters, the ~ character is used as an indication that this menu command produced this result. If a line has been commented manually, it will not uncomment it correctly.
Box Comment	Comment the selected text using the /* in the beginning of the block and the */ at the end. It also places an * at the beginning of each line (to box the comment).
Stream Comment	Comment the selected text using the /* in the beginning of the block and the */ at the end.

Table 10: Edit Menu Items

Data type	Description
Make Selection Uppercase	Makes the current selection into all uppercase characters.
Make Selection Lowercase	Makes the current selection into all lowercase characters.

Search

The Search menu has commands for navigating the text file.

Table 11: Search Menu Items

Data type	Description
Find...	Brings up the Find Text dialog.
Find Next	Finds the next occurrence of the previously found text.
Find Previous	Finds the previous occurrence of the previously found text.
Find In Files...	Finds text in a set of files.
Replace...	Find and Replace within the document.
Go To...	Moves the cursor and scrolls the window to a specified line number.
Next Bookmark	Moves the cursor and scrolls the window to the next bookmark in the document.
Previous Bookmark	Moves the cursor and scrolls the window to the previous bookmark in the document.
Toggle Bookmark	Set a bookmark at the current line if none exists or remove it if it does.
Clear All Bookmarks	Remove all bookmarks in the current document.

View

The View menu has commands for show or hiding various editor interface elements.

Table 12: View Menu Items

Data type	Description
Toggle Current Fold	Toggle the fold (expanded/contract) where the cursor resides.
Toggle All Folds	Toggle (expanded/contract) all folds.
Tool Bar	Show/Hide the tool bar.
Tab Bar	Show/Hide the file name tab bar.
Status Bar	Show/Hide the status bar.
Whitespace	If checked, show the whitespace characters with dots, otherwise hide the dots.
End Of Line	If checked, show the end of line characters (CR LF), otherwise hide them.
Indentation Guides	If checked, show the graphical vertical indentation lines, otherwise hide them.

Table 12: View Menu Items

Data type	Description
Line Numbers	If checked, show line numbers in the margin, otherwise hide them.
Margin	Show/Hide the gray margin area.
Fold Margin	Show/Hide the fold margin area.
Output	Show/Hide the output panel.
Parameters	Present a dialog box where you can enter up to four parameters that will be sent the script when you run it. This can be helpful for testing a script that you intend to use as a subroutine.

Exec

The Exec menu has commands for running scripts and for analyzing the result.

Table 13: Exec Menu Items

Data type	Description
Run/Go	This command starts running the script in the currently active text window.
Next Message	Move cursor to the next error message in the Output Pane.
Previous Message	Move cursor to the previous error message in the Output Pane.
Clear Output	Clear the output pane.
Switch Pane	If the cursor is in the output pane, move to the text pane. If the cursor is in the text pane, move the cursor to the output pane.

Debug

The Debug menu has commands for the interactive source debugger. See “Interactive Debugger” on page 88 for detailed information on the debug commands.

Table 14: Debug Menu Items

Data type	Description
Run In Debug Mode	Start a script in debug mode. Execution will stop at the first command. Note: you can only run in debug mode if the file is saved.
Stop Debug	Stop a debugging session. The script will stop when you run this command.
Step Into	Execute the current command and stop at the before next command. You can use this command to step through the commands one at a time.
Step Over	Execute the current command and stop at the before next command, but skip going into any subroutine or function.
Run To Cursor	The script will run until it gets to the command where the cursor resides. If that command is not reached, then the script will continue until a break point is encountered or the end of the script occurs.

Table 14: Debug Menu Items

Data type	Description
Toggle Breakpoint	Sets a breakpoint (red dot) at the line where the cursor reside if none exists. If there is already a break point at this line, then it wil remove it.
Clear All Breakpoints	Clears all break points in the current file.
Examine Dataspace	Displays a modeless dialog which allows you to inspect the current variables (Script, Local, Parameter, as well as Globals). You can also set watches, where you can enter your own expressions.

Options

The Options menu has items that allow you to change some over all options for the editor.

Table 15: Options Menu Items

Data type	Description
Always On Top	Keep the editor window as the top most window.
Vertical Split	If checked, make the output panel vertical, otherwise it will be horizontal.
Wrap	If checked, line wrapping will take place, otherwise no line wrapping.
Line End Characters	Specify the line end characters to use (CR+LF, CR, LF). Note this will only change the characters for any additional line ends that occur. To convert all line ends to the new characters, use the following convert command.
Convert Line End Characters	This command will convert all the line end characters to the specified line end settings.
Change Indentation Settings...	Displays a dialog that allows you to set the tab size, indent size and whether to use actual tab characters or space.
Use Monospaced Font	Changes to to a mono-spaced font.

Windows

The Windows menu has items that allow you to navigate through the open windows.

Table 16: Windows Menu Items

Data type	Description
Previous	Make the previous window active.
Next	Make the next window active.
Close All	Close all text windows.
Save All	Save all open documents.
At the bottom of this menu is a list of names of the open documents. You can select them and make them the active window.	

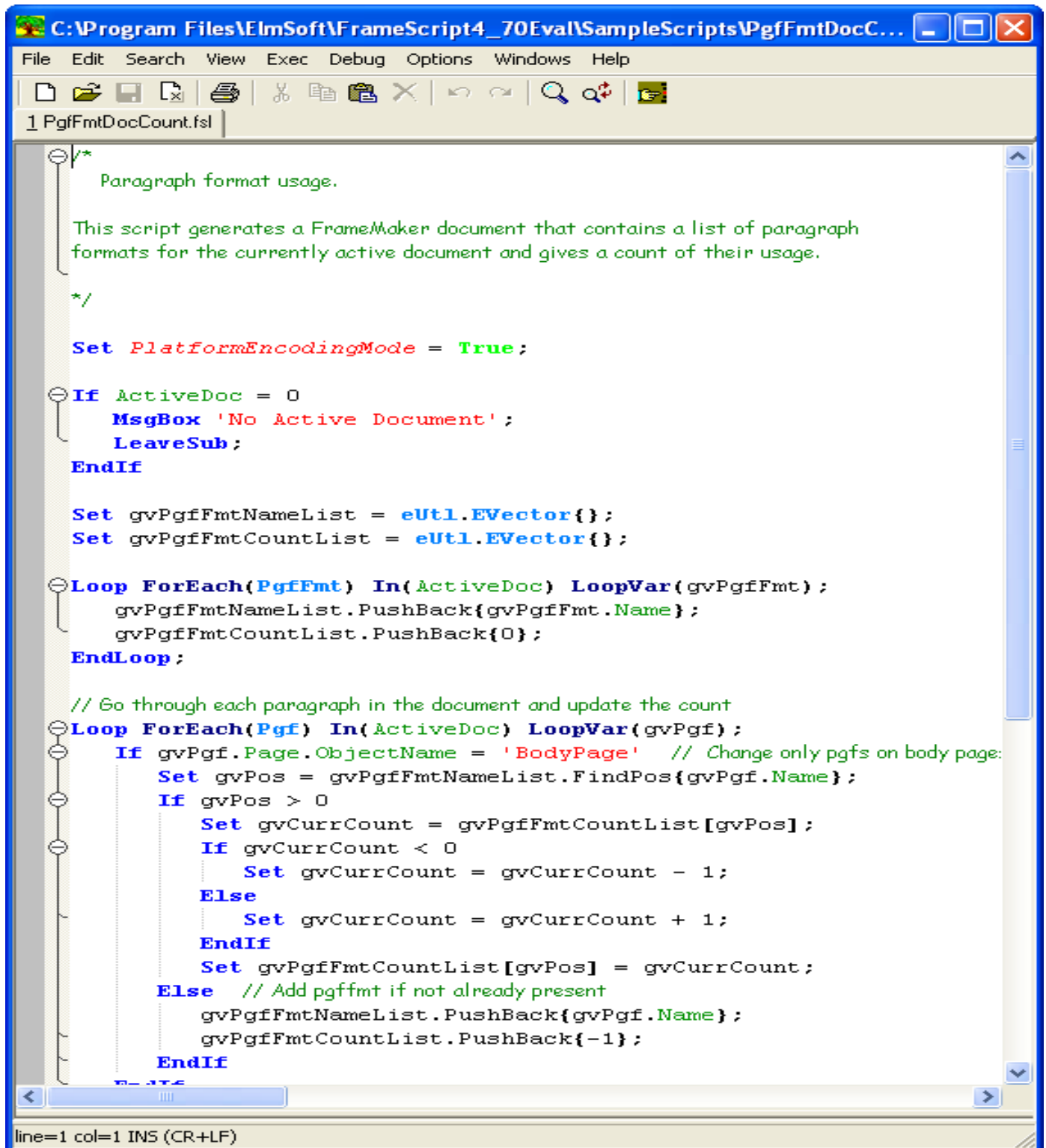
Help

The Help menu has items that allow you to display various help files.

Table 17: Help Menu Items

Data type	Description
Help	<p>Displays a FrameScript help file. The default value is to bring up the FrameScript RefManual.pdf file. To bring up your own file (such as an HTML file) modify the entry in the ElmStudioUser.properties file to use the following entry:</p> <pre>command.help.*.fsl="\$(ElmStudioUserHome)\MyFile.html"</pre> <p>Replace the Myfile.html with the name of your own file. The file should be in the FrameScript folder.</p>
Editor Help	<p>Displays a help file for using the editor. By default, it brings up the FrameScript UsersGuide.pdf file. If you want to bring up your own file, add the following entry to the ElmStudioUser.properties file:</p> <pre>command.scite.help="\$(ElmStudioUserHome)\MyFile.html"</pre> <p>Replace the Myfile.html with the name of your own file. The file should be in the FrameScript folder.</p>
About ElmStudio	<p>Displays the standard "About..." screen.</p>

Figure 6-1 ElmStudio Window



Customization

The editor has three customization files, `ElmStudioGlobal.properties`, `ElmStudioUser.properties` and `ElmStudio.properties`. The options in the `ElmStudio.properties` file override those in the `ElmStudioUser.properties` file which, in turn, override the options in the `ElmStudioGlobal.properties` file. The options in these '.properties' files have the following format:

```
[#]optionname[.optname2[.optname3]]=value
```

A '#' in the first position indicates a comment line.

For most users there should be no need to change these files. The `ElmStudio.properties` file is updated by ElmStudio itself reflecting any changes made by the user with the interface, such as window position, view settings, etc. This file will be rewritten everytime ElmStudio quits. This file should not be changed except in unusual circumstances. The `ElmStudioGlobal.properties` file contains the predefined options and also should not be changed. If you find it necessary to customize the ElmStudio editor (colors, syntax highlighting, etc.), you should modify the `ElmStudioUser.properties` file. See the web site at <http://scintilla.sourceforge.net/SciTEDoc.html> for information about the possible options.

Abbreviations file

The file `abbrev.properties` contains the abbreviations used in the 'Expand Abbreviation' and 'Insert Abbreviation...' menu commands. You can add your own abbreviations to this file if you wish. The format for this file is simple. It contains a list of abbreviations (one per line) in the following form:

```
abbreviation=expansion
```

See the file itself for examples.

Interactive Debugger

Up until now, debugging a script usually involved some of the following steps.

- Examining the script in the text editor. Look at it and hope that something becomes clear.
- Inserting some debug statements, such as `Display` or `Write Console`, at various places in the script and seeing what values are stored at various times. This can be hit and miss.
- Use the Trace facility (eDebug, available in FrameScript Version 3 or greater) to follow the order of the commands as they are executed. This tends to generate a lot of output.

Now, with FrameScript 4, you can now do interactive debugging using ElmStudio. When you start a script in debug mode, you can stop it anywhere you choose and examine the contents of any variables at that point in time. You can also step through a script command by command or run the script to some predefined point (breakpoint).

To run a script from the ElmStudio, you open the script file (or type the script into a new document) then select the `Exec->Run/Go` menu item (or press F5 or press the `Run Script` button on the toolbar). The script does not have to be saved to a disk file, it will use the text in the window as the script.

To run a script in debug mode, however, you must have a saved script file. Open a script file (or type in a new one), and if you are any unsaved changes be sure to save them. Use the `Debug->Run in Debug Mode` command to start the script in debug mode.

When you run a script in debug mode, the script stops at the first executable command. From here you can press the F11 key to step through the script command by command or you can set breakpoints at various places then press F5 to run the script until it reaches one of the breakpoints (or the end of the script).

IMPORTANT: Scripts running in debug mode will run slower than normal.

Setting Breakpoints

You set breakpoints using the Debug->Toggle Breakpoint menu command. This is also possible using the right-click menu. Move the text cursor to the line where you want to insert the breakpoint, then do the command. A red dot should appear in the margin. To remove a breakpoint, repeat the procedure. You can also clear all breakpoints in the script file by selecting the Debug->Clear All Breakpoints menu command.

IMPORTANT: Breakpoints are disabled when a modal form appears on the screen. A modal dialog intercepts all the keystrokes and other events from the application, making it impossible to step through the script or interacting with the debugger in any way. You can set breakpoints, but they will be ignored as long as the modal form is on the screen.

Examine Dataspace Window

The examine dataspace window lets you look at the current state of the user defined variables. There are five nodes in the main window. These are labeled Global, Script, Local, Parm and Watchlist. The Global variables are those that you defined in the Initial script. The script variables are variables the you create as part of your script. The Local variables are those created with the Local command and Parm variables are the current set of parameters if you are inside a subroutine or function. Watch expressions can be created by pressing the "Add Watch..." button and then entering an expression. All these values are recalculated whenever the script reaches a stopping point (such as a breakpoint or a step). To delete a watch in the watchlist, select the item and press the "Delete Watch" button.

Sample Debug Session

The following is a short tutorial on using the interactive source debugger. The script for this example is designed to illustrate the features of the debugger. It performs a simple function. It is suppose to count the number of paragraphs in the current document and also count the number of empty paragraphs. For some reason, it is not working correctly. It is saying that there are no empty paragraphs and we know there must be some empty paragraphs (on reference pages for example). We will try to find the problem using the interactive debugger.

- Start FrameMaker
- Open (or create a new) document.
- Start ElmStudio (FrameScript->Script Window...).

In the following steps, the menu selections are from the ElmStudio window.

- Turn on line numbering (View->Line Numbers).
- Open the DebugTest.fsl script, located in the Demos\DebugDemo folder.
- Just to see the problem, run the script without the debugger, by pressing the Run button (or selecting the Exec->Run/Go menu item).

A dialog box should appear telling you that there are some number of paragraphs but zero empty paragraphs. Now lets debug the script.

- Start the script in debug mode (Debug->Run in Debug Mode).

The script should stop at the first command and there should be a yellow triangle in the margin of line number 2. The yellow triangle in the margin will indicate the line of the command that is about to be executed.

- Bring up the Dataspace window (Debug->Examine Dataspace).

The dataspace window should appear showing the various catagories of data.

- Expand the Script variables node.
- Make the ElmStudio window active by clicking in the caption area.

Whenever you leave the ElmStudio window you have to make it active again or it will not recognize the keyboard commands.

- Press F11.

The yellow triangle should move to the next line (line 3). Notice also that a value appears in the dataspace window, under the script variables.

- Press the F11 key a few more times until the triangle gets to line 14 (the start of the loop command). Watch the data window as the script proceeds to see the changes in the script variables.

At this point you can continue to use the F11 key and step through each command. However, this loop will continue for 100 iterations and stepping through that many commands will be time consuming and tedious. Unless we are looking for a bug that is occurring inside the loop, we might want to just let the script run until the loop is finished.

- Move the text cursor to line 19 (a set command). Select the "Run to Cursor" EmStudio menu command (Debug->Run to Cursor).

The script should run and stop at line 19.

- Look at the dataspace window and see the results of the calculation from the loop (the gvCount and gvSum variables).

Lines 19 is a set command with user functions in the expression. If you wish to enter the function and follow the script use the F11 key. If you wish to "step over" the function and go to the next command (line 20), use the F10 key.

- Press F11.

The triangle should move inside the function (to line 44). Look at the dataspace window. Expand the parm variables node. You should see the values for the two parameters passed to the function.

- Make the ElmStudio Window active again by clicking in the caption area.
- Press F11.

A local variable should now be created. Expand the Local variable node in the dataspace window to see this. You should also see the Result variable.

- Make the ElmStudio Window active again by clicking in the caption area.
- Step through (press F11) the rest of the function until it finishes and the yellow triangle goes out of the function to line 20 (the line after the previous command). Notice that the local and parm variables are now gone. They are deleted as soon as the function ends.
- Since (in this example) we do not need to go into the function again. Press F10 to skip over the function call. The yellow triangle should skip to line 22. Remember, the function was called as usual. The F10 (step over) key just stopped the debugger from showing it. In the dataspace window you should see the results of the function calls in the variables gvVar3 and gvVar4.
- The loop command starting at line 22 is very short. Press the F11 key repeatedly and watch the script go through it. See the index variable (gvIdx) increase as the loop proceeds.
- When the loop finishes the yellow triangle should be at line 26. Stop here.

Now we are down to the part of the script that actually counts the number of paragraphs in the active document. If you did not open (or create a new) document, the script will stop after the **If** command on line 26. If you did open (or create a new) document, then we can proceed.

Looking ahead we can see the **Loop** that scans through all the paragraphs and we can also see that the **gvPgEmptyCount** variable is suppose to accumulate the number of empty paragraphs. Since we want to look at the text property of the each paragraph to see if it is empty, we will add a watch to the examine dataspace window.

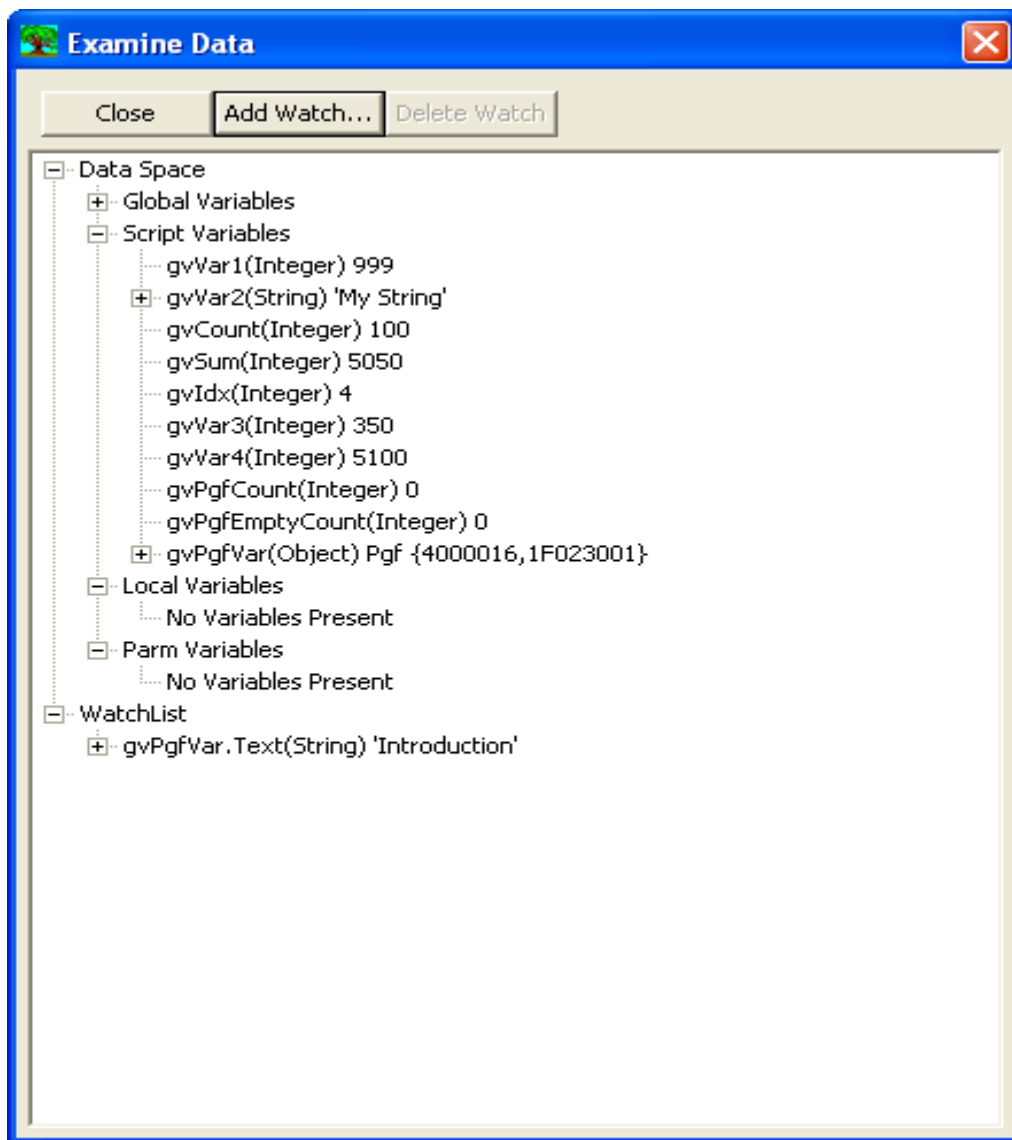
- In the examine dataspace window, press the "Add Watch..." button.
- Enter the following in the resulting dialog box:
`gvPgFVar.Text`
- Press OK.

A new node appears under the watchlist node. This will let you see the value of a specific property. Notice that the value of this is currently Null. At line 26 we have not yet created this variable.

- Press F11 4 times to get to line 30. See "ElmStudio Window with script running in debug mode" on page 94 for a picture showing how the window should look.

Now we should see a string value in the Text property of `gvPgFVar`. It might be an empty string or it might be a real string value. It depends on the document you have open. See "ElmStudio Examine Dataspace Window" on page 92 for a picture of the examine dataspace window at this point in the run.

Figure 6-2 ElmStudio Examine Dataspace Window



At this point, we could step through the loop until we reach an empty paragraph, then see what happens to the counter, but this could take a long time. We see that the **If** command on line 32 checks for empty paragraphs (text size is less than 1). We want to see what happens when we get there. So, to save time, we will set a breakpoint inside the **If** command on line number 33.

- Move the text cursor to line 33 (using the mouse or arrow keys).
- When the cursor is on line 33, use the toggle breakpoint command (Debug->Toggle Breakpoint).

A red circle should appear in the margin marking this line as a breakpoint line.

- Press the F5 key.

The script will continue running until it reaches line 33. If it never reaches line 33, it will continue to run until the end of the script. But, of course, our script should stop there.

When it stops at line 33, look at the current value of `gvPgfEmptyCount` in the dataspace window. It should be zero. Also look at the examine dataspace window and look at the value of `gvPgfVar.Text`. It should be an empty string. This means we have not made a mistake with our test.

- Press F11 to do the command and see what happens to the counter.

Look at the counter again. The value is still zero. What went wrong?

If we look closer at the `Set` command, we will discover that we used the wrong operator. To count values we want to add them, therefore we want to use the plus (+) operator, not the multiplication operator (*).

Now that we've found the problem, we do not need to keep debugging any longer. We can clear all the breakpoints (Debug->Clear All Breakpoints) and press F5, then the script will continue running until the end. On the other hand, there is no use continuing to run the script, since the result will still be incorrect. Also, in some scripts, it may take a long time to finish. Therefore, we should just stop running immediately, so we can fix the problem.

- Do the Stop Debugging command (Debug->Stop Debug) to stop the script.

The script will stop running and the Examine Dataspace window will disappear.

- Change the * to a + on line 33 to fix the problem.
- Run the script (without the debugger) by pressing the run button.

This time you should see the correct answer.

Figure 6-3 ElmStudio Window with script running in debug mode

```

1
2   Set gvVar1 = 999;
3   Set gvVar2 = 'My String';
4
5   If gvVar1 > 500
6     Write Console 'Greater than 500';
7   Else
8     Write Console 'Less than or equal to 500';
9   EndIf
10
11  Set gvCount = 0;
12  Set gvSum = 0;
13
14  Loop InitVal(1) Incr(1) LoopVar(gvIdx) While(gvIdx<=100)
15    Set gvCount = gvCount + 1;
16    Set gvSum = gvSum + gvCount;
17  EndLoop
18
19  Set gvVar3 = TestFunc{gvCount, 300} + 20;
20  Set gvVar4 = TestFunc{gvSum, 300} + 50;
21
22  Loop InitVal(1) Incr(1) LoopVar(gvIdx) While(gvIdx<=3)
23    Write Console 'In Loop Idx-' + gvIdx;
24  EndLoop
25
26  If ActiveDoc
27    Set gvPgfcCount=0;
28    Set gvPgfcEmptyCount=0;
29    Set gvPgfcVar=ActiveDoc.FirstPgfcInDoc;
30    Loop While(gvPgfcVar)
31      Set gvPgfcCount=gvPgfcCount+1;
32      If gvPgfcVar.Text.Count<1
33        Set gvPgfcEmptyCount = gvPgfcEmptyCount * 1;
34      EndIf
35      Set gvPgfcVar = gvPgfcVar.NextPgfcInDoc;
36    EndLoop
37    MsgBox 'Total Paragraphs-' + gvPgfcCount +
38          ' Empty Paragraphs-' + gvPgfcEmptyCount;
39  EndIf
40
41

```

line=30 col=7 INS (CR+LF)