Script Debugger User's Guide

VERSION 3.0

FOR APPLE POWER MACINTOSH

Script Debugger Version 3.0

Mark Alldritt and Late Night Software Ltd.
333 Moss Street
Victoria, B.C.
CANADA V8V 4M9

Technical Support

For technical support, contact us on the web, by email, fax or telephone:
Web: http://www.latenightsw.com
email: support@latenightsw.com
fax: (250) 383-3204
tel: (250) 380-1725
You must be a registered user to receive technical support.

In Memoriam

Dr. Stephen PY Lee

Credits

Software Design & Development: Mark Alldritt
Contract Software Development: Rod Moorhead, Adrian C Ruigrok
Documentation: Gerry Tubin, Judith McDowell
Packaging: Gerry Tubin, Alisa VanderKolk
Graphic Design: Bernard Michaleski
Sales & Marketing: Gerry Tubin

Our appreciation to the following for their invaluable help:

Matt Neuburg for his genius
Jon Pugh for his continued contribution to the design and usability of Script Debugger
Stephan Somogyi for finding all the tough bugs

Contents

CHAPTER 1

Introduction

Script Debugger is a replacement for Apple's Script Editor. With it, you can create and edit scripts, but it has many additional features that you will find invaluable if you are a script developer.

Script Debugger provides you with a development environment for AppleScript. Script Debugger has advanced editing features, including support for scripts larger than 32K, and drag-and-drop editing. It also provides you with a complete debugging environment. Script Debugger allows you to single-step through your scripts, and to examine the contents of variables while your script is executing. Script Debugger also provides tools to explore the scripting capabilities of other applications.

Script Debugger is scriptable, recordable, and attachable. You can use AppleScript to customize Script Debugger by adding scripts to its menu, and by attaching scripts to its other menu items. This allows you to customize and extend Script Debugger for your work habits and needs.

We have engineered Script Debugger to be the best scripting tool available. We are confident that it will meet and exceed your expectations.

About This Manual

This manual explains how to install and use the Script Debugger software.

Chapter 1, *Introduction*, gets you started with Script Debugger. It tells you what you need in order to begin using the software, and walks you through the installation process.

Chapter 2, *Creating and Editing Scripts*, introduces you to each of Script Debugger's editing features. It shows you how to get the most out of those features by walking you through creating and editing a sample script.

Chapter 3, *Debugging Scripts*, describes Script Debugger's debugging capabilities. It steps you through debugging some example scripts.

Chapter 4, *Reference*, enumerates Script Debugger's windows and menus.

Appendix A, *Issues With Scriptable Applications*, lists some technical considerations having mostly to do with Script Debugger's Dictionary Explorer feature.

Appendix B, *Projector Support*, explains Script Debugger's support for the MPW (Macintosh Programmer's Workshop) Projector source code control system.

Appendix C, *AppleScript Information*, points you to further information about the AppleScript scripting language.

Appendix D, *Scripting Script Debugger*, describes how Script Debugger itself is recordable, scriptable, and attachable.

This manual assumes that you are already familiar with the Macintosh desktop and have basic Macintosh skills such as using the mouse and working in the Finder. It assumes also that you have had some exposure to AppleScript.

On-screen Help

Script Debugger includes on-screen information that you can consult when you need help. Balloon Help is a feature of Mac OS that explains the function or significance of items you see on the screen. To turn on Balloon Help, choose Show Balloons from the Help menu. When you point to an item on the screen, a balloon with explanatory text appears next to the item. To turn off Balloon Help, choose Hide Balloons from the Help menu.

(Balloon Help, and the Help menu, are absent under Mac OS X.)

System Requirements

To use Script Debugger 3.0, you will need the following hardware and software:

- a PowerPC Macintosh (for best performance, Late Night Software recommends a PowerPC G3 or greater).
- Mac OS version 8.1 or later. For best performance, Late Night Software recommends Mac OS 8.6 or later. If you have Mac OS X, Script Debugger runs natively.
- a hard drive with at least 15 MB free space.
- at least 32 MB of RAM (random-access memory), with 5 MB allocated to Script Debugger. You may need to allocate 10 MB to Script Debugger when working on large scripting projects. Under Mac OS X, you should have at least 64 MB of RAM (128 MB recommended).
- a CD-ROM drive, if you obtained Script Debugger on CD-ROM.

Installing Script Debugger

Follow the steps in this section to install the Script Debugger software on your computer.

1. Insert the Script Debugger CD-ROM if you have one.

Skip this step if you obtained Script Debugger electronically.

2. Run the installer.

Install the Script Debugger 3.0 software by opening the Script Debugger 3.0 Installer application. You will be greeted by an introductory dialog box. Press the Continue button to proceed.

3. Read the Script Debugger software license agreement.

Please read this document carefully, because it explains the terms under which Late Night Software Ltd. is licensing the Script Debugger Software to you. If you decline to accept this agreement, press Decline and return the software to the location where you purchased it. Click the Accept button to indicate your acceptance of the Script Debugger 3.0 Software License Agreement.

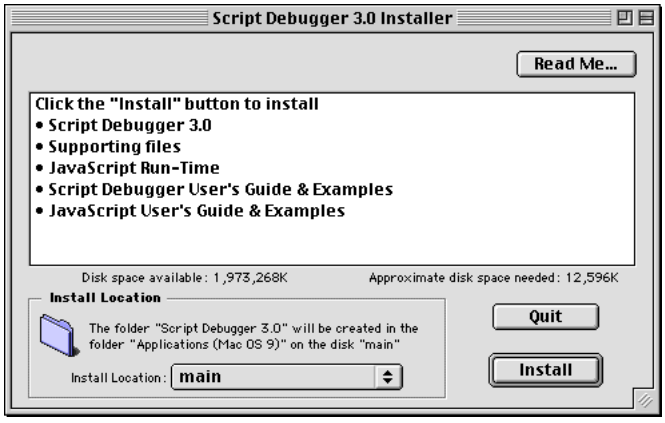
4. Review the Read Me file.

The Read Me file explains issues that may have come to light after this User's Guide was written. Press the Continue button to proceed with the installation.

5. Choose an installation location and complete the software install process.

By default, the installer will offer to create a Script Debugger 3.0 folder in the root folder of your Macintosh's startup volume; under Mac OS 9 or Mac OS X, this will be in the computer's Applications folder. You can use the Install Location popup menu to choose another volume or another folder (Figure 1-1). You can also accept the default location and then move the Script Debugger 3.0 folder at any later time.

Figure 1-1
Script Debugger Installer Dialog



When you have selected an install location, press the Install button to complete the software installation. Under Mac OS 8/9, you'll need to restart the computer before using Script Debugger.

What's Installed Where

Script Debugger is almost entirely self-contained, with the Script Debugger application and supporting files going into a single folder. However, Script Debugger's debugging capabilities depend upon the presence of its own scripting language. On Mac OS X, this is enabled through the file **Script Debugger.component**, which should be in the Library/Components directory. On earlier systems, the file **Script Debugger Nub** should appear in the Extensions folder.

Also, Script Debugger is accompanied by another scripting language, JavaScript. This is installed partly for your enjoyment (JavaScript can be a great scripting language), but also because some of Script Debugger's utility scripts depend upon it. To enable the JavaScript scripting language, **JavaScript.component** should go into the Library/Components directory on Mac OS X, or the **JavaScript** extension should go into the Extensions folder on earlier systems.

By default, the installer will simply "do the right thing." If you're installing under Mac OS X, ScriptDebugger.component and JavaScript.component are installed in the Library/Components folder of the startup drive, and Script Debugger Nub and JavaScript are installed in the Extensions folder of the corresponding Classic system folder. If you're installing under an earlier system, Script Debugger Nub and JavaScript are installed in the Extensions folder of the startup drive.

However, it may be that you have specialized needs or a more complicated partition setup, where you eventually want to run Script Debugger under a system that the installer doesn't "see" at the time of installation. To make this possible, you can start up under that system and run the installer again, but this has the downside that it installs the whole Script Debugger folder a second time. A simpler solution is just to put the two components or extensions into the right place yourself. You'll find them in the Tools & Goodies folder in the Script Debugger folder. Note that under Mac OS X, you may have to create the Components folder in the Library folder first; then use the Finder to copy the component files into place (don't use the Terminal's `cp` command, which will corrupt the files). Under Mac OS 8/9, you'll have to restart the computer in order to use Script Debugger under the system where you just installed the extensions.

Observe that you do not need more than one copy of Script Debugger, regardless of how many partitions and systems you have. If you start up under Mac OS X, Script Debugger runs as a native Carbon application. If you start up under Mac OS 8/9, Script Debugger runs as an ordinary PowerPC application. (You cannot run Script Debugger under Mac OS X as an ordinary PowerPC "Classic" application.)

Registering Your Copy of Script Debugger

In order to provide you with service, we need to know who you are.

If you received Script Debugger on CD-ROM, please take the time now to complete and mail the product registration card and fill out the short questionnaire attached to it. Thank you in advance for your prompt response.

You can also register your copy of Script Debugger online. Simply double-click the Register OnLine URL clipping file to go to our online registration Web page (http://www.latenightsw.com/sd3.0/registration.html) in your Web browser.

You may also want to visit the Late Night Software Web site (http://www.latenightsw.com/) from time to time for news regarding updates and upgrades to Script Debugger 3.0, technical support, and news of other Late Night Software products.

What's New in Script Debugger 3.0

A number of the new features that appear in Script Debugger 3.0 were drawn from the extensive wish list that we received from our users following the release of version 2.0, winner of the 2000 Macworld Eddy for Best Development Software. Feedback from users and the release of Mac OS X played prominently in the evolution of Script Debugger, which is now more streamlined, easier to use, and compatible with Mac OS X. Script Debugger 3.0 follows in the tradition of its predecessors — a powerful yet flexible tool that lets you write both simple and complex scripts with speed and ease.

Carbon!

Script Debugger is delivered in two forms: Carbon and non-Carbon. The Carbon version is intended for use on Mac OS X systems only. The non-Carbon version is intended for Mac OS 8 and 9 systems.

Aqua appearance

In keeping with the new look of the Mac OS X, Script Debugger fully supports the Aqua appearance. This includes honoring Aqua's redesigned menu layout, controls, and backgrounds. Script Debugger also supports live window resizing, live feedback, and sheets.

Trace command

It is now possible to trace (continuous stepping) through a script.

Expressions window

The Expressions window lets you enter expressions that are evaluated against the currently running script. Expressions are re-evaluated each time the script pauses or when execution stops.

Console palette

The Console palette allows you to execute script fragments against the current script. It's a perfect tool for invoking handlers, modifying open scripts, or experimenting with other applications.

Stand-alone OSA component

The AppleScript debugger is now a stand-alone OSA component (the Script Debugger Nub file). This means that scripts running outside Script Debugger can execute even when Script Debugger is not running. Under Mac OS X, use of a stand-alone OSA component is the only way to offer AppleScript debugging to other applications.

Break on exceptions

It's now possible to have the debugger pause whenever an exception occurs (when an `on error` block is entered).

Flatten script

Scripts that use Script Debugger's libraries feature can now be flattened to make them compatible with other script editors. Flattening a script causes the source code from all libraries to be merged with the main script.

Revised Script Error dialog

The Script Error dialog now displays additional information about run-time and compile-time script errors.

Code coverage indication

The AppleScript debugger now indicates which portions of a script have been executed. This new feature allows you to see if your testing and debugging is executing all of your script's code.

Mac OS X scripting additions

The Scripting Additions dictionary window displays scripting additions installed in all four Mac OS X Scripting Additions folders.

Mac OS X–native applets & droplets

Script Debugger can now create Mac OS X Native (i.e. Carbon) applets and droplets in addition to Classic Mac OS applets and droplets.

JavaScript becomes part of Script Debugger installation

JavaScript is now included as part of the standard Script Debugger installation. A number of supporting Script Debugger scripts will be ported to JavaScript as we proceed to make JavaScript a more integral feature of the product.

JavaScript dictionary syntax support

The Dictionary windows can now display commands in JavaScript OSA–compatible syntax.

HTML script export

It is now possible to export script source as formatted HTML for inclusion in Web pages or in HTML email.

Integrated script rescue

The functionality of the Script Rescue utility is now fully integrated into Script Debugger.

Clippings menu

A new  menu has been added to mirror the contents of the Clippings palette in the same way that the  menu mirrors the Scripts palette.

User-settable Command keys

Script Debugger now provides you with full control over Command key assignment, both for build-in commands and for all scripts and clippings. It now also supports a much wider range of modifiers and Command keys, including Function keys.

Command keys in Find dialog

Script Debugger now offers Command key access to all controls in the Find-and-replace dialog box.

Improved text editor

Script Debugger now supports all of BBEdit's cursor and delete key operations and modifiers.

BBEdit integration

Script Debugger now provides an Edit With BBEdit command that allows you to quickly transfer your script between Script Debugger and BBEdit for complex editing tasks.

Scripting improvements

New script property and script handler classes provide information about properties and handlers defined in scripts. Dictionary objects provide full access to dictionary information (suites, events, classes and contents).

Usability improvements

A number of subtle and not so subtle improvements have been made to usability within the application, including new tooltips, improved menu structures, etc.

What Was New in Script Debugger 2.0

Script Debugger 2.0 included many new features that improved creativity, productivity, and integration. There were more then 1,000 individual changes in all. Yet, Script Debugger 2.0 retained much of the feel, look, and functionality that endeared the original product to many script writers.

Here, at a glance, are the most significant new features that appeared in Script Debugger 2.0:

Split-pane editing

Enhances your ability to work with long scripts by allowing you to view several sections of your script at the same time.

Optional line wrapping

You can now turn on line wrapping when editing scripts containing long lines and see the entire line at once without having to scroll left and right.

Support for all OSA languages

You can now use Script Debugger to edit scripts written in any OSA language, not just AppleScript.

Improved find and replace

Find and replace has many new options, including the ability to search backwards, to choose whether or not to wrap, and to search for whole words. There are also new keyboard shortcuts that allow you to search for the currently selected text, and so forth.

Integrated library management

You can now link script libraries to your script and Script Debugger will automatically merge the contents of the library with your script at compile time. It is much simpler to modularize long and complex scripts using this feature.

Text wrap scripts

Script Debugger 2.0 builds on its scriptability to offer a broad range of text wrapping scripts. These scripts quickly create commonly used AppleScript constructs such as `if` and `try` blocks.

Scripts palette

The Scripts palette provides quick access to the scripts that once existed only in the Scripts menu (known as the Extensions menu in Script Debugger 1.0). Double-clicking a script in the palette invokes the script on the current selection in the script window.

Windows palette

Script Debugger 2.0's design reduces the number of windows you have to work with, although you may still find yourself with a large number of open windows. The Windows palette allows you to quickly get to the window you need on a cluttered or small screen.

Applications palette

The Applications palette gives you instant access to the dictionaries of the applications you commonly use. It also provides tools for building `tell` blocks.

Clippings palette

The Clippings palette provides access to templates for commonly used AppleScript constructs. Simply double-click an item to build elements such as an AppleScript `repeat` loop or handler declaration. The Clippings palette is customizable, allowing you to add your own clippings.

Tell Context palette

The Tell Context palette lets you see the elements and properties within the target application of a `tell` block. The palette also allows quick access to the dictionary of the target application.

View local variables

Script Debugger 2.0's AppleScript debugger now allows you to view local variables as your script executes. Local variables occur in handlers, `repeat` loops and `on error` blocks. This is a dramatic improvement over Script Debugger 1.0 which could only display global variables and properties while debugging.

Auto-discovery of variables

When debugging, Script Debugger automatically discovers all variables and displays them. This is in contrast to Script Debugger 1.0 and all other scripting tools that force you to enter the names of all variables manually during debugging.

Watchpoints

Script Debugger 2.0 allows you to set watchpoints on variables. When a script changes the value of a variable with a watchpoint, execution pauses. This allows you to track where your script changes a certain variable.

Tooltips

Script Debugger 2.0 makes extensive use of tooltips to help you understand your script and application dictionaries. In the code window, tooltips display the result of expression evaluations. In dictionary windows, tooltips display descriptions of properties and elements.

Improved execution control while debugging

Script Debugger 2.0 introduces the capability of stepping over handler calls, stepping into handlers, and stepping out of a handler. These new facilities offer much greater control over script execution during debugging.

External debugging

You can use Script Debugger 2.0's AppleScript debugger outside the Script Debugger 2.0 application. This allows you to debug applets, droplets, folder actions, CGIs, and other kinds of scripts. By doing so, you can debug these scripts in real-world settings, instead of having to simulate them within the Script Debugger application.

Improved Scripting Additions dictionary browser

The Scripting Additions dictionary window now allows you to expand and collapse suites. It also allows you to view commands grouped by the Scripting Addition that provides them.

Improved dictionary browser

The application dictionary window has been redesigned to provide easier access to application information. It is now divided into three panels: Dictionary, Object Model, and Explorer.

- The Dictionary panel provides dictionary information in a format similar to that offered by Apple's Script Editor.
- The Object Model panel provides Script Debugger 1.0's object hierarchy display.
- The Explorer panel provides a new view of an application's dictionary so that you can see the actual value of properties and elements and the count of elements in collections.

New Explorer dictionary view

The application dictionary window offers a new Explorer view where you can see the actual values of the application's properties and elements. This view also allows you to experiment with an application by editing property values in place, without having to write AppleScript code. Finally, you can drag property references, element references, or values, directly from the Explorer into your script to create fully formed and correct object specifications and values.

Apple event dictionary syntax

All dictionary windows allow you to view event and class descriptions in two ways: AppleScript syntax, where items are described as they might appear in a script, and Apple event syntax, where items are described in terms of their internal 4- and 8-character codes.

Expanded customization

Script Debugger 2.0 is vastly more customizable than version 1.0. In addition to having a range of new preferences options, Script Debugger 2.0 is more scriptable and more attachable. You can also extend Script Debugger's Templates menu, menu and palette, Clippings palette, and Applications palette.

Note to Script Debugger 1.0 Users

Script Debugger 2.0 reassigned some Command-key shortcuts from those in Script Debugger 1.0. By default, Script Debugger 3.0 uses the new shortcuts (although you can change them using the Set Menu Keys command).

- The old Step command shortcut, Command-], is now mapped to the Indent Selection command (Editing Tools submenu of the menu). The shortcut for the Step command is now Command-Y.
- In Script Debugger 2.0, the Step command steps over handler calls instead of stepping into handlers as it did in Script Debugger 1.0. To step into a handler in Script Debugger 2.0, use the Step Into command (Command-I). For more details, please see Chapter 3, *Debugging Scripts*.

CHAPTER 2

Creating and Editing Scripts

Even before you do any debugging, Script Debugger provides a superb environment for the development of your scripts, with many features that ease the tasks of entering, navigating, and editing code, and of communicating with target applications. This chapter introduces you to these features and to the Script Debugger environment in general.

Getting Acquainted

If you've done any scripting with Apple's Script Editor, you will be familiar with Script Debugger's basic features. Script Debugger has a code window in which to enter your script, and a results window that shows the value returned from your script when you run it. Let's discover these features by developing a simple script. We'll see that it's easy to work in Script Debugger immediately. We'll also find that Script Debugger has many higher-level abilities that we can draw upon as needed.

The Edit-Run Cycle

Our first script won't involve communication with any other applications. Its purpose will be simply to let us explore Script Debugger's code window and editing features. We'll talk about communication with other applications and Script Debugger's dictionary window later in this chapter.

As you know, AppleScript has a List datatype and some properties and commands for manipulating lists. However, it has no native command for removing an item from a list. Let's write one.

1. Start up Script Debugger.

A new code window appears, called Untitled 1. This is where we will develop and test our code. You may also see various supplementary palettes (floating windows). We will talk about them later on, but for now, leave only Untitled 1 on the screen.

Let's develop the script in stages. Our strategy will be: first, to obtain everything in the list before the given index; then, to obtain everything in the list after the given index. All we'll have to do after that is to put those two results together.

We'll start by obtaining everything in the list before the given index. As sample data, we will need a list and the index number of the item to remove.

2. Enter the following into the code window:

```
set L to {"Manny", "Moe", "Jack"}
set whatItem to 2
-- obtain everything before the given item
items 1 to whatItem of L
```

(There's already a bug in this code — two, actually. I did that on purpose. We'll discover and fix them soon enough!)

As you typed this code, did you notice what happened when you typed the right curly brace at the end of the first line? The left curly brace at the start of the literal list was momentarily highlighted. This is Script Debugger's delimiter-balancing feature. If you had typed the right curly brace without the left curly brace being present, Script Debugger would have beeped.

You can also test at any time to see whether delimiters are properly balanced. To try this feature, place the insertion point anywhere inside the literal list in the first line, and choose Balance from the Edit menu. The entire literal list is selected. If the curly braces had not balanced, Script Debugger would have beeped.

TIP: *The Balance command also knows about handlers, conditions, and loops.*

3. To compile the code, hit the Enter key. Edit the code as necessary until it compiles.

Alternatively, choose Compile from the Script menu. Explicit compiling isn't actually necessary; if your code needs compiling, Script Debugger will automatically compile it when you ask to run it.

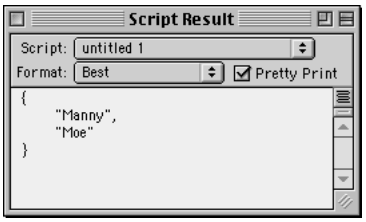
The first time we compile this script, a dialog appears reporting a bug. We can't use the word "to" the way we have done, so the code didn't compile. Notice that the code window shows clearly where the error is: a red arrow points to the problematic line. The fix is to change "to" to "thru". Now the code will compile.

TIP: *Instinctively, you'll probably dismiss the error dialog, stare at the problematic line, and try to figure out what's wrong with it. But now you can't remember what the error dialog said! Solution: just click on the red arrow, or choose Show Last Error from the Script menu, to see the error dialog again. Plus, when the error dialog is showing, you can copy the error message; then you can dismiss the dialog, paste the error message into your code, and contemplate it.*

4. To run the code, hit Command-R. Edit the code as necessary until it gives the right result.

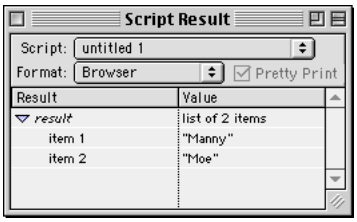
Alternatively, choose Run from the Script menu. The first time we run the code, the Script Result window appears, displaying the value of the last statement of the script. The Script Result window deliberately doesn't come to the front, however, so as not to interfere with your view of the script. To bring it to the front, choose Script Result from the Window menu. Notice that the layout of the result is extremely clear; you're looking at a list whose items are indented from the curly braces and appear on individual lines. This is Script Debugger's "pretty-print" feature.

Figure 2-1
A Pretty-Printed Result



The Script Result window is capable of displaying data in several other formats. Try switching it to Browser format. Here, the items are displayed as rows of a listbox. If you like, double-click the "item 1" line. Another browser window opens, showing that item. All of these windows can be scrolled and resized. You can see that no matter how large or complex a piece of data is, you'll never have any difficulty getting a clear look at it.

Figure 2-2
A Result in Browser Format



Most script development, short of actual debugging (discussed in the next chapter), is a variation on this simple edit-run cycle. There is one typical last step.

5. To save the code, hit Command-S.

Alternatively, choose Save from the File menu. Once the script has been saved, you can quit Script Debugger and then open and work on the script again later. You'll probably want to give the saved script a more helpful, less generic name than "Untitled 1".

Navigation Shortcuts and Utility Scripts

Our code still has a bug: the result is wrong. "Moe" is item 2, and should have been eliminated from the list; but it hasn't been. We try changing the last line of our code to this:

```
items 1 thru whatItem - 1 of L
```

But now the code doesn't compile. Again, Script Debugger shows clearly where the problem is by highlighting the phrase "1 of L." Evidently, we need parentheses around "whatItem - 1". You could just type these parentheses manually, but let's take this opportunity to explore more of Script Debugger's editing environment.

With "1 of L" still highlighted, do this:

1. Type Option-Left-Arrow twice to position the insertion point before "whatItem". Then type Shift-Option-Right-Arrow three times to select "whatItem - 1".

Script Debugger has many keyboard shortcuts for navigating and selecting code. A list of these appears later in this chapter.

2. From the menu, in the Editing Tools submenu, choose Make Sub-Expression.

Script Debugger is itself scriptable and comes with many utility scripts to accomplish common editing tasks. This one surrounds the current selection with parentheses. If you'd like to examine the script, hold down the Option key while choosing Make Sub-Expression. We'll talk more about Script Debugger's utility scripts later in this chapter.

Clippings

We should check to see whether the script works with any value of `whatItem`. Try changing the second line so that `whatItem` is `1` and running the script:

```
set L to {"Manny", "Moe", "Jack"}
set whatItem to 1 -- this is the changed line
-- obtain everything before the given item
items 1 thru (whatItem - 1) of L
```

If you're using a recent version of AppleScript, this won't work, because AppleScript balks at the notion of item `0`. To correct this, we will have to treat separately the special case where `whatItem` is the first item of the list. Rather than typing an if-then-else construct ourselves, we'll let Script Debugger do it.

1. From the menu, choose "if then else".

Script Debugger inserts the template of an if-then-else construct into our script. Script debugger comes with many text clippings for commonly needed constructs. Similarly, in the second part of the script, we'll need another if-then-else construct to check for the last item.

Another way to insert an if-then-else construct is to choose Wrap With If-Then-Else from the Editing Tools submenu of the menu. This has the advantage that whatever is selected becomes the content of the "if" construct. Eventually, you should spend some time experimenting with the various clippings and Editing Tools scripts. You can also write your own.

Expressions

We are closing in on a final version of our script. Let's say it now looks something like this:

```
set L to {"Manny", "Moe", "Jack"}
set whatItem to 2 -- or whatever
-- obtain everything before the given item
if whatItem = 1 then
   set firstPart to {}
else
   set firstPart to items 1 thru (whatItem - 1) of L
end if
-- obtain everything after the given item
if whatItem = length of L then
```

```
   set secondPart to {}
else
   set secondPart to items (whatItem + 1) thru (length of L) of L
end if
firstPart & secondPart
```

You can verify for yourself that this gives the right result when `whatItem` is 1, 2 or 3.

After running the script, you might like to observe an interesting feature of Script Debugger: if you select any expression in the script and then hover the mouse over it, Script Debugger evaluates the expression and displays the result in a tooltip that pops up. For example:

1. Select "firstPart" in the last line and hover the mouse over it.
2. Select the entire expression "firstPart & secondPart" in the last line and hover the mouse over it.

This feature can be of great value in the course of developing a script.

More About the Editing Environment

This concludes our initial tour of Script Debugger's code-editing environment. Now let's go back and take a closer look at some features we briefly encountered during the tour.

Controls

Instead of typing Enter to compile a script and Command-R to run it, you can click the Compile and Run buttons. These are part of the Controls palette. To see the palette if it isn't visible, choose Show Controls in the Palettes submenu of the Window menu. You can toggle the Controls palette's orientation between horizontal and vertical by means of its zoom button.

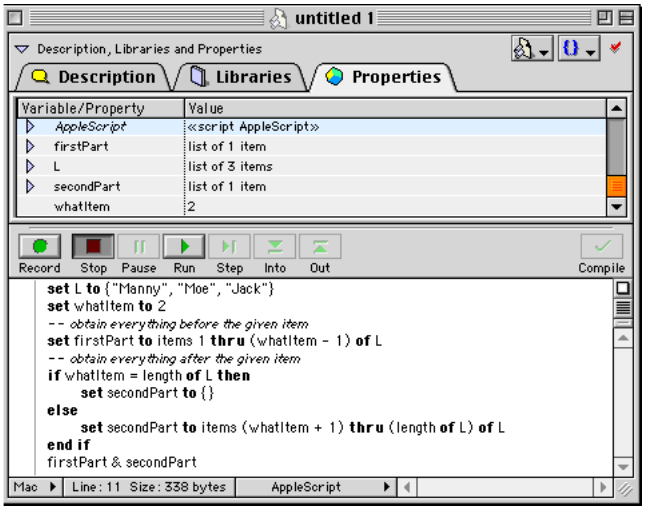
Figure 2-3
The Controls Palette

Additionally, you can make the Controls palette appear as part of each code window, or not; to do so, click the top icon above the vertical scrollbar in a code window.

Code Window Features

You can learn more about the different parts of the code window by turning on Balloon Help, in the Help menu, and hovering the mouse over the part you're interested in. (This doesn't work currently on Mac OS X.) Here is a summary.

Figure 2-4
A Code Window



At the upper left of the code window is a disclosure triangle labelled “Description, Libraries, and Properties.” Click the triangle to reveal or hide the Description, Libraries, and Properties tabs. (Alternatively, drag downwards on the horizontal line in the upper middle of the window.)

- You may type a description of your script into the Description tab. You can style the text of this description. If you save the script as a script application designated to show a startup screen, that startup screen will contain this description in the styles you have set. Also, if a script has a description, you can view its text as you are about to open the script from within Script Debugger by pressing the Show Preview button in the Open dialog. This is useful for finding a desired script.
- Libraries are discussed later in this chapter (“Libraries” on page 30).
- The Properties tab displays the value of script properties and global variables. When you are debugging, it changes to the Debugging tab. This will be discussed further in the next chapter, *Debugging Scripts*. Alternatively, you can see the Properties tab as a separate window by choosing Properties & Globals from the Window menu; however, this version of the window does not display AppleScript’s own properties and globals.

Note that you can adjust the dimensions of the various window elements. For example, if you’re looking at the Properties tab, you can move the vertical divider between the Variable/Property column and the Value column, and you can move the horizontal divider between the Properties tab and the code.

At the top right is a popup menu (its icon is a pair of curly braces) for navigating among markers and handlers within your script. A marker is a comment where the first non-space characters are >>, like this:

```
-- >> This is a marker
(* >> This is another marker *)
```

A handler is a construct beginning with the keyword on.

To the left of the handler popup menu is a popup menu for determining the nature of the saved script. You can save it as text, as a compiled script, or as a script application. A compiled script can consist of just a data fork; this is a Unix-style file format suitable for Mac OS X only. A script application can be a Classic application or a Mac OS X application. If the script is to be saved as a script application, two buttons appear for determining whether the script application should stay open after its Run handler executes and whether it should show its startup screen. The same choices are available from the Save As dialog.

At the top of the vertical scrollbar is the button that toggles whether the controls appear in the code window. Below that is a button that determines whether code should wrap automatically or scroll off the right end of the window. In the latter case, the horizontal scrollbar is enabled.

Beneath the wrap button is a slider that you can drag to split the code region horizontally into two panes. This allows you to view two regions of your script’s code simultaneously. You can drag the slider again to create a third pane, and so forth. (If there’s no room to create an additional pane, the slider will beep when you click it.) You can resize the panes by sliding the arrow icon that appears above the divider line. Double-click an arrow icon to eliminate its pane.

In the lower left of the window is a popup menu letting you select how end-of-line characters should be stored when saving the script as text only. To the right of this is a display of the script’s size and the line number in which the insertion point is positioned. Click it to bring up the Go To Line dialog (alternatively, type Command-J or choose Go To Line from the Search menu).

To the right of this is the OSA Language popup menu. You use this to designate another OSA scripting language as the language of this script, just as in Apple's Script Editor. Notice that with Script Debugger you've got at least two more OSA scripting languages besides AppleScript! Late Night's implementation of JavaScript as a scripting language is discussed in a separate document. The other language, AppleScript Debugger, is the subject of Chapter 3, *Debugging Scripts*.

Navigation and Selection

Here is a summary of Script Debugger's keyboard shortcuts for navigating within a code window.

- Arrow keys move one character at a time.
- Option-Left and Option-Right move one word at a time.
- Command-Left and Command-Right move to the start and end of the line.
- Option-Up and Option-Down move one screenfull at a time.
- Command-Up and Command-Down move to the start and end of the script.
- Command-Option-Up and Command-Option-Down move one handler at a time. Alternatively, choose Go To Previous Handler and Go To Next Handler from the Search menu, or use the handler popup at the top right of the code window.
- To jump to a particular line, choose Go To Line from the Search menu, or click on the line number display at the bottom left of the window.

Here is a summary of ways you can select text.

- Hold Shift while using a keyboard navigation shortcut to extend the selection.
- Double-click to select a word at a time. Triple-click (alternatively, click just to the left of a line) to select a line at a time. To extend the selection by the same unit, Shift-click elsewhere, or drag without releasing from the last click.
- Choose Balance from the Edit menu (good for selecting literal lists, material in parenthesis, loops, conditions, and handlers).

Adding modifier keys to the Delete key lets you delete in chunks corresponding to the navigation shortcuts. Adding Shift turns deleting backwards into deleting forwards. For example, if you have the text "Hey nonny no" and the insertion point is before "nonny", then Option-Delete would delete "Hey", but Shift-Option-Delete would delete "nonny".

External Editor

In general, Script Debugger's keyboard navigation, selection, and deletion shortcuts are modelled after BBEdit. If you would feel more comfortable editing a script in BBEdit itself, you can! To do so, choose Edit With BBEdit from the File menu.

This creates a copy of the script and opens it with BBEdit. When you save the edited script within BBEdit, the changes are automatically saved back into Script Debugger's original version of the script.

Drag-and-Drop

Cutting and pasting works in the usual way; but Script Debugger also makes extensive use of drag-and-drop. You can drag within code windows, between code windows, between a code window and another application that supports drag-and-drop (such as the Finder), from the Properties tab to code, and from most of Script Debugger's other windows, such as the Clippings palette or Script Result window, to a code window.

This is not just a way of moving text, but also a way of typing text. For example:

- to type a global variable or property name, drag the name from the Properties panel into the code.
- to type a file's pathname, drag the file from the Finder into the code window.
- to type a reference to a property of a record, drag that property's listing from the Script Result window or Properties tab into the code.

This list is far from exhaustive. Some other examples appear later in this chapter. Experiment!

Searching for Text

Script Debugger can find (and replace) text within a code window. The find commands have keyboard shortcuts; to see them, look in the Search menu. The commands and shortcuts are more extensive than you might at first suspect; hold the Shift key down and look in the Search menu again to see the full repertoire. A particularly useful shortcut is Command-H, which searches for the currently selected text (backwards, if the Shift key is used).

TIP: *A shortcut for searching for the currently selected text is to Command-Option-double-click some text; this selects a word and instantly searches for the next occurrence of it, in a single step.*

Of the find commands, only Replace All has no keyboard equivalent; it is available only through the Search menu, or as a button in the Find dialog that appears when you choose Find from the Search menu (Command-F). This is a deliberate safety feature, because Replace All is not undoable. Note that Replace All's exact behavior depends upon the find options. These options include things like Start at Top, Wrap Around, and so forth, and are accessed as checkboxes in the Find dialog.

Scripts Menu and Palette

Any folders and compiled scripts located in the Scripts folder, which is in the same folder as Script Debugger, appear in the menu. Alternatively, you can use the Scripts palette. To see it, choose Show Scripts from the Palettes submenu of the Window menu.

Observe that by default, the Scripts palette is part of a three-tab palette that also contains the Windows and Clippings palettes. However, you can separate it by dragging its tab off the palette. If you change your mind, you can reunite palettes by dragging a separated palette's tab into another palette. In general, Script Debugger's palettes let you separate and recombine them however you please.

To run a script, choose it from the menu or double-click it in the Scripts palette. To edit it, Option-choose it from the menu or Option-double-click it in the Scripts menu. These are expected to be scripts that drive Script Debugger; they therefore are run in the context of Script Debugger, which is why they do not contain `tell` blocks specifying Script Debugger as the target.

You are free to place scripts in the Scripts folder as a way of adding custom commands to Script Debugger. The icon with which a script appears in the Scripts palette is the icon that it has in the Finder. The Scripts palette has a popup menu at the upper right that lets you elect whether to show or hide the icons; you can also use this popup menu to have the Finder open the currently selected script's folder. This popup menu is also where you set a script's keyboard shortcut ("Set Keystroke"). A script's tooltip, which appears when you hover the mouse over its entry in the Scripts palette, is its description from the Description tab (see "Code Window Features" above).

Clippings Menu and Palette

Any folders and text clippings located in the Clippings folder, in the same folder as Script Debugger, appear in the menu. Alternatively, you can use the Clippings palette. To see it, choose Show Clippings from the Palettes submenu of the Window menu.

The purpose of text clippings is to act as mini-templates for frequently used code structures. If you choose a clipping from the menu, or double-click a clipping in the Clippings palette, it is pasted into your code at the insertion point. Alternatively, you can drag-and-drop from the Clippings palette into your code. You can set a clipping's keyboard shortcut, using the popup menu at the upper right of the Clippings palette. If

you hover the mouse over a clipping's entry in the Clippings palette, its contents will appear as a tooltip. Option-choosing a menu item from the  menu, or option-double-clicking a clipping in the Clippings palette, opens its containing folder in the Finder. Alternatively, select a clipping in the Clippings palette and choose Reveal In Finder from the popup menu at the upper right of the palette. You can then open the text clipping in the Finder to read it.

To make a new clipping, drag text from a code window into the Clippings folder.

Templates

A template is a complete code window saved as stationery. When a template is opened, its code and all its window-based settings (such as its size, which tab is showing, whether lines wrap, what sort of document the script is to be saved as, and so forth) are copied into a new untitled window. When this new window is edited and saved, it does not overwrite the original template.

To create a template, save a code window as an ordinary compiled script and close the window. Then select the saved script file in the Finder, open its Get Info window, and check the Stationery Pad checkbox. You can store a template anywhere, but if you put it in the Templates folder, in the same folder as Script Debugger, it will appear as a subitem of the New menu item of the File menu. Choosing such a subitem creates a new script as a copy of the template. To edit a template script itself, Option-choose the menu subitem.

If you create a template called Default Script and put it in the same folder as Script Debugger, it will be used every time you choose New Script (Command-N) from the File menu. To edit the Default Script stationery file later on, Option-choose New Script from the File menu.

If you simply want to set the default size for future code windows that don't derive from a template, set up a code window the way you like it and choose Set Default Window Size from the Window menu. Certain other default features of new code windows can be set as preferences (see "New Script Defaults panel" on page 60).

Inter-Application Communications

Script Debugger has many features for helping you write scripts involving inter-application communications, such as the Dictionary window, the Applications palette, and the Tell Target palette. To explore these, we'll write some scripts that involve driving scriptable applications.

Suppose we have a FileMaker Pro database where every record is a piece of music. One of the fields is the composer's last name. We wish to know all the pieces by any composer whose last name begins with "B". Assume that we know virtually nothing about how to speak to FileMaker Pro using AppleScript. Script Debugger can help us find out.

The first step is to open FileMaker's dictionary. Here's the quickest way.

1. Start up FileMaker Pro and open the target database.

You don't have to start up an application in order to open its dictionary; but in Script Debugger, you get a shortcut to open a dictionary if the application is already open, as we'll see. Besides, we're going to need access to the target database from Script Debugger, so we may as well have it ready.

2. In Script Debugger, hit Option-Command-O. In the resulting list of open applications, choose FileMaker Pro.

This is the same as going into the Open Dictionary submenu of the File menu and choosing Running Application. You could have just chosen Application, but then you would have to navigate your hard disk. This way, with the application running, you can get to it very easily.

The Applications Palette and Menu

At this point, before examining the Dictionary window that has just appeared, stop and look at the Applications palette. If it isn't visible, choose Show Applications from the Palettes submenu of the Window menu. Also, look in the Open Dictionary submenu of the File menu. Notice that FileMaker Pro is now listed in both places. This means that from now on, whenever you want to see FileMaker Pro's dictionary, you can do so in one easy step: all you have to do is double-click its entry in the Applications palette, or choose it from the Open Dictionary submenu of the File menu. All applications whose dictionary you ask to see are remembered. To remove an entry, select it in the Applications palette and choose Forget Application from the popup menu in the upper right corner of the palette.

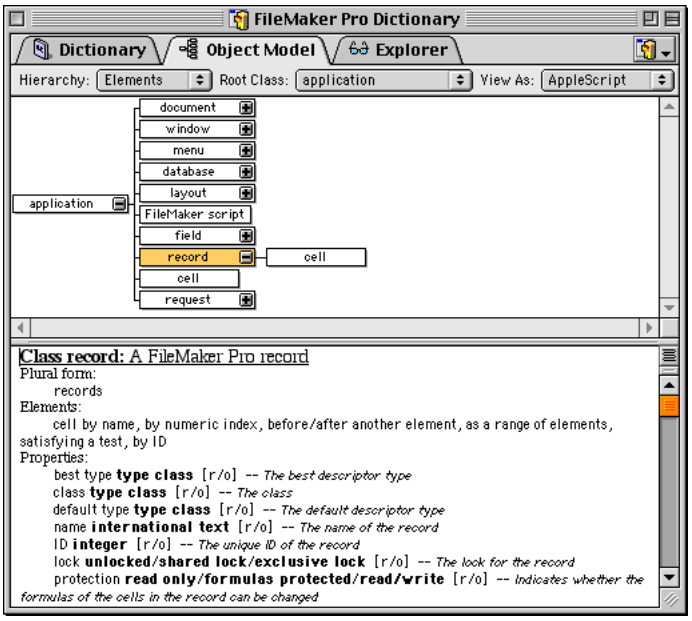
The Dictionary Window

Now let's examine the Dictionary window. The first panel, the Dictionary panel, is much like the Dictionary window in Apple's Script Editor. Classes and commands are listed down the left side. You click one to make the corresponding entry appear on the right side. But this window is considerably more convenient than the Script Editor's. You can widen the left side by dragging the vertical separator between the panes. Instead of seeing both events and classes organized together by suite, you can elect to see only the events or only the classes; use the popup menu at the upper left. Finally, you can switch the view of the right panel from AppleScript to a raw Apple event. This is an advanced feature providing information that you probably won't need if you're just going to write AppleScripts, but which can be extremely useful if you intend to send Apple events in some other programming context such as UserLand Frontier or REALbasic.

The Dictionary window is not giving us much of a clue as to how to query FileMaker Pro for the information we want, so let's switch to the next panel, Object Model. Here we see the object hierarchy displayed graphically. We can click on an object to show its elements, as well as to see its dictionary entry listing both elements and properties.

As Figure 2-5 shows, records are top-level entities. In the lower pane, we discover that every record has a unique ID property. In the upper pane, we see that the feature of each record that we're going to want to inquire about is its cells.

Figure 2-5
The Object Model Panel



Now switch to the last panel, the Explorer. This amazing panel inquires of FileMaker Pro as to its state *at this instant*. Using the dictionary to learn what classes FileMaker Pro supports, it counts the actual number of objects of each class and reports the *actual values* of each one. This is why it was important to open the target database beforehand. We are now looking at that database in just the same way as FileMaker makes it accessible through AppleScript.

WARNING: *Not every scriptable application is as easily studied through the Explorer as FileMaker Pro! In Eudora, the Explorer is less directly useful. In Microsoft Excel or Word, you're liable to crash the computer. See Appendix A, Issues With Scriptable Applications.*

Let's examine what we're seeing (Figure 2-6). Again, we see that records are a top-level entity. At this point, we could click the disclosure triangle to the left of the "records" line to see all the records. However, we also see that there are many records, and Script Debugger would have to inquire about each of them, which could take quite a while in a large database. So instead, let's use the popup menu at the right of the "records" line to get more detail about just one record — say, the first.The "records" line changes to "first record". If we now double-click this line, we get a Browser window describing only the first record (Figure 2-7).

Figure 2-6
The Explorer Panel

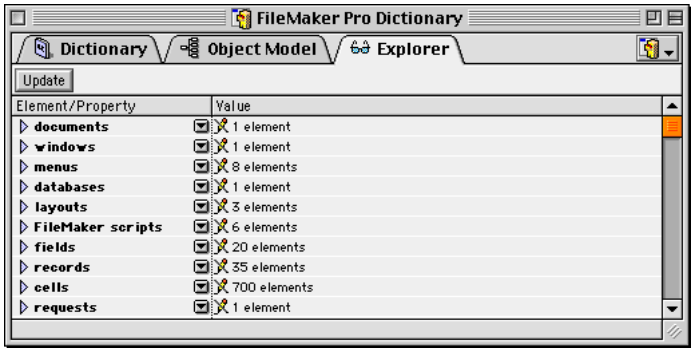
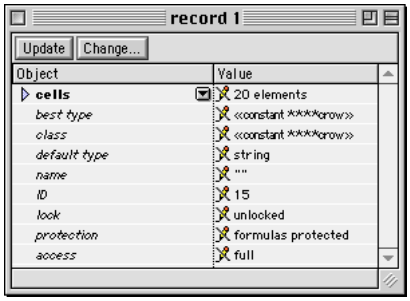


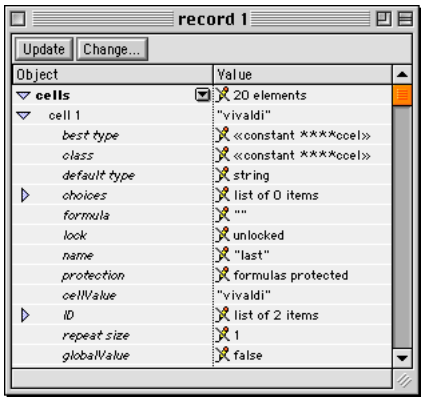
Figure 2-7
A Browser Window



We see that the record does indeed have a unique ID. Since we're going to ask for records depending on something about some cell, we need to know which cell it will be. To find out, we click the disclosure triangle at the left of the "cells" line. With a little further exploration (Figure 2-8), we see clearly that the cell we want is cell 1, identified by its name property, "last", evidently meaning the last name of the composer. This is confirmed by the fact that its value for this record is "vivaldi"; clearly this is the value we're going to be inquiring about.

WARNING: *Observe that in Figure 2-8, certain values are not preceded by a slashed pencil icon. Such values can be changed directly through the Explorer.*

Figure 2-8
A Closer Look at the Same Browser Window



To do so, select a value and hit Enter to make it editable. You're changing the actual data in the actual document, so use this feature with caution!

Developing a Script

We are now ready to construct our query. We will be able to do this almost without doing any typing. Returning to the code window, do this.

1. In the Applications palette, select FileMaker Pro, and press the Paste Tell button.

A `tell` block targeting FileMaker Pro appears in our code window. The selection is ready to type the content of the block.

2. In the Browser window for the first record, select the "ID" line and Copy. Return to the code window and Paste.

Script Debugger pastes a reference to the object copied from the Browser window:

```
ID of record 1
```

But of course, we want the ID of every record, and not every record absolutely, but only those where the cellValue of cell 1 starts with "B".

3. In the Browser window for the first cell, select the "CellValue" line and Copy. Return to the code window and Paste.

We now have nonsense:

```
ID of record 1 cellValue of cell 1 of record 1
```

But, with just a tiny bit of typing, we can transform it into the query we want:

```
ID of every record where cellValue of cell 1 starts with "B"
```

Now we're ready to try it out!

4. Run the Script.

The Script Result window appears, producing a list of the ID numbers of the desired records.

Values and Datatypes

When you're doing inter-application communication, one of Script Debugger's most useful features is its ability to display coherently the nature and format of the values of elements, properties, and results. This is helpful because much of the trick of scripting applications lies in knowing the structure of what you can say and of what the application is saying to you.

Let's look at a couple of examples using the Script Result window. First, we will script Clip2Gif to return the data of a picture.

1. Start up Clip2Gif. In Script Debugger, open its dictionary and make a new code window.
2. In the Finder, locate a picture file and drag it into a Script Debugger code window.

This is a shortcut to give us a reference to the file (an alias) without doing any typing. For example, in my case, I get this:

```
alias "binkie:pix:gatespie2.pct"
```

The reference is still selected so we can continue to construct the script without doing any typing.

3. In the Applications palette, select Clip2Gif and press the Paste Tell button.

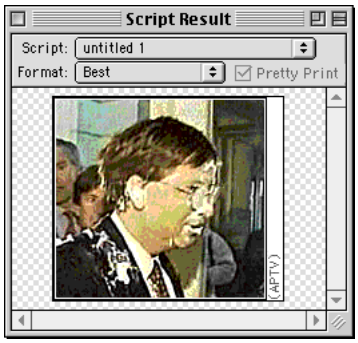
This surrounds the file reference with a `tell` block. Now let's finish the script by hand to get this:

```
tell application "clip2gif"
   open alias "binkie:pix:gatespie2.pct"
   content of window 1
end tell
```

4. Run the script. If necessary, change the Script Result window format to "Best".

The Script Result window, if you select Best format, displays the actual picture as a picture!

Figure 2-9
Script Result Window Displaying a Picture



Similarly, if we ask a styled text application such as QuarkXPress or Tex-Edit for the style of the selection, we get something like this:

```
{
   class:text style info,
   on styles:{
      bold,
      underline,
      condensed
   },
   off styles:{
      italic,
      outline,
      shadow,
      expanded,
      strikethrough
   }
}
```

This shows clearly that what we have received is a text style info record and also what the structure of that record is.

The Explorer window can also be very helpful in this regard. For example, if you look in the Explorer window for QuarkXPress at the top level, you'll see a property "gray TIFF resolution". Next to its value is a popup menu containing the items "use 16 levels" and "use 256 levels". This makes it easy to see that "gray TIFF resolution" is some sort of enumeration and that these are its two possible values.

Scripting Additions

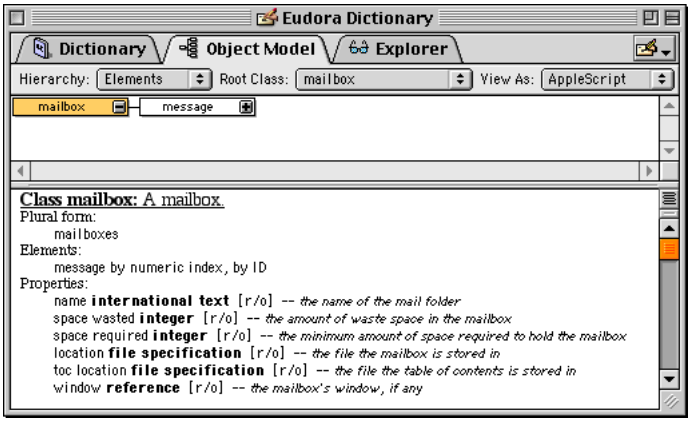
To demonstrate the use of scripting additions, and as another example of inter-application communication, we'll write a script that involves some user interaction.

Here's the plan. First, our script will ask the user to choose a Eudora mailbox. Then, it will tell the user the range of dates of the messages in that mailbox — the earliest message and latest message — and will ask the user for a cutoff date somewhere between them. (The idea is that we might then proceed to archive all the messages that are older than the cutoff date. For example, we might transfer them to some other mailbox. But we won't bother to write that part of the script.)

Our first task is to obtain the name of every Eudora mailbox and present the names as a list from which the user can choose. For purposes of this example, we'll ignore the possibility of mail folders entirely.

Unfortunately, scripting Eudora is more cumbersome than scripting FileMaker Pro, because Eudora doesn't support `whose` or `every`, or even `count`, for most of its objects. For example, we can't learn the name of every mailbox by asking for "the name of every mailbox" or the number of mailboxes by asking for "count mailboxes". This also means that the Explorer panel doesn't work very well, and we have to be more proactive in order to investigate the existing objects. However, in the Dictionary window for Eudora, we see from the Object Model panel (by looking in the Root Class popup) that `mailbox` is a top-level object and that we can ask for a mailbox's name. This suggests a construct in which, since we cannot ask how many mailboxes there are, we loop through the mailboxes gathering their names until we hit an error.

Figure 2-10
Part of Eudora's Object Model




```
tell application "Eudora"
   try
      set L to {}
      repeat with i from 1 to 10000 -- until we error out, actually
         set end of L to name of mailbox i
      end repeat
   on error
      -- presumably, we reached the last mailbox
   end try
   L
```

```
end tell
```

This works, as the Script Result window shows, and we are now ready to present the list to the user. We need to find a scripting addition that can do this.

1. Type Shift-Command-A to bring up the Scripting Additions Dictionary window.

Alternatively, you could choose Scripting Additions from the Open Dictionary submenu of the File menu. The point is that scripting additions have their own dictionary window. There are no Object Model or Explorer panels, and all scripting additions are displayed in a single window. We can view the information arranged in four ways: all events; all classes; by scripting addition; and by suite. Here, viewing the information by suite, the "User Interaction" suite seems most promising; and sure enough, it contains a Choose From List command. This has rather a long involved syntax; but there is no need to memorize it.

2. Flip open the User Interaction heading, select "choose from list" in the left-hand column, and Copy. Switch to the code window and Paste.

In another wonderful time-saving maneuver, Script Debugger captures a template of the command and lets us paste it into our code! Working with this template, we have soon written something like this:

```
tell application "Eudora"
   try
      set L to {}
      repeat with i from 1 to 10000 -- until we error out, actually
         set end of L to name of mailbox i
      end repeat
   on error
      -- presumably, we reached the last mailbox
   end try
end tell
choose from list L ¬
   with prompt "Pick a mailbox:" ¬
   without multiple selections allowed and empty selection allowed
```

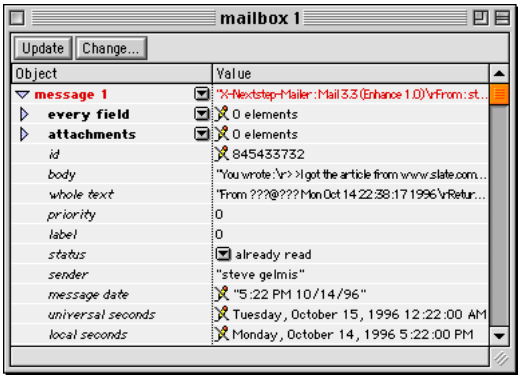
Experimentation shows that this works fine. If the user cancels the dialog, the result is `false`; otherwise, it is a list consisting of one item — the name of the mailbox chosen by the user.

Now we want to cycle through the messages of the chosen mailbox to examine their dates. At this point, we can only think what a pity it is that the Explorer panel doesn't work very well with Eudora. We'd like to study messages and their properties so that we can see which property to use as the date; but messages and mailboxes don't even show up in the Explorer panel. However, all is not lost. It turns out that we can help Script Debugger by handing it a reference that works, and from which it can start a new Explorer window.

1. In the Applications palette, double-click Eudora to open its Dictionary window, and switch to the Explorer panel.
2. Choose New Explorer from the Open Dictionary submenu of the File menu.
3. In the resulting dialog, type "a reference to mailbox 1", and hit OK.

Although Eudora doesn't permit us to count mailboxes, it graciously permits us to count the messages of a mailbox. So once we have started with a specific mailbox, the Explorer works fine. We can see all the messages in this mailbox, and we can open the disclosure triangle for a message to see all its properties.

Figure 2-11
Exploring a Eudora Mailbox and Messages



This shows that a message has a universal seconds property which is a date, properly calibrated against GMT. Since this is the reference that will form the basis for our script, we can save ourselves some typing by selecting the "universal seconds" property and copying and pasting it into our code. The result is this:

```
universal seconds of message 1 of mailbox 1
```

We will alter this and build our code around it. So, to learn the maximum and minimum message dates, we just start with an impossibly large date and an impossibly small date and keep track as we cycle through all the messages of whatever mailbox the user has asked for:

```
if the result is not false then
   tell application "Eudora"
      set m to item 1 of the result
      set msgCount to count messages of mailbox m
      set minDate to current date
      set year of minDate to (year of minDate) + 1 -- next year, hehheh
      set maxDate to date "Friday, January 1, 1904 12:00:00 AM"
      repeat with i from 1 to msgCount
         set d to universal seconds of message i of mailbox m
         if d < minDate then
            set minDate to d
         end if
         if d > maxDate then
            set maxDate to d
         end if
      end repeat
   end tell
   -- next user interaction goes here
end if
```

Now we are ready for our next user interaction. We want to tell the user the range of dates we found and allow the user to supply a date between them as a cut-off point. Examining the Scripting Additions Dictionary, we see that this is a job for Display Dialog. To copy and paste its template into our script is easy, and we then quickly arrive at the following:

```
   display dialog "Messages range from " & minDate & " to " & ¬
      maxDate & ". Your cut-off date?" default answer "" ¬
      buttons {"Cancel", "OK"} ¬
      default button 2
```

For the sake of completeness, we next test to see that the user hit OK and that whatever the user typed is indeed coercible to a date within the given range. It is easy to see what to do, because the Script Result window clearly shows what sort of thing to expect as the result of the previous statement. In fact, we can just drag the "button returned" entry from the Script Result window into the code to get things rolling with the phrase "button returned of the result"! Ultimately, we wind up with something like this:

```
if button returned of the result is "OK" then
   set whatDate to date (text returned of the result)
   -- let ourselves error out if not a date
   if whatDate < minDate or whatDate > maxDate then
      display dialog "Out of range!"
   end if
end if
```

Here, then, is the entire script:

```
tell application "Eudora"
   try
      set L to {}
      repeat with i from 1 to 10000 -- until we error out, actually
         set end of L to name of mailbox i
      end repeat
   on error
      -- presumably, we reached the last mailbox
   end try
end tell
choose from list L ¬
   with prompt "Pick a mailbox:" ¬
   without multiple selections allowed and empty selection allowed
if the result is not false then
   tell application "Eudora"
      set m to item 1 of the result
      set msgCount to count messages of mailbox m
      set minDate to current date
      set year of minDate to (year of minDate) + 1 -- next year, heh heh
      set maxDate to date "Friday, January 1, 1904 12:00:00 AM"
      repeat with i from 1 to msgCount
         set d to universal seconds of message i of mailbox m
         if d < minDate then
            set minDate to d
         end if
         if d > maxDate then
            set maxDate to d
         end if
      end repeat
   end tell
   display dialog "Messages range from " & minDate & " to " & maxDate ¬
      & ¬
      ". Your cut-off date?" default answer ¬
      "" buttons {"Cancel", "OK"} ¬
      default button 2
   if button returned of the result is "OK" then
      set whatDate to date (text returned of the result)
      -- will error out if not a date
```

```
      if whatDate < minDate or whatDate > maxDate then
         display dialog "Out of range!"
      end if
   end if
end if
```

This is enough to demonstrate the use of scripting additions, as well as to show that a task that might have proved daunting or tedious in something like Apple's Script Editor, is a snap with Script Debugger.

The Manifest

The Manifest, summoned by choosing Manifest from the File menu, lists the applications and scripting additions referred to by the script in the frontmost code window, and lets you reveal them in the Finder. This can be useful if you receive a script written by someone else, or if you intend to send a script to someone else or use it on a different machine and want to make sure that the necessary resources will be present. The script must be compilable, so if an application is completely missing, you won't be able to obtain a manifest for the script. But if what's missing is a scripting addition, the Manifest will list any calls to it as an "unknown Apple event." You may be able to use this information to identify the missing scripting addition (and the Manifest will even try to help, if you like, by querying an online database of scripting additions over the Internet).

For example, after compiling the script developed in "Scripting Additions" above, the manifest shows that the script depends on the application Eudora and the scripting addition Standard Additions. That's very useful information; it means that this script doesn't depend on any scripting additions that the user is unlikely to possess (because Standard Additions is part of the default AppleScript installation), so this script will probably run correctly for any user who has Eudora. On the other hand, if the script turned out to use scripting additions that you think another user might not have, you'd know what they are and where to find them, so that you could include a copy of them when distributing the script.

The Console

The Console, summoned by choosing Show Console from the Palettes submenu of the Window menu, is a miniature code window. Its code can't be saved or debugged. The intention is that any code you run here should be pretty simple! You can run the single line the cursor is on, or you can select several lines and run the entire selection, by pushing the Run button in the console window, or by hitting Enter or Command-R. Any result returned is shown within the console itself, and the cursor is positioned at the line after that, ready for you to type and run another line of code. So you can envision the console as a place to hold small conversations in code.

The console's chief purpose is as a way of driving Script Debugger itself, and in particular as a way of talking to the frontmost code window. Script Debugger helps you do this by setting the console's `tell` context to Script Debugger's frontmost code window, automatically.

To see why this is useful, imagine a script containing a number of handlers. Using the console, you can call any one of those handlers, to test it. For example, suppose the frontmost code window contains this script:

```
on announce(what)
   display dialog what & "!"
end announce
on query(what)
   display dialog what & "?"
end query
```

If you were to run this script, nothing would happen, because it has no top-level statements. You could insert some top-level statements, calling the individual handlers and supplying some parameters, as a way of testing the handlers; but thanks to the console, you don't have to. You can just say, directly from the console, something like this:

```
announce("Hello")
```

This works even when the script is in debug mode (as explained in the next chapter); you can step through an individual handler and debug it, by calling it from the console.

You can also work with values from the frontmost window, but you must use the keyword `its` to do so. For example, after running the script developed in "Scripting Additions" above, where we talk to Eudora and to the Standard Additions scripting addition, the local variable `d` ends up with a value, and we can ask for it in the console:

```
its d
```

The Log Window

The Apple Event Log Window, summoned by choosing Apple Event Log from the Window menu, is particularly valuable for tracking the course of an extended interaction with another application or a scripting addition. It intercepts Apple events as they fly out from Script Debugger and records them and the answers. It only does its work when it is actually showing.

For example, here is an excerpted version of what appears in the Log window if it is showing when I run the entire script developed in the "Scripting Additions" section. Notice that replies are preceded by a sort of arrow glyph (`-->`).

```
Script “mailboxer” started
tell application "Eudora"
   get name of mailbox 1
   -->"In"
   get name of mailbox 2
   -->"Out"
   get name of mailbox 3
   -->"Trash"
   get name of mailbox 4
   -->"contacts"
   get name of mailbox 5
   -->"conversant"
[ ... ]
   get name of mailbox 40
   -->"tidbitstalk"
   get name of mailbox 41
end tell
tell current application
   choose from list {
      "In", "Out", "Trash", "contacts", "conversant", ... "tidbitstalk"}
         with prompt "Pick a mailbox:"
         without multiple selections allowed and empty selection allowed
   -->{
         "conversant"
      }
end tell
tell application "Eudora"
   count every message of mailbox "conversant"
   -->6
```

```
   current date
   -->date "Wednesday, May 3, 2000 11:35:47 AM"
   get universal seconds of message 1 of mailbox "conversant"
   -->date "Wednesday, April 5, 2000 10:02:00 PM"
   get universal seconds of message 2 of mailbox "conversant"
[ ... ]
   get universal seconds of message 6 of mailbox "conversant"
   -->date "Tuesday, May 2, 2000 4:39:07 PM"
end tell
tell current application
   display dialog "Messages range from
      Tuesday, March 28, 2000 7:53:55 PM to
      Tuesday, May 2, 2000 4:39:07 PM.
      Your cut-off date?"
      default answer ""
      buttons "Cancel", "OK"}
      default button 2
   -->{
         text returned:"march 29, 2000",
         button returned:"OK"
      }
end tell
Script “mailboxer” finished
```

This sort of thing can be tremendously helpful as a way of proving that we are saying the right things and getting back the right information in our interaction with other applications. As a bonus, the format of the Log window can be set to show raw Apple events, which is useful in advanced situations where we eventually intend to send the Apple events directly (for example, in developing a REALbasic application).

You can also send commands to the Log window using AppleScript. This can be very helpful for posting information outside the debugging context without interrupting the flow of the script with `display dialog`. For example, a script can say this:

```
log x
```

This causes the current value of the variable x to be placed in the Log window, within special delimiters. For example:

```
(*5*)
```

This only works when the Log window is open, and only from within Script Debugger; but there’s no penalty for using the `log` command outside Script Debugger, so you can use it in an ordinary AppleScript script application or compiled script — it just won’t do anything, except when run from within Script Debugger with the Log window open. Also, the `start log` and `stop log` commands allow a script to limit which parts of itself get recorded to the Log window.

Libraries

Commonly used code can be spun off into a library, which is simply a compiled script to which any other script can refer. AppleScript supports libraries through the Load Script command, but this requires that you keep track of your libraries and that you remember to load them. Script Debugger automates the whole process by keeping track of your libraries for you and by letting you attach a library directly to a script, like a kind of pseudo-property.

Creating a Library

To create a library, simply add a compiled script to the Script Libraries folder in the same folder as Script Debugger. For example, let’s go back to our script for removing an element from a list, turn it into a handler, and save it in the Script Libraries folder.

1. Enter the following code into a code window.

```
on remove at whatItem out of whatList
   set outcome to {}
   if whatItem > 1 then
      set outcome to items 1 thru (whatItem - 1) of whatList
   end if
   if whatItem < length of whatList then
      set outcome to outcome & ¬
         items (whatItem + 1) thru (length of whatList) of whatList
   end if
   return outcome
end remove
```

This is simply a more efficient version of our previous code, wrapped up in a handler so that we can specify the list and the item number at runtime.

2. Compile the script and save it into the Script Libraries folder.

You can give it any name you like, but let's say you call it "Remove Lib".

Accessing a Library

To access a library, use the Libraries panel of a code window. For example:

1. In a new code window, enter the following:

```
set L to {"Hey", "Ho", "Hey Nonny No"}
remove at 2 out of L
```

2. Show the code window's Libraries tab. From the "Plus" popup menu at the right, choose the desired library.

In this example, you would choose "Remove Lib". A listing for "Remove Lib" appears in the Libraries tab to remind you that it is attached to this script. Everything in the "Remove Lib" script is now available from your code, so the reference to the `remove` handler will work. To see that this is true:

3. Run the script.

It works!

To detach a library from a code window, select the library's listing in the Libraries tab and hit the Trash button at the right. To study or edit a library, double-click its listing in the Libraries tab. The advantages of this mechanism are that you don't need to load the library explicitly in code, and that you have a common repository for libraries — plus, you get access to the library's script objects and properties, as well as its handlers.

The Libraries mechanism does have one downside: since your script does not explicitly load the library, the mechanism only works when you've edited and saved the script from Script Debugger. If you edit the script with Apple's Script Editor, for example, the script will break. To get around this — for example because you want to send the script to someone else, or run it in some other context than Script Debugger — you can "flatten" the script, as follows:

1. Choose Flattened Script from the Export submenu of the File menu.

The result is a script in which any library scripts attached to your script have been copied into your script.

You don't really have to keep libraries in the Script Libraries folder. Any compiled script can be dragged into a code window's Libraries tab to attach it to that window. Alternatively, single-click the plus-icon to get an Open dialog from which you can choose a script to attach to the window. Another alternative is to place an alias to a library in the Script Libraries folder; the advantage is that the library will then show up in the plus-icon menu. However, it is probably best to keep libraries in the Script Libraries folder, because Script Debugger will always be able to find them.

CHAPTER 3

Debugging Scripts

So far, we've seen how to create and edit scripts with Script Debugger. But now you're probably eager to see how Script Debugger lives up to its name by helping you debug your scripts! We'll start by walking through the process of debugging a typical sample script, trying out all of Script Debugger's various debugging techniques. Then we'll describe some further, higher-level features of the debugger.

Typical Debugging Techniques

Let's begin by creating a script that contains a bug. We'll encounter the bug and use Script Debugger to track it down in various ways.

1. Start up Script Debugger and enter the following script into the editing window:

```
on remove at whatItem out of whatList
   set outcome to {}
   if whatItem > 1 then
      set outcome to items 1 thru (whatItem - 1) of whatList
   end if
   if whatItem < length of whatList then
      set outcome to outcome & ¬
         items (whatItem + 1) thru (length of whatList) of whatList
   end if
   return outcome
end remove
set L to {"Manny", "Moe", "Jack"}
repeat with i from 1 to length of L
   if item i of L begins with "M" then
      set L to remove at i out of L
   end if
end repeat
```

The idea is as follows. AppleScript lacks a native verb that removes an item from a list; so we previously developed a utility handler, `remove`, which does this. Now we wish to use this handler to remove from the list `{"Manny", "Moe", "Jack"}` all items that begin with the letter "M". But, as we are about to discover, our script has a bug in it.

2. Compile the script, and run it.

An error message appears: `Can't get item 3 of {"Moe", "Jack"}`. What on earth does this mean? Rather than trying to puzzle it out, let's debug the script. The first thing to do is to turn on Script Debugger's debugging features.

Debug Mode

1. Change the language popup menu at the bottom of the window from "AppleScript" to "AppleScript Debugger".

This puts us into debugging mode so that we'll be able to track the values of variables while stepping through the script.

2. Flip down the "Description, Libraries, and Properties" triangle at the top of the window, and click the Debugging tab.

The Properties tab has changed to the Debugging tab, and this is one of the places where we'll be tracking the values of our variables.

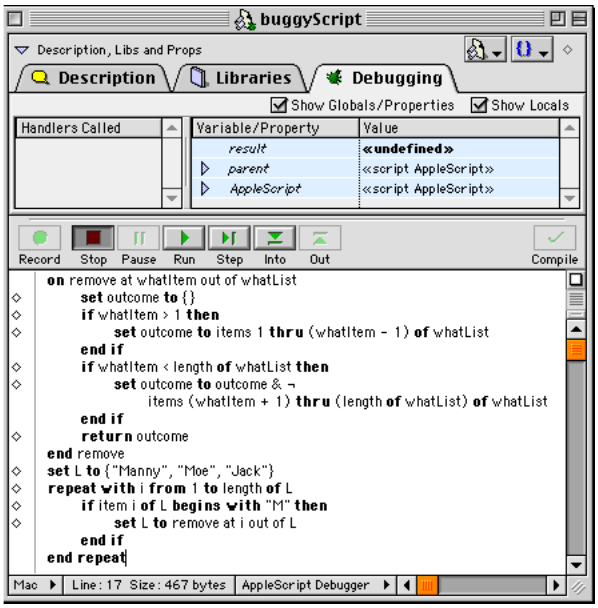
TIP: *Alternatively, choose Enable Debugging (Shift-Command-D) from the Script menu. This changes the language popup to "AppleScript Debugger", and opens the Debugging tab, in a single step.*

3. Compile the script.

This is necessary because debugging mode is a different language from regular AppleScript and the script has not yet been compiled in this different language.

The window should now look something like Figure 3-1. You can adjust the dimensions of the various window elements. For example, you can move the vertical divider between the Variable/Property column and the Value column, and you can move the horizontal divider between the Debugging tab and the code.

Figure 3-1
Getting Ready to Debug

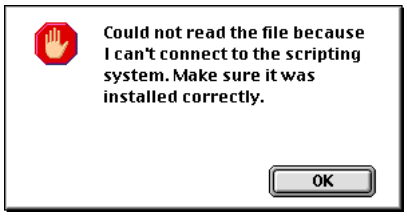


WARNING: *The script is now no longer an AppleScript script and cannot be opened without Script Debugger!*

While this warning isn't meant to scare you, it's important that you be alert to the fact that a script that has been compiled in AppleScript Debugger mode is no longer an AppleScript script. The reason for this is that AppleScript Debugger isn't merely a mode: it's actually a different language from AppleScript. It looks like AppleScript, and it acts like AppleScript, but it has some important differences from AppleScript — for example, you can debug it!

The single most frequently asked question in connection with Script Debugger is this: "I saved my script, and then the next day when I tried to open it, I got a mysterious message about not being able to connect to the scripting system." An example of such a message is shown in Figure 3-2. The problem is that the user saved the script in AppleScript Debugger mode and then tried to open it in the absence of Script Debugger.

Figure 3-2
Mysterious Message



The good news is that with Script Debugger 3 there's going to be a lot less of this sort of thing than previously, because the AppleScript Debugger language is now encapsulated in a system extension (or, on Mac OS X, a system component), which works even when the Script Debugger program is not running. But you can still get a big surprise if you use some other program (such as the Script Editor) to open a script saved in AppleScript Debugger mode, because when you try to run the script, it will start up Script Debugger and switch to it! And of course you could still get a message like this on a computer where Script Debugger isn't installed, if you tried to open a script saved in AppleScript Debugger mode.

In general, the solution and the moral are both very simple. When you're done debugging, switch the script back to normal AppleScript (choose Disable Debugging from the Script menu, or change the language popup menu in the window to AppleScript), and compile and save it that way!

Anyway, we are presently in AppleScript Debugger mode because we want to be, because we want to debug. Let's start by stepping through the script.

Stepping

1. Press the Step button twice.

Alternatively, you can choose Step Over (Command-Y) from the Script menu.

There are two things to notice. First, you will see that the blue arrow at the left of the code advances to show the progress of execution. (The blue arrow always shows *the line that is about to execute*.)

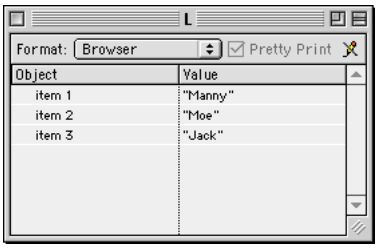
Second, since we have now passed the point where the variable `L` is brought into existence, the variable `L` appears in the list of variables in the Debugging tab. If you can't see it, scroll down within the Debugging tab or adjust the height of the tab. Because the variable `L` is a list, its value is not immediately displayed. To see its value, you have three choices: you can click open the triangle at the left of `L`'s listing, in which case the values of its items are shown beneath it; or, double-click `L`'s listing to open a Viewer window dedicated to `L`; or, if all you want to do is glance at `L`'s value momentarily, hover the mouse over `L`'s value listing in the Debugging tab or within the script, to cause a tooltip to appear.

Personally, I like the Viewer window best, especially because if you have several variables to keep track of, you can maintain separate Viewer windows for each. Such a window can display its contents in several formats, just like the Script Result window; here, the default Browser format is fine. So let's look at it.

2. Double-click L's listing in the Debugging tab to open its Viewer window.

The resulting window is shown in Figure 3-3.

Figure 3-3
A Variable Viewer Window



3. Return to the main window and press the Step button again.

By advancing past the `repeat` line, we have now brought the variable `i` into existence. Its value appears in the Debugging tab.

This is the moment we have been waiting for, because we need to understand both what the value of `L` and the value of `i` are doing in order to follow the logic of the script. You might even like to open a Viewer window for `i`. We will now continue to step through the script, watching three things: the value of `L`; the value of `i`; and the path of execution.

4. Press the Step button repeatedly.

This causes the interior of the `repeat` loop to execute repeatedly. If you're observant, you will have noticed the following phenomena: on the first pass through the `repeat` loop, everything goes fine; on the second pass, `"Moe"` is not deleted (why not?) and the material inside the `if` condition is never executed (why not?); on the third pass, `i` is `3` but `L` has only two items, and we get our error.

It's probably obvious to you at this point what the problem is. But let's pretend that it isn't. This will give us an excuse to explore the script more deeply.

Stepping In and Stepping Out

One reason why it might not be totally clear what's causing our bug is that we are not getting to watch the execution path dive into the `remove` handler that is called at the deepest level of the `repeat` loop. On the first pass through the loop, we see `L` change but we do not step through the `remove` handler. Why is this? It's because the Step button means Step Over. This specifies that we don't step through subroutines (they are executed in a single step).

If you'd like to dive into `remove`, then instead of repeatedly pressing the Step button, you'd repeatedly press the Into button (or choose Step Into, Command-I, from the Script menu). This progresses one line at a time, just like Step, but when a subroutine is encountered, it dives into the subroutine. The blue arrow progresses into the subroutine, leaving a hollow arrow at the point where the subroutine was called.

If you try pressing Into repeatedly so that the execution path dives into `remove`, you'll notice that the calling chain is displayed at the left side of the Debugging tab. You can click a handler name to change the scope of the variables displayed on the right side. So if, while the execution path is within `remove`, you click `Run (implicit)`, the parameters `whatItem` and `whatList` vanish because they are not visible at the top level of scope. If you click `remove`, they reappear because they are visible there.

As long as we're paused inside the `remove` handler, this is the moment to mention the Out button (Step Out from the Script menu). This is a quick way to "escape" from a handler that you've dived into. It executes all lines within the present handler in a single step, pausing back in the caller after execution of the line that called the handler.

Missing Variables

When the execution path steps into the `remove` handler, you will observe that the important variable `outcome` is not displayed. Why not? The answer has to do with the way Script Debugger has to hack into AppleScript in order to provide debugging facilities. It turns out that AppleScript doesn't give Script Debugger quite enough information for Script Debugger to know about a variable *when that variable is implicitly declared and is local to a handler.* To get around this, you have to give Script Debugger a little help by declaring local variables explicitly. If you insert this line as the first line of the `remove` handler:

```
local outcome
```

then when you step into `remove`, after you pass through this line, the variable `outcome` is displayed in the Debugging tab as soon as it acquires a value. The point is important enough to deserve a note:

NOTE: *If you want Script Debugger to be able to display variables local to a handler, you need to declare those variables explicitly in your script.*

Missing Parameters

Under certain circumstances, Script Debugger can have the same sort of problem with parameters passed to handler that it has with implicit local variables in a handler: it can't detect the parameters, so it can't display them in the Debugging tab.

If you encounter this problem, you can help Script Debugger by declaring the parameter names explicitly. In fact, this is also an alternative to the use of a `local` declaration to bring a local variable to Script Debugger's attention. To do this, you use a special comment syntax: a comment starting with an equal-sign and consisting of the name or (comma-delimited) names of parameters or local variables that already have a value but that Script Debugger is failing to discover on its own. For example, the `remove` handler could start like this:

```
on remove(whatItem, whatList)
```

```
set outcome to {}
--= outcome, whatItem, whatList
```

Notice that since the comment line is being used to tell Script Debugger about `outcome`, it needs to appear after `outcome` has acquired a value. Here's a note to summarize:

NOTE: *If, despite your best efforts, a local variable or parameter is simply failing to appear in the Debugging tab, insert a comment-equals line into the script after a point where the local variable or parameter has acquired a value.*

Please keep in mind that this special comment syntax should be used *only as a last resort*. You should *not* actually use it in the situation depicted above, because `outcome` can be made to appear by declaring it as local, and `whatItem` and `whatList` appear automatically. Since Script Debugger is not actually having problems finding the parameters or locals of the `remove` handler, you should not use the special comment syntax; it was used here purely to show you that the special comment syntax exists. Later, we'll see an example where the special comment syntax is actually needed.

Breakpoints

Stepping through the code one line at a time can be tedious and confusing. And it often happens that what's of interest is just a particular line of code. In our example, we notice that there is really only one moment when the state of the variables is of interest: the `if` line inside the `repeat` loop. This is the line that is triggering the bug, and it would be helpful to see nothing but the state of `L` and `i` just before this line executes each time through the loop. To arrange this, here's all you do:

1. Click in the diamond in the left margin of the "if" line.

This sets a breakpoint at that line. Alternatively, you could select within the line and choose Set Breakpoint from the Script menu. The diamond is filled with red to show that a breakpoint has been set. Now, instead of stepping:

2. Press Run.

Script Debugger performs multiple lines in a single step, pausing at the point where the breakpointed line is about to be executed. Thus, all we have to do is:

3. Press Run three more times.

On each press of the Run button, we perform another iteration of the `repeat` loop and pause before the breakpoint. This allows us to study the state of the variables and to draw the same conclusions as before. On the next-to-last press of the Run button, we can see that `i` is `3` but `L` has just two items; on the last press, we get our error.

A breakpoint always supersedes the rules for stepping. For example, if you set a breakpoint inside the `remove` handler and then debug by pressing Step repeatedly, you will pause inside the `remove` handler before the breakpointed line, even though Step means not to dive into subroutines. Similarly, if you press Out, Script Debugger normally continues execution until it has passed out of the current handler; but if a breakpoint is encountered first, Script Debugger will pause there.

Command-clicking on an empty diamond is a shortcut for setting a breakpoint and issuing a Run command, in a single move.

Option-clicking on an empty diamond is a shortcut for setting a *temporary* breakpoint and issuing a Run command, in a single move. This means that execution will pause before this line the next time it is encountered, and the breakpoint will then be automatically removed. If you option-click an empty diamond and the implicit Run causes execution to hit a different breakpoint, you will actually see the temporary breakpoint (it has an orange fill). But if no other breakpoints are encountered, you'll never see the temporary breakpoint, because this single click will set a breakpoint, cause execution to progress to and pause at that breakpoint, and remove the breakpoint, all in

a single move — known in high-class debugging circles as Execute To Here. Alternatively, select within the code itself and option-choose Run To Cursor from the Script menu (Command-Option-R).

Tracing and Code Coverage

To trace a script means to watch its execution path. It works just like Run except that the blue arrow signifying the line presently being executed pauses momentarily at every line of code encountered. In this way, you can see where the path of execution goes.

To start tracing, shift-choose Trace from the Script menu (Shift-Command-R). Alternatively, shift-click the Run button. Another alternative is to shift-click an empty diamond: this sets a temporary breakpoint and starts tracing.

You can adjust the speed at which tracing occurs. At one extreme, you can barely see the blue arrow as it zips through your script. At the other extreme, the blue arrow plods along from line to line. To make the adjustment, choose Preferences from the Edit menu (the Script Debugger menu in Mac OS X) and look in the Scripting Settings panel.

If you try tracing our buggy script, you'll see that we only dive into the Remove handler once, the first time through the Repeat. Since there are two items in the list beginning with "M", this is a big clue as to what's wrong.

A feature related to tracing is code coverage. This simply means that as the execution path passes through a line of code, Script Debugger marks that line. Sometimes, tracing is overkill or too cumbersome, and code coverage is really all you need. To try it out, first choose Clear Coverage Marks from the Script menu, then run the script. Afterwards, you'll see some vertical blue lines next to the divider between the breakpoint area of the code window and code area. Those are the coverage marks, showing all executable code that was actually executed. We don't learn much from this information in the case of this particular script, but we do notice that this line is never executed:

```
set outcome to items 1 thru (whatItem - 1) of whatList
```

From this we can conclude that the condition in the "if" line preceding is never true, and so the only item removed from L must be its first item. That is, in fact, a clue as to what's wrong with the script.

(Observe that we would not have gotten such clear information if we had used the single-line "compound" form of if-statement:

```
if whatItem > 1 then set outcome to ...
```

Since the condition is tested, the line is shown as executed, even if the "then" part doesn't execute. This could be a good reason to use the multi-line `if...end if` construct, as shown.)

Watchpoints

Another technique for debugging this script might be to use a watchpoint, a device which instructs Script Debugger to pause when the value of a certain variable changes. Indeed, this technique makes a great deal of sense in this case, since the entire problem here is the interplay between two variables, `L` and `i`. So, let's set a watchpoint on `i`. The marvelous thing is that we can simply do this without bothering to work out, from the logic of our script, where in the course of execution these variables are actually set. Of course, in this particular example, that's trivial to understand. But with a script of only slightly more size and complexity, a watchpoint could be much more informative and convenient than any technique we've seen so far.

To test the use of watchpoints, first remove all breakpoints, perhaps by choosing Clear All Breakpoints from the Script menu. Now:

1. Press Step three times.

This progresses us far enough into the script that both `L` and `i` have come into existence in the Debugging tab.

2. Click in the diamond to the left of i in the Debugging tab.
3. Press Run.

Script Debugger stops. The listing for `i` in the Debugging tab is marked by a red arrow, meaning that the watchpoint for `i` has been triggered, and is printed in red, meaning that the value of `i` has changed. This is a good time to use the tooltips feature. You can hover the mouse over `i` and `L` in the line about to execute to discover that we are about to test whether `"Jack"` begins with "M", skipping `"Moe"`.

4. Press Run.

We pause again. Now we see that `i` is about to test the non-existent third item of `L`, which will cause the error message.

Expressions

The final debugging device provided by Script Debugger is expressions. An expression is a sort of question that you ask, using the AppleScript language, about the variables and values in your script. Once you've created an expression, your question is automatically posed, and the answer provided, any time the script stops at a breakpoint. You also can treat an expression as a watchpoint, so that the script stops every time the expression changes and you're shown its new value.

To create an expression, choose New Expression from the Script menu, or select some code and choose Copy To Expressions. Expressions are listed in the Expressions window, which can also be summoned from the Window menu. The "+" button in the Expressions window is the same as choosing New Expression; the trash button lets you delete an expression.

This might be the best way of all to debug our script. Let's try it:

1. In the second line of the Repeat loop, select "item i of L" and choose Copy To Expressions.
2. In the Expressions window, click the diamond next to "item i of L" to set a watchpoint on this expression.
3. Click the "+" icon and type "L" (and hit Return) to create a second expression consisting of just the variable "L".
4. Go back to the script and make sure all breakpoints are cleared.
5. Press Run several times.

The first time you hit Run, the Expressions window shows that "item i of L" is "Manny"; and you can flip open the triangle next to "L" to see that it consists of the three-item list "Manny", "Moe", and "Jack". So far, so good.

The second time you hit Run, the Expressions window shows no useful information. What's happened is that the query "item i of L" has lost its meaning because "i" has gone out of scope. This counts as a change in the value of this expression, so the script has paused. It happens that this doesn't interest us, so we just proceed.

The third time you hit Run, the Expressions window shows that "item i of L" is "Jack" and that L is "Moe", "Jack". This reveals vividly what's gone wrong. "Moe" has slid down into first place in the list, but meanwhile we're testing "Jack", skipping "Moe" altogether.

The fourth time you hit Run, the Expressions window shows that "item i of L" evaluates to an error! This is our same old error. The script itself has not yet generated the error, because the line in question hasn't executed yet. But Script Debugger is looking ahead; it has evaluated the expression before the line containing that expression executes, so you know in advance that this expression is the problem and that it is about to bring the script to a grinding halt. If you hit Run once more, it will.

This completes our tour of the debugger's basic features. I assume that by now you know what's wrong with the script! Just in case you've been enjoying playing with Script Debugger so much that you haven't been giving any thought to this question, I'll tell you

the answer: it's that the Repeat loop runs forwards through the list at the same time that it deletes items from the list. After it deletes the first item, it proceeds to look at the second item, ignoring the new first item completely. When it gets to the third item, there is no longer any third item. The solution is for the Repeat loop to cycle backwards through the list instead:

```
repeat with i from length of L to 1 by -1
```

This prevents the error message, squashes the bug, and gives the right answer.

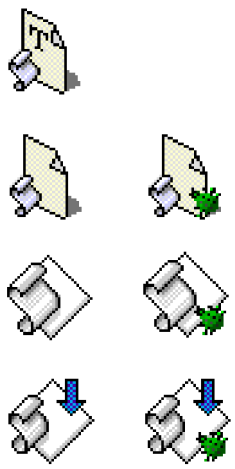
Debugging Details

You've now been introduced to the basics of the debugger, and you know enough about Script Debugger to start using it immediately. This section presents some further miscellaneous details about the debugger.

File Types

We saw earlier ("Debug Mode") that if you save a script when the language popup at the bottom of the code window says AppleScript Debugger, the resulting file cannot be run except by Script Debugger itself. To help you distinguish between the various sorts of files it creates, Script Debugger uses different icons, shown both in the Finder and at the top of each script window. These are displayed in Figure 3-4. Notice the little green bug in the icons of the column to the right. These are files saved in AppleScript Debugger mode. Running one of them will cause Script Debugger, if present, to open and to display the script, ready to debug. If Script Debugger is absent, such a script can't be run at all.

Figure 3-4
Script Debugger's File Icons



(Unfortunately, the second-column icons don't exist in Mac OS X. That's because they are implemented through a feature called "icon badges", which Mac OS X doesn't currently support.)

The first row shows a text file. Because this is pure text and not a compiled script, it has no language and can be opened by just about any program (the Script Editor, any word-processor, and so forth).

The second row shows a compiled script. The file on the left can be opened by Script Editor, and can be executed by programs such as OneClick or OSAMenu that know how to do that sort of thing. The file on the right was saved from the debugger.

The third row shows a script application. The file on the left will execute if opened from the Finder. The file on the right was saved from the debugger and can't be run except from inside Script Debugger.

The last row is the same as the one above it, except that the script applications have Open handlers (they are "droplets").

Variable and Property Names

Variable and property names in the Debugging tab are color-coded. A variable or property whose value has changed since the last pause is shown in red. Script properties and global variables have a grey background (white in Mac OS X), and can be hidden by unchecking the Show Globals/Properties checkbox. Local variables have a yellow background, and can be hidden by unchecking the Show Locals checkbox. AppleScript's own global properties (such as the Text Item Delimiters) are shown in italic with a blue background.

You can alter the value of globals and properties while a script is paused in debugging mode. To make a value editable, select it and hit Return. But you cannot do this with local variables.

One of AppleScript's global properties is the `result`. This is the first item listed in the Debugging tab. Keep in mind that this doesn't refer merely to the final outcome of the script; *every* executable line of AppleScript has a `result`, which you can use to help track the action of your script as you step through it. Also, if the Properties or Debugging tab is visible, then after running a script, the Script Result window does not automatically appear; the reasoning here is that you already have access to the `result` through the Properties or Debugging tab.

One At a Time

A very important limitation to be clear about is that Script Debugger can only debug one script at a time. If you forget this, you can easily become confused. If you're in the middle of debugging a script, and you leave it in a paused state, and you start working with a second script in a different code window, you'll find you can't compile or run that second script, and you won't understand why.

If this happens, look in the Window menu. You'll see that one of the code windows listed there is marked with a chasing-arrows icon (two curved arrows arranged in a circle). That's the script that's paused. You can switch to it and stop it if you want to compile or run a different script.

Script Applications

When you debug with Script Debugger in the ordinary way as described in the first section of this chapter, by pressing Step or Run for example, what you are debugging is the script's Run handler. This handler can be implicit, in which case the Run handler consists of the script's top-level commands (those outside any handler), or it can be explicit, meaning that an actual `on run` block is present.

Script applications, however, can also have an Open handler, executed when files or folders are dropped onto the application's icon in the Finder. If a script application stays open after executing its Run handler and/or its Open handler, it can also have an Idle handler and a Close handler. To debug an Open handler, an Idle handler, or a Quit handler, you need a way to tell Script Debugger that you want the path of execution to start within this particular handler and not, as in the default case, the Run handler.

For this purpose, Script Debugger provides four utility scripts in the Debugging Tools subfolder of the Scripts folder. You can run these utility scripts either by double-clicking the appropriate entry in the Scripts palette (shown in Figure 3-5) or by choosing the appropriate item from the  menu. If you're not in debugging mode, you simply run the specified handler; if you are in debugging mode, Script Debugger steps into the specified handler and pauses, ready to debug.

The Open handler is special in that it must receive a parameter which is a list of aliases; the two scripts for running the Open handler therefore bring up a dialog to pick a file or a folder that will serve as the first and only item in that list. Unfortunately, a single-item list may not exercise your Open handler in a very interesting way. Fortunately, there's another technique: instead of using the utility scripts, select some items in the Finder

Figure 3-5
The Scripts Palette

and drag them directly onto your code window in Script Debugger. If that code window contains an Open handler, a dialog appears asking what you want done with the Finder items. The first choice is to use them to call the Open handler.

The following example of a script application will clarify. It reports the number and total size of the Finder items dropped onto it.

1. Enter the following code into a new Script Debugger code window and place the window in debugging mode.

```
on open theFiles
   --=theFiles
   local total, f
   set total to 0
   tell application "Finder"
      repeat with f in theFiles
         set total to total + (size of f)
      end repeat
   end tell
   display dialog "You dropped " & length of theFiles & " file(s) " & ¬
      "totalling " & (total div 1024) & "K."
end open
```

2. Select some files in the Finder and drag them into the Script Debugger code window.
3. In the dialog that appears, select the first option.

The debugger is now paused inside the Open handler, before its first executable line. The local variables `total` and `f` do not yet appear in the Debugging tab, but they will as they are assigned values when you step through the code.

Observe that the parameter `theFiles` does appear in the Debugging tab; you can see that it consists of a list of the Finder items that you dropped on the window. That's due to the comment-equals line at the start of the handler:

```
--=theFiles
```

This syntax was explained earlier ("Missing Parameters"), and is necessary here; without it, Script Debugger can't display the value of the Open handler's parameter. Indeed, command handlers are the main place you'll need this syntax.

Script Objects

Script objects provide AppleScript's object-based programming facilities. Debugging in the presence of script objects is not more complicated; but it does introduce some additional levels of context.

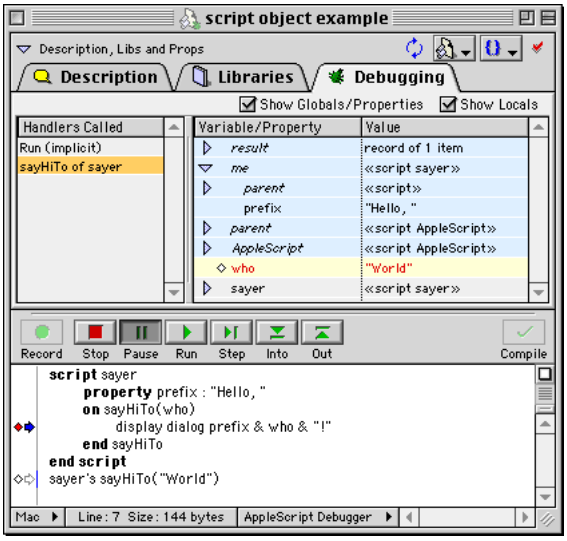
Here's a rather artificial example:

```
script sayer
   property prefix : "Hello, "
   on sayHiTo(who)
```

```
      display dialog prefix & who & "!"
   end sayHiTo
end script
sayer's sayHiTo("World")
```

Set a breakpoint on the "display dialog" line, and run to it. You'll notice two things. First, you'll observe the presence in the Debugging tab of the script object `sayer` as a global. Second, once we have stepped into the `sayer` script object, the AppleScript property `me` appears. It is equivalent in this context to the script object `sayer`. If the `me` listing in the Debugging tab is expanded, it shows the script object's `parent` property (which in this case is the main script). This is also where any properties of the script object are displayed. (See Figure 3-6.)

Figure 3-6
Debugging a Script Object



The truth is that there is an AppleScript property `me` in *every* context. This is needed, for example, so that a script can explicitly distinguish its own properties from those of the target application in a `tell` block. But Script Debugger has no need to identify `me` explicitly except in the context of a script object.

External Debugging

Script Debugger can be invoked from other contexts — that is, having written an AppleScript in some other environment that can run OSA scripts, such as the Script Editor, or UserLand Frontier, or HyperCard, you can debug the script using Script Debugger. When you use Script Debugger in this way, debugging takes place in Script Debugger in a temporary code window, which vanishes when debugging ends.

The reason this is possible is the same reason why you can save a Script Debugger script in a state where it *can't* be run by other applications ("Debug Mode," above). Script Debugger's AppleScript Debugger is an OSA language, present at system level and available from other OSA-compliant script editing environments.

Here's a brief exercise to test this feature using the Script Editor. Make sure before starting that Script Debugger is running and that it is not in the middle of any debugging.

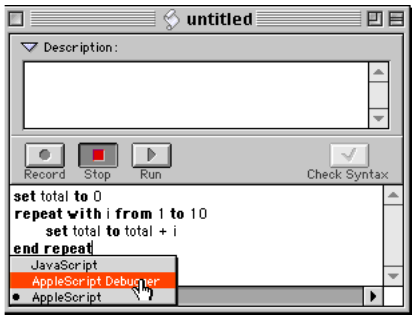
1. With Script Debugger running, start up Apple's Script Editor and enter this script:

```
set total to 0
repeat with i from 1 to 10
   set total to total + i
end repeat
```

2. Using the popup menu at the bottom of the Script Editor code window, switch the OSA dialect from AppleScript to AppleScript Debugger.

This step is illustrated in Figure 3-7.

Figure 3-7
Preparing to Debug Externally



3. Press the Run button.

Script Debugger comes to the front, showing the script you entered in the Script Editor in debugging mode, paused before the first executable line. Now you can step through the script as usual. When the script is finished, the window vanishes and you are automatically returned to the Script Editor.

WARNING: *Script Editor can save a script whose OSA dialect is AppleScript Debugger. Afterwards, though, Script Editor won't be able to run the script without Script Debugger. This is the same situation discussed earlier ("Debug Mode" and "File Types").*

Folder Actions

It might seem that external debugging is merely a trick or a piece of esoterica. After all, you could always copy the script into Script Debugger and debug it there, and you'd probably prefer to do so, because Script Debugger is such a good editing environment. But there are situations where external debugging is highly practical — where it is, in fact, the *only* viable way to debug a script. A folder action script is one such situation.

Folder action scripts are triggered by the Folder Actions background-only application, which looks in the script for a handler of the relevant type and passes it the correct parameters. Since the folder action script has no Run handler or top-level statements, and since the script's behavior depends upon the parameters that only the Folder Actions application can supply, there is no way to test the script properly except under the real-life conditions where it is triggered by the Folder Actions application.

The way to debug a folder actions script, then, is to save it from Script Debugger in AppleScript Debugger mode. You then leave Script Debugger running and perform an action in the Finder that will cause the Folder Actions application to trigger the script. Since the script is in AppleScript Debugger mode, this will cause you to fall into Script Debugger with the script displayed as for external debugging. You can then debug as usual. Once the script is debugged, you save it again as AppleScript.

As an example, we will create and debug a folder action script that calls StuffIt Expander to expand any binhex files that are dropped into a folder.

1. In the Finder, create a new empty folder.

I will call this the "test folder". In fact, I will also name mine TestFolder, but you can call yours anything you like.

2. Start up StuffIt Expander.

This is just so that the script will know, the first time it runs, where to find StuffIt Expander.

3. Start up Script Debugger and create this script:

```
on adding folder items to this_folder after receiving added_items
   local f
   --= added_items
   repeat with f in added_items
      if (f as string) ends with ".hqx" then
         tell application "StuffIt Expander" to Expand f
      end if
   end repeat
end adding folder items to
```

4. Switch to debugging mode, save the script, and close it.

You can save the script anywhere you like and give it any name you like, but you have to remember where it is in order to perform the next step. In this case, I'm calling the script Expander and keeping it in TestFolder.

5. In the Finder, select the folder you created and control-click on it to bring up the contextual menu. Choose Attach a Folder Action and in the resulting dialog choose the script you created.

The folder action is now created. For example, in my case, I have now attached the Expander script as a folder action to the TestFolder. Now comes the really interesting part — we're going to debug the folder action script.

6. In the Finder, open the folder you created and drag a file or files into it.

The folder must be open in the Finder, because the folder action script won't be triggered otherwise. Since it *is* open, the folder action script *is* triggered, and since the script is in AppleScript Debugging dialect, Script Debugger comes to the front and displays the script in external debugging mode, paused before its first executable line. You can now debug as usual. As you step through the script, the value of `f` changes to reflect each of the files that you dragged into the folder. If the name of any of those files ends with ".hqx," the file is expanded by StuffIt Expander.

If you do drop an ".hqx" file into the folder, you'll observe something interesting. When StuffIt Expander expands the file, this creates a new file (or more than one, for example a ".sit" file and a fully expanded file or folder) — and the new file triggers the folder action script again! Script Debugger makes this fact perfectly clear because after the script has completed and vanished from Script Debugger, it suddenly reappears, and we can see that this is because of the new expanded file. This is just the sort of information that would have been hard to discover without Script Debugger. At this point, if we think this is a problem, we can modify the script. Luckily, the new file's name usually doesn't end in ".hqx", so we probably won't end up in some sort of vicious cycle.

We are now finished debugging and can close up shop. Here's how.

7. Quit StuffIt Expander.

The script will run fine from now on without StuffIt Expander running because it knows where to find StuffIt Expander on your hard disk.

8. In Script Debugger, open the folder action script, switch to AppleScript dialect, save it, and close it.

This way, from now on whenever the folder action script is triggered, it will just run, without entering external debugging mode.

You've now created and debugged a folder action script.

CHAPTER 4

Reference

This chapter provides brief descriptions of Script Debugger's windows and menu items. You may never need these descriptions, but isn't it nice to know they're here?

The previous chapters have provided a hands-on tour of Script Debugger's main features. They were meant to be helpful, friendly, and instructive. This chapter is not! It is meant to be compendious, factual, and stark. Don't try to read this chapter from start to finish — your eyes will glaze over. Come here to look up something. Even then you should be looking at the relevant interface item in Script Debugger itself or you won't understand a word.

In the interests of space and efficiency, screen shots are not provided in this chapter. After all, you're presumably already looking at the interface item in your own copy of Script Debugger. What you want to know is what the interface item does, not what it looks like. In general, interface items within a window are listed roughly in the order left-to-right, top-to-bottom. If a tab or other interface item is needed to reveal other interface items, they are listed in that order, i.e. the thing that reveals the others, followed immediately by those others.

Windows

Code window

Script Debugger's document window. This is where you create, edit, and debug a script. See also various places in Chapter 2, *Creating and Editing Scripts* (especially "Code Window Features" on page 15), and the entirety of Chapter 3, *Debugging Scripts*. On Projector support features, see Appendix B, *Projector Support*.

Header expander

Click the disclosure triangle to show or hide the Description, Libraries and Properties tabs.

Header divider

Drag the horizontal divider line to adjust the percentage of the window devoted to the Description, Libraries and Properties tabs, above it (as opposed to the code, below it). This percentage can be zero, so this is another way to show or hide the tabs.

Description tab

Area for entering a description of your script. This description can be displayed when script applications are launched. It also appears in the Help Balloon for script applications and when Show Preview is enabled while opening files.

Libraries tab

Lists libraries attached to your script. Drag entries to reorder them. Double-click an entry to open a library.

Add library (plus-icon)

Click for an Open dialog to attach a library to your script. Click and hold for a popup menu listing items in the Script Libraries folder; choose from this menu to attach one.

Remove library (trash-icon)

Click to detach selected libraries from your script.

Properties tab

Displays the script's properties and global variables. When debugging, this becomes the Debugging tab and can also display local variables. Click on triangles to reveal subitems. Double-click an entry to display in its own Viewer window. When debugging, click on diamonds to set or clear watchpoints; select an entry and hit Return to edit a value (not local variables). For color-coding, see "Variable and Property Names" on page 42.

Stack frames

When debugging, displays the chain of called handlers. Click on a handler name to change the scope of the variable display and to highlight the line currently being executed in that handler. Double-click on a handler name to highlight the line that called the handler.

Show globals checkbox

Toggles display of global variables and properties when debugging.

Show locals checkbox

Toggles display of local variables when debugging.

Progress arrows

Indicates the state of an executing script. Blue means the script is running; orange means the script is waiting for an Apple event to complete. A flashing microphone means the script is recording.

Script type popup

Determines the type of file this script will be saved as. During external debugging, this changes to the client application button (see next paragraph).

Client application button

Brings the calling application frontmost during external debugging.

Show startup button

Visible only if the script is a script application. Toggles whether or not your script's description is shown when the application is launched.

Stay-open button

Visible only if the script is a script application. Toggles whether or not the application keeps running after its Open or Run handler executes.

Handler popup

Click for a menu navigating among markers (comments starting with >>), handlers, properties, and explicitly declared global variables in the script. Option-click to see handlers only.

Dirty icon

A red checkmark means changes have been made since the script was last saved.

Controls display icon

Click to toggle display of controls (“Record”, “Stop”, etc.) in the window. See “Controls palette” on page 53.

Line wrap icon

Click to enable or disable line wrapping. Straight lines mean line wrapping is off.

Splitter

Drag to create a new pane.

Resizer

Drag to resize split panes. Double-click to collapse split panes back into one.

Left margin

Red arrows show the location of the most recent error; click to display the last error. Blue arrows indicate the current line when debugging. Gray arrows indicate where a handler has been called. Click on diamonds to toggle breakpoints; for modifier-clicking, see “Breakpoints” on page 38. A red diamond is a breakpoint; an orange diamond with a cross in it is a temporary breakpoint. Blue vertical lines at the margin divider indicate lines that have been executed (see “Tracing and Code Coverage” on page 39).

Status area

Displays the current line number and length of the script. Click to jump to a specific line.

Languages popup

Click to switch to another Open Scripting Architecture (OSA) language.

Line-delimiter popup

Click to switch to a different end-of-line delimiter; applies only when the script is saved as text. The default is Macintosh (CR) unless you open a text file saved with a different line delimiter.

Script Result window

Displays the result after a script runs, appearing automatically if the Properties or Debugging tab is not open. For pane splitting and line wrapping, see on the Code window, above (page 48).

Script popup

Switches among the results of recently run scripts that are still open.

Format popup

Switches among display formats for the script result. Source Language means the scripting language (e.g. AppleScript). AEPrint displays the raw Apple event. In Best view, you can see pictures, QuickTime movies (press the Play button to start the movie), and QuickDraw 3D drawings. In Browser view, you can click on triangles to reveal subitems and double-click an item to display it in its own Viewer window.

Pretty-print checkbox

Check to have lists and records formatted in a more readable fashion.

Apple Event Log window

Displays Apple events sent by scripts as they run (and replies received). See also “The Log Window” on page 29. For pane splitting and line wrapping, see on the Code window, above (page 48). For formatting and pretty-printing, see on the Script Result window, above (page 51).

Show Events checkbox

Toggles display of outgoing Apple events.

Show Replies checkbox

Toggles display of incoming replies. Replies are marked by an arrow glyph (- - >).

Append Logs checkbox

Toggles whether or not existing contents are retained the next time a script runs.

Trash icon

Click to clear window contents.

Expressions window

Displays the values of defined expressions. See also “Expressions” on page 40.

Script popup

Switches the display of expressions among open code windows.

Add expression (plus-icon)

Click to add an expression. Type the expression in the editable field and hit Return.

Remove expression (trash-icon)

Click to delete the selected expression.

Expression column

Click a diamond to set or remove a watchpoint on an expression. Double-click an expression (or select and hit Return) to edit it.

Value column

Double-click a value to create a Viewer window for that expression.

Properties & Globals window

Script popup

Switches the display of properties and globals among open code windows.

Viewer window

This is the independent window, dedicated to the display of a single value, that appears when you double-click an entry in a browser view such as the Properties tab, the Script Result window (in Browser view), the Properties & Globals window, the Expressions window, the Explorer window, or another Viewer window. See further on the Script Result window, above (page 51).

Dictionary window

Displays an application's dictionary (`'aete'` resource). See "The Dictionary Window" on page 20. For pane splitting and line wrapping, see on the Code window, above (page 48).

Application popup

Chooses an application whose dictionary will be displayed in this window. Also allows the currently chosen application to be launched, quit, or revealed in the Finder.

Dictionary panel

Displays dictionary entries verbally: name on the left, content on the right. Click a name to view its content. Copy and paste an event name (verb) into a code window to get a template for that event.

List popup

Filters and orders the names of dictionary entries displayed.

View As popup

Determines syntax for displaying the content of dictionary entries.

Object Model panel

Displays dictionary entries hierarchically (classes only): hierarchical diagram on top, content on the bottom. In the hierarchical diagram, click a class to display its dictionary content. Click a plus or minus button to show or hide the hierarchy below a class.

Hierarchy popup

Determines the type of hierarchy to be displayed: classes as elements of one another, or classes as inheriting from one another.

Root Class popup

Determines which top-level class should be the starting point for the hierarchy diagram.

Explorer panel

Displays the application's actual current objects and their values, as an interactive browser. Elements are bold, properties are italic. Click a disclosure triangle to request polling and display of subitems; double-click an entry's name to display it in a new Inspector window. Hover the mouse over a property name to see a tooltip displaying its dictionary comment. Use the popup menu at the right of an element's name to specify the reference form for it; the default is "every" (option-choose this popup menu to force all its items to be enabled). Drag or copy-and-paste an entry's name into a code window to get a reference to that object. Double-click a value to open a Viewer window for that value. Select a value (if not read-only, indicated by a slashed-pencil icon) and hit Return to edit the value. Did you read the warning on page 21??? See also Appendix A, *Issues With Scriptable Applications*.

Update button

Refreshes the currently selected item and its subitems. Shift-click to refresh all items.

Inspector window

This is the independent browser window that appears when you double-click an element name in the Dictionary window's Explorer panel. You can also summon an Inspector window by choosing New Explorer from the Open Dictionary submenu of the File menu. See on the Explorer panel, above.

Change button

Lets you specify a different base object reference for this window.

Palettes

Controls palette

Provides quick access to Script Debugger's script execution, debugging, and recording commands with respect to the frontmost code window. All the commands offered by this palette are also present in the Script menu. The zoom box toggles between horizontal and vertical orientations. The Control palette buttons can also appear in code windows. See "Controls" on page 15.

Record

Begins recording the actions of other applications (if recordable) in your script.

Stop

Stops recording, or stops your script running or debugging.

Pause

Pauses your script. Selected automatically when Script Debugger pauses during debugging.

Run

Compiles and, if successful, runs your script. During debugging, proceeds to the next breakpoint or watchpoint. Shift-clicking traces through your script.

Step Over

In debugging, executes the next statement. If the statement invokes a handler, all the statements of that handler are executed (unless a breakpoint or watchpoint is encountered).

Step Into

In debugging, executes the next statement. If the statement invokes a handler, execution pauses at the first statement of that handler.

Step Out

In debugging, executes the balance of the current handler (unless a breakpoint or watchpoint is encountered first), and pauses after the handler completes.

Compile

Compiles your script.

Windows palette

Provides rapid access to all open windows (meaning normal windows, not palettes). This is the same list that appears in the Window menu. Double-click an item to bring that window to the front. Open windows can also be brought to the front by way of the command-key shortcuts that are automatically assigned to the first nine of them opened. The Windows palette can be displayed separately or as a tab of a combined palette.

Close

Closes the selected window.

Save

Saves the selected window.

Print

Prints the selected window.

Run

Runs the script in the selected window, if it is a code window.

Options popup

Collapses or zooms the selected window; reveals the corresponding file in the Finder (a saved script, or a dictionary's application); and toggles the display of icons in the palette.

Applications palette

Lists commonly used applications for rapid access. The items listed here also appear in the Open Dictionary submenu of the File menu and in the Application popup of a Dictionary window. Double-click an item to open its dictionary. Option-double-click an item or drag it to a code window to paste a `tell` block. Applications are added automatically when you open a dictionary, or manually through the More Dictionary Settings panel of the Preferences dialog. The Applications palette can be displayed separately or as a tab of a combined palette. See also "The Applications Palette and Menu" on page 20.

Paste Tell

Pastes a `tell` block for the currently selected application into the frontmost code window.

Open Dictionary

Opens the currently selected application's dictionary.

Options popup

Launches, quits, or reveals an application; removes the selected application from the Applications palette; and toggles the display of icons in the palette.

Scripts palette

Displays all folders and scripts (or aliases to folders and scripts) in the Scripts folder. Double-click a script to execute it. Option-double-click a script to open it. Double-click a folder to open it in the Finder. The items listed here also appear in the  menu. To define a tooltip comment for a folder, create a text file named " Read Me" (note the initial space) in that folder. To define a tooltip comment for a script, give it a description. The Scripts palette can be displayed separately or as a tab of a combined palette. See also "Scripts Menu and Palette" on page 18.

Run

Executes the currently selected script.

Edit

Opens the currently selected script.

Options popup

Shows the currently selected script or folder in the Finder; toggles the display of icons in the palette; and sets or changes an item's keyboard shortcut.

Clippings palette

Displays all folders and text clippings (or aliases to folders and text clippings) in the Clippings folder. The items listed here also appear in the  menu. Double-click or drag a clipping to paste it into the frontmost code window. Option-double-click a

clipping to reveal it in the Finder. Hover the mouse over a clipping to see its contents. Tooltips for folders may be added just as for the Scripts palette (page 55). The Clippings palette can be displayed separately or as a tab of a combined palette. See also “Clippings Menu and Palette” on page 18.

Paste

Pastes the currently selected item into the frontmost code window.

Options popup

Toggles the display of icons in the palette, and shows the currently selected item in the Finder.

Tell Target palette

Displays the name of the application targeted by the innermost `tell` statement or `tell` block containing the insertion point in the frontmost code window, along with that application’s properties and elements. Behaves like a miniature, automatic version of the Explorer panel of the Dictionary window — see above, page 53. Double-click or drag an item name to paste a reference to that object into the frontmost code window. The Tell Target palette can be displayed separately or as a tab of a combined palette.

Update

Refreshes the currently selected item and any subitems. Shift-click to refresh all items. The palette also updates automatically any time the insertion point changes its `tell` context; the only way to prevent this is to close the palette.

Paste

Pastes a reference to the currently selected item into the frontmost code window.

Dictionary

Opens the dictionary window for the current target’s application.

NOTE: *If an application quits or crashes, the Tell Target palette may have difficulty locating the application when you place the cursor within a* `tell` *block. If this happens, run your script so that AppleScript tries to send an Apple event to the lost application; the Tell Target palette will then be able to display properly.*

Console

Permits snippets of code to be executed, and displays their results. Code is directed at the frontmost code window; this window’s handlers can be called, and its properties are available using the `its` keyword. See also “The Console” on page 28. The Console palette can be displayed separately or as a tab of a combined palette. On pretty-printing and line-wrapping, see “Script Result window” on page 51, above.

Language popup

Changes the OSA scripting language used in the palette.

Stop

Stops running the code.

Run

Runs the code for the line where the insertion point is, or the lines where the selection is. Alternatively, hit the Enter key.

Preferences

Preferences are set through the panels of the dialog that appears when you choose Preferences from the Edit menu (the Script Debugger menu in Mac OS X).

Universal buttons

These buttons appear on every panel.

OK

Applies preference changes and dismisses the dialog.

Cancel

Cancels preference changes and dismisses the dialog.

Factory Settings

Resets the current preference panel's settings to their factory value.

Revert Panel

Reverts the current preference panel's settings to their previous value.

General Settings panel

Startup Action

What should Script Debugger do when first launched? Your choices are to do nothing (no window opens), to create a new untitled document, or to present an Open file dialog.

Services

Toggles Frontier menu sharing. This is a facility which allows UserLand Frontier to display its own menu inside Script Debugger. Choosing an item from this menu runs a Frontier script (because the menu belongs to Frontier).

File Saving Options

When Script Debugger saves a file that it did not create, should it apply its own creator code (`'asDB'`), or preserve the existing creator? This will affect which application is launched when you later open the file from the Finder.

Scrolling Options

Toggles whether or not window content should move while the scrollbar "thumb" is being dragged.

Script Error Actions

This part of the panel deals with issues that arise because, even though a script is running in Script Debugger, some other application may be in front at the time an unhandled runtime error occurs in the script. Should Script Debugger then automatically come to the front? Should Script Debugger beep?

Enable Script Debugger Dictionary

Enables Script Debugger's own dictionary. If Script Debugger's dictionary is not enabled, other applications such as AppleScript will not realize that Script Debugger is scriptable, and won't be able to display its dictionary. The default is Off, and you can see this if you are studying a script in the Scripts folder which drives Script Debugger — if some of its objects appear as raw codes in angle brackets, that's because Script Debugger's dictionary is not enabled. However, you should enable the dictionary only when you are actively writing scripts that control Script Debugger, because its terms might conflict with (and block access to) those of some scripting additions. You must quit and re-launch Script Debugger for a change to this setting to take effect.

Record Script Debugger Actions

Allows Script Debugger to record its own actions.

Path To Me

The issue here is how scripts running in Script Debugger should interpret the phrase `path to me`. If it refers to the document, then the result is an alias to the script file. If not, or if the script has never been saved, then the result is an alias to Script Debugger.

Unhandled Events

The issue here is what happens when Script Debugger receives an unrecognized scripting command from outside (valuable if you want to test a named handler from outside Script Debugger (e.g. from a different application). If this option is off, it goes to the frontmost window, which might be a dictionary window. If this option is on, it goes to the first code window even if that window is not frontmost.

Background Activity

The issue here is what should happen during long compiles. This option makes long compiles even longer, but also lets you do other work while they are going on.

"Tell Variable" Constructs

This option addresses a rare issue where a very, very long script causes the debugger to choke. You shouldn't be writing scripts long enough to make this happen! But if you insist, then as a workaround, you can turn on this option, which reduces the amount of "instrumentation" Script Debugger performs, and lets much longer scripts work. The debugger, however, will then choke if it encounters a certain kind of `tell`, where the thing you're talking to is a list or record variable.

Pause Script

Turning this option off means that Script Debugger won't automatically pause before the first line of a script on which you're doing external debugging ("External Debugging" on page 44) when that script was already open for debugging in Script Debugger.

Trace Speed

Sets the speed for tracing ("Tracing and Code Coverage" on page 39).

Editor Settings panel

Auto Indent

If checked, Script Debugger indents a new line of code to match the previous line — it copies the initial tab-characters from this line of code to the beginning of the next when you press Return. You can hold down the Shift key when pressing Return to override.

Expression Tool-tips

Toggles the tooltip feature that evaluates expressions when the mouse hovers over them.

Drag & Drop

Toggles whether or not drag-and-drop text editing is enabled.

Smart Drag & Drop

Toggles whether or not Script Debugger should attempt to preserve correct spacing between words after drag-and-drop.

Bracket & Brace Matching

Toggles the feature that highlights matching brackets and braces as you type.

Highlight Delay

Lets you set the duration of brackets and braces highlighting.

Scroll if necessary

Toggles whether or not Script Debugger will scroll if required when highlighting a matching bracket or brace.

Projector Aware

Toggles MPW Projector support (see Appendix B, *Projector Support*).

Show OSA Formatting

Toggles the display of OSA script formatting. This is the styling whose details are set in the AppleScript Settings panel. Turning off script formatting results in unstyled text, but may improve performance when working on long scripts on slower machines.

Sort Table of Contents Menu

Toggles the order of the handler popup menu (page 50) between order of occurrence in the script and alphabetical order by name.

Syntax in Toc Menu

Deals with the handler popup menu again (page 50). The choice you're making here is for items in script objects, between "a's b" and "b of a", which is significant when alphabetical order is used.

Dropping Action

The question here is what Script Debugger should do when a file is dropped onto a code window (including a file alias, as in the Applications palette). The first possibility is to ask, meaning that Script Debugger will present a dialog that offers a choice of the remaining four possibilities: invoke the script's Open handler (if it has one) with the dropped file(s); paste a `tell` block to the dropped file (if it is an application); paste a reference to the file (an alias); or paste the file's contents (if a text file, compiled script, or script application). Even if you've chosen one of the four possibilities as the default preference, you can invoke the dialog by holding the Command key as you drop the file.

Background Color

Sets the background color for code windows.

New Script Defaults panel

All the settings in this panel can be changed elsewhere, at any time, for particular scripts. These are merely the values that a subsequently created code window will have at first (and only if the code window is not based on a template).

Line Wrapping

Toggles whether or not line wrapping will be turned on in new scripts.

Enable Debugging

Toggles whether or not new scripts will open in debugging mode.

Preferred Size

The default for the preferred partition size, in kilobytes, for script applications. Should be greater than or equal to the minimum size.

Minimum Size

The default minimum partition size, in kilobytes, for script applications.

Show Startup

Toggles whether or not new script applications should display their description in a startup screen when launched.

Stay Open

Toggles whether or not new script applications should keep running after their Open or Run handlers have completed.

Non-Persistent Settings

Toggles whether or not pausing at breakpoints, pausing at watchpoints, and pausing on exceptions are turned on. These settings apply also to existing scripts when initially opened from disk (because these features of a script are not saved with the script — that's what "non-persistent" means here).

Explorer / Browser Settings panel

These preferences affect windows in Browser format, including the Explorer window.

Show Item Count

Toggles whether lists and records are displayed as a count of their items or as a list of all their items.

Tool-tips

Toggles whether or not tooltips should appear when you hover the mouse over the description (left side of the browser window) and/or the value (right side).

Palette Window Settings panel

The primary issue here is whether you would like Script Debugger's floating windows, once opened, to disappear and reappear automatically depending on the type of non-floating window that is frontmost. The panel is self-explanatory — for each type of non-floating window, you get to say whether each floating window should become temporarily invisible when the non-floating window comes to the front.

Shorter Palette Names

Toggles whether or not names of floating windows should appear in abbreviated form on their tabs or titlebars.

AppleScript Settings panel

Provides access to AppleScript's global preferences, chiefly the styling of the text in compiled code. These are AppleScript features, not Script Debugger features. Thus, they are the same preferences accessed by the AppleScript Formatting dialog in the Script Editor. The Factory Settings button is disabled because Apple provides no way to return to the factory defaults (short of deleting the AppleScript preferences file and rebooting).

JavaScript Settings panel

Provides access to text styling preferences for JavaScript scripts, as implemented by Late Night Software's JavaScript OSA. At the moment all you get is font and size, but perhaps some day we'll add syntax coloring and so on.

Dictionary Settings panel

Text styling preferences for the content sections of the Dictionary and Object Model panels of the Dictionary window.

More Dictionary Settings panel

Scan for Elements

The issue here is what the Explorer panel should do when the target application doesn't support `count` for certain elements. You can have Script Debugger fall back on using `exists` in such cases. This is more time-consuming because Script Debugger must poll for each index one at a time. Also, it still might not work (as in Eudora). See Appendix A, *Issues With Scriptable Applications*.

List Inherited

Toggles inclusion of inherited properties and elements in class descriptions. For example, if you turn this off, then the Finder's `file` class description omits properties and elements inherited from the `item` class.

Use Terminology Extensions

Toggles whether or not terminology plugins are enabled. See Appendix A, *Issues With Scriptable Applications*.

List & Record Items

The issue here is whether, when you expand in the Explorer an object whose default value is a list or record, you should be shown, in addition to the object's properties and elements, the items of that list or record. For example, in FileMaker, expanding `record 1` would show, in addition to the record's `cells`, `best type`, `class`, and so forth, also its `item 1`, `item 2`, and so on. The problem is that this information is somewhat misleading — you can't actually `get item 1 of record 1`, for example — so it is usually best suppressed, which is what this preference lets you do.

Show Comments

Toggles whether or not comment information from the dictionary should appear as tooltips when the mouse hovers over an object name in the Explorer panel.

Open Dictionary Menu

Lets you add and remove applications manually from the list that appears in the Open Dictionary submenu of the File menu and in the Applications palette.

Automatically Add

Toggles the feature that automatically adds applications to the list when their dictionary is first opened.

Automatically Remove

Toggles the feature that automatically removes an application in the list when there is an error in opening its dictionary (for example, because the application has been deleted).

HTML Export Settings panel

Make URLs in Scripts Clickable

If chosen, URLs in scripts are treated as live links in the exported HTML.

Formatting

You get to choose between HTML and CSS (style-sheet) formatting. The issues are a bit complicated, but briefly, HTML formatting does a pretty accurate job of representing your script, but it uses tags that are increasingly deprecated as HTML evolves. CSS formatting is the newer way, and is capable of greater subtlety, but it isn't interpreted correctly by all browsers. Observe that you have far more control over the details of HTML export formatting than is shown in this preference panel. The formatting is actually controlled by an XML file called HTML Export Templates which is located in the Plugins folder, and you can open this file with BBEdit or your favorite XML editor or text editor and customize the HTML templates.

Menus

File menu

New Script

Opens a new code window. If there is a Default Script stationery template in the same folder as Script Debugger, it is copied (see "Templates" on page 19). Otherwise, settings from the New Script Defaults preferences panel are used (see "New Script Defaults panel" on page 60).

New

Opens a new code window as a copy of a stationery file in the Templates folder (see "Templates" on page 19).

Open

Brings up the standard file Open dialog, where you can open an existing text file, compiled script, or script application (as a code window), or a scriptable application (as a dictionary window).

Open Dictionary: Scripting Additions

Opens a dictionary window listing all installed scripting additions.

Open Dictionary: Application

Brings up the standard file Open dialog, where you can choose a scriptable application on disk to open its dictionary. In Mac OS X, brings up the Launch Services browser listing all applications. You only need this if you've never opened the application's dictionary before, since after that it appears in the Applications palette and you can open the dictionary from there (see "Applications palette" on page 55). There are other ways to open an application's dictionary: use Open, as just mentioned above; or drag an application's icon onto Script Debugger's icon in the Finder (or the Mac OS X Dock).

Open Dictionary: Running Application

(Not available under Mac OS X.) Brings up the Mac OS PPCBrowser dialog, where you can choose an application running on this or a remote computer to open its dictionary. Remote applications appear only if Program Linking is turned on for that computer and if Sharing has not been turned off for that application. A remote application which

appears in this list can be used as the target of Apple events. You can drive it through AppleScript, and you can change its values through the Explorer panel (please read the warning on page 21).

Open Dictionary: New Explorer

Opens a new separate Explorer window on the application targeted by the frontmost Dictionary or Explorer window, starting at an object reference that you provide in a dialog. (For an example, see page 25.)

Open Dictionary: Change Explorer

Like New Explorer, except that instead of opening a new window, changes the base object reference of the frontmost dictionary Explorer window. Available only when this window was generated with New Explorer. Same as the Change button in the Explorer window.

Open Dictionary: Update

Prompts the frontmost Explorer window to probe its target again. Same as shift-clicking the Update button in the Explorer window.

Open Dictionary: Application List

Opens the dictionary of a listed application. This is the same list as in the Applications palette, and the effect is the same as double-clicking an item in the Applications palette.

Open Recent

Opens a recently opened document.

Open Viewer

Opens a separate Viewer window for the currently selected browser item. Same as double-clicking the item. In the case of an element in an Explorer window, the new window is an Inspector window, and this menu item is the same as choosing New Explorer (or Change Explorer) and providing a reference to that item.

Close

Closes the current window.

NOTE: *You cannot close a code window while the script is running or paused. If you try to close a running script, Script Debugger will offer to stop the script first.*

Close All (Option)

Closes all code and dictionary windows.

Close All Scripts (Shift)

Closes all code windows.

Close All Dictionaries (Option-Shift)

Closes all dictionary windows.

Save

Saves the frontmost code window to disk (if “dirty”).

Save All (Option)

Saves all “dirty” code windows to disk.

Forced Save (Option-Shift)

Saves the frontmost code window to disk, even if it isn’t “dirty”.

Save As

Saves the current document in a new file. Besides saving from a code window, also permits saving contents of the Script Result window, the Log window, or a Dictionary window (the name of the menu item will change appropriately).

Save A Copy As

Saves the frontmost code window in a new file, but leaves the window pointing to the old file.

Revert to Saved

Opens the frontmost code window’s file from disk, throwing away changes made since the window was last saved.

Import Recovered Script

Opens a compiled script using the embedded source code instead of decompiling the compiled code. This can be done only when the script was saved by Script Debugger (because otherwise there is no embedded source code), and is intended chiefly as an emergency measure when decompilation of the compiled code is impossible or undesirable.

Export Run-Only

Saves a copy of the frontmost code window as a new file without source code.

WARNING: *A Run-Only script file cannot be edited. Save normally if you intend to edit this script ever again.*

Export Flattened Script

Saves a copy of the frontmost code window as a new file incorporating any referenced library files into the code (see page 31). Disabled if the window has no attached libraries.

Export HTML

Saves a copy of the frontmost code window as a new HTML file. The exact HTML used depends upon the HTML Export Templates file in the Plugins folder. This file contains two pairs of templates, and which pair is used depends upon your preference settings (see “HTML Export Settings panel” on page 62).

Edit With BBEdit

Opens the frontmost code window for editing with BBEdit (see “External Editor” on page 17).

Manifest

Opens the Manifest window, describing the frontmost code window in terms of what applications and scripting additions it targets (see “The Manifest” on page 28).

Add Library

Attaches a library to a code window. Same as clicking the plus-icon in the window’s Libraries tab (see “Libraries” on page 30).

Modify Read-Only

When editing scripts managed using the Projector source code control system, makes read-only scripts modifiable. Applied to the file only when changes are saved. Thus, as long as you don’t save changes, Revert To Saved will undo this command. (See Appendix B, *Projector Support*.)

Modify Read-Only (Shift)

Identical to the preceding but bypasses the confirmation dialog.

Page Setup

Brings up the Page Setup dialog, where you can change printer settings.

Print

Brings up the Print dialog, where you can print the current document.

Print Diagram

Brings up the Print dialog, where you can print the current dictionary diagram from the Object Model panel. You do not actually have to be in the Object Model panel (but you must be in a dictionary window).

Quit

Quits Script Debugger. Any open documents will be closed. You will be prompted to confirm the saving of any changes you have made. On Mac OS X, this menu item is called Quit Script Debugger and appears in the Script Debugger menu.

WARNING: *You cannot quit Script Debugger (nor, therefore, can you restart or shut down your computer) if any code window is running or paused. If you try to do so, Script Debugger will offer to stop the running code window first.*

Edit menu

Undo / Redo

Undoes or redoes (after undoing) the last operation.

Cut

Deletes the current selection and copies it to the clipboard.

Copy

Copies the current selection to the clipboard. When the current selection is an object in the Explorer, the Properties tab, or any other object browser, copies a reference to the object.

Copy Event Template

When the current selection is an event in a dictionary window, copies a template for the event.

Copy Handler Template (Shift)

When the current selection is an event in a dictionary window, copies a template for the event, wrapped in a handler block. The idea is that you could customize the event, overshadowing it with this handler, which calls the event but also adds your own code. This is particularly useful when writing an attachment handler, e.g. for FaceSpan or for Script Debugger itself.

Paste

Inserts the contents of the clipboard into your document, replacing the current selection.

Paste Reference

Inserts into your document a reference to the object placed on the clipboard by a script. For example, after telling the Finder to `set the clipboard to disk 1`, what is on the clipboard is not text but an object reference; hence it cannot be pasted in any ordinary sense. This menu item, however, allows you paste the phrase “disk 1” into a script.

Clear

Deletes the current selection.

Select All

Selects all text or all items in a list.

Balance

Expands the text selection to include the enclosing parentheses, curly braces, square brackets, or AppleScript block structure (such as an `if` block, `repeat` block, `try` block, `tell` block, and so forth), or beeps if an unbalanced structure is detected.

Font, Size, Style, Color

Changes styling of selected text. Applicable only in the Description tab of a code window; styling of code and dictionary text is set through preferences (see page 61).

Preferences

Brings up a dialog where you can change Script Debugger and AppleScript preferences. See “Preferences” on page 57. Under Mac OS X, this menu item appears in the Script Debugger menu.

Set Menu Keys

Brings up a dialog where you can change the keyboard shortcuts for all menu items. You can use any key with any combination of modifiers that includes Command or Control. You can also use Function keys (F5 and up, since F1–F4 are reserved). These don’t require a modifier, and, if you include a modifier, do not require that Command or Control be included. Under Mac OS X, this menu item appears in the Script Debugger menu. Observe that keyboard shortcuts for clippings and scripts, shown in the  menu and the  menu, are changed using the Clippings and Scripts palettes, respectively (page 55).

Search menu

Find

Brings up the Find dialog. The Find dialog is the *only* place where you can set the find options, such as wrapping and whole-word searching, which determine precisely how all the other Find and Replace commands will behave. See also “Searching for Text” on page 18.

Find Backwards (Shift)

Brings up the Find dialog with the Backwards checkbox checked.

Find Again

Finds the next occurrence of the contents of the Find dialog’s Search For box, without bringing up the Find dialog.

Find Again Backwards (Shift)

Finds the previous occurrence of the contents of the Find dialog’s Search For box, without bringing up the Find dialog.

Enter Search String

Copies the current selection into the Find dialog's Search For box, without bringing up the Find dialog.

Enter Replace String (Shift)

Copies the current selection into the Find dialog's Replace With box, without bringing up the Find dialog.

Find Selection

Copies the current selection into the Find dialog's Search For box and finds the next occurrence, without bringing up the Find dialog.

Find Selection Backwards (Shift)

Copies the current selection into the Find dialog's Search For box and finds the previous occurrence, without bringing up the Find dialog.

Replace

Replaces the current selection with the contents of the Find dialog's Replace With box, without bringing up the Find dialog.

Replace & Find Again

Replaces the current selection with the contents of the Find dialog's Replace With box, and then finds the next occurrence of the contents of the Find dialog's Search For box, without bringing up the Find dialog.

Replace & Find Again Backwards (Shift)

Replaces the current selection with the contents of the Find dialog's Replace With box, and then finds the previous occurrence of the contents of the Find dialog's Search For box, without bringing up the Find dialog.

Replace All

Replaces all occurrences of the contents of the Find dialog's Search For box with the contents of the Find dialog's Replace With box, without bringing up the Find dialog.

Go To Line

Brings up a dialog where you can enter a line number to jump to. Same as clicking on the status area of the code window.

Go To Next Handler

Jumps to the next handler definition in the code window.

Go To Previous Handler

Jumps to the previous handler definition in the code window.

Script menu

The first eight items in the Script menu are available also as buttons on the Controls palette and/or in the code window. See “Controls palette” on page 53.

Run

If your script is already compiled, runs it; otherwise, attempts to compile it, and if successful, runs it. During debugging, proceeds to the next breakpoint or watchpoint.

Run To Cursor (Option)

In debugging, proceeds to just before the line containing the insertion point or the start of the selection.

Trace (Shift)

In debugging, traces through the script (see “Tracing and Code Coverage” on page 39).

Trace To Cursor (Shift-Option)

In debugging, traces to just before the line containing the insertion point or the start of the selection.

Stop

Stops your script running or debugging.

Pause

Pauses your script.

Step Over

In debugging, executes the next statement. If the statement invokes a handler, all the statements of that handler are executed (unless a breakpoint or watchpoint is encountered).

Step Into

In debugging, executes the next statement. If the statement invokes a handler, execution pauses at the first statement of the handler.

Step Out

In debugging, executes the balance of the current handler (unless a breakpoint or watchpoint is encountered first), pausing after the handler completes.

Record

Begins recording in your script the actions of other applications (if recordable).

Compile

Compiles your script.

Set/Clear Breakpoint

Sets or clears a breakpoint on the line containing the current selection. Same as clicking in the diamond to the left of the line.

Clear All Breakpoints

Removes all breakpoints.

Breakpoints Enabled

Toggles whether or not breakpoints should trigger a pause when encountered during debugging. Applies only to the script that is currently frontmost. See also “Non-Persistent Settings” on page 61.

Break On Exceptions

Toggles whether or not the first executable statement of an `on error` block should be treated as if it were a breakpoint. Applies only to the script that is currently frontmost. See also “Non-Persistent Settings” on page 61.

Clear All Watchpoints

Removes all watchpoints.

Watchpoints Enabled

Toggles whether or not watchpoints should trigger a pause when encountered during debugging. Applies only to the script that is currently frontmost. See also “Non-Persistent Settings” on page 61.

New Expression

Opens the Expressions window if necessary, and creates a new expression, selected and ready for you to type the expression. Same as clicking the plus-icon in the Expression window. See “Expressions” on page 40.

Copy To Expressions

Opens the Expressions window if necessary, and adds to it an expression copied from the currently selected text.

Clear All Expressions

Clears all expressions from the Expressions window.

Clear Coverage Marks

Removes the blue coverage marks. See “Tracing and Code Coverage” on page 39.

Show Last Result

Sets the contents of the Script Result window to the result from the last time the frontmost code window was run. Identical to selecting the script from the Script popup menu in the Script Result window. Does *not* bring the Script Result window to the front.

Show Last Error

Brings up again the dialog which previously announced the current script’s last compile-time or runtime error. Identical to clicking on the red arrow to the left of the error line.

Enable / Disable Debugging

Toggles the frontmost code window between normal and debugging modes.

Options: Applet Memory Partition

For code windows designated as script applications, lets you change the preferred and minimum memory partitions for the script application. (The default size is set in the New Script Defaults preferences panel; see page 60.)

Options: Stay Open

For code windows designated as script applications, toggles whether or not the application keeps running after its Run handler executes. Same as the Stay Open button in the code window.

Options: Show Startup Screen

For code windows designated as script applications, toggles whether or not your script’s description is shown when the application is launched. Same as the Show Startup Screen button in the code window.

Options: Build Script Context / Build Compiled Script

A highly technical feature of the OSA, and best left alone unless you know exactly what you’re doing. In AppleScript, you always want what are called Script Contexts; building Compiled Scripts will almost certainly break your script. Other OSA dialects and runtime environments may apply different meanings to these options, so the choice is provided just in case.

Options: Default Target Application

By default, AppleScript sends undirected Apple events (those appearing outside any `tell` block) to the application running the script. When editing scripts in Script Debugger, this will be the Script Debugger application. If you want to simulate the runtime environment of some other application, you can use this menu to make that application the target for undirected Apple events. The new target application's dictionary will then be used to resolve names of events, classes and properties appearing outside a `tell` block. Choose Current Application to return to the default (choosing Script Debugger would have a similar effect, but scripts would run slower because AppleScript would not generate send-to-self Apple events).

Options: Parent Script

Each AppleScript script has a parent script used to resolve references to otherwise undefined variables and handlers. By default, AppleScript assigns its own global script object as the parent of all scripts. This menu allows you to override that default and make any open script the parent of the current script. This is useful for simulating other runtime environments that maintain AppleScript parent script relationships. Choose None to return the script to its default setting. Closing a script's designated parent script window also returns the script to its default setting.

Windows menu

Palettes

Toggles the visibility of the Controls, Windows, Applications, Scripts, Clippings, Tell Target, and Console palettes. If a palette is part of a combination palette, showing it is the same as selecting its tab in the combination palette, but hiding it is the same as dismissing the entire combination palette.

Zoom Window

Zooms the frontmost window. Same as clicking in the window's zoom box.

Collapse Window

(Mac OS X only.) Minimizes the window into the Dock. Same as clicking the window's minimize button.

Send To Back

Makes the frontmost window the rearmost window; the window that was previously second is now frontmost.

Exchange With Next

Makes the frontmost window the second window; the window that was previously second is now frontmost.

Bring All Windows To Front

(Mac OS X only.) Brings all of Script Debugger's windows in front of the windows of all other applications.

Set Default Window Size

Makes the size of the frontmost code window the default for the initial size of any subsequently created completely new code windows. (But in the case of code windows opened from a saved script or from a template, the file dictates the window's initial size.)

Reset Default Window Size

Returns the size of subsequently created code windows to the factory setting.

Script Result

Summons the Script Result window.

Apple Event Log

Summons the Apple Event Log window.

Expressions

Summons the Expressions window.

Properties & Globals

Summons the Properties & Globals window.

Window list

Lists all open windows. Same as the list in the Windows palette. Choosing a menu item brings the window to the front.

 menu

Lists the folders and clippings from the Clippings folder. Choosing a clipping pastes it into the frontmost code window. Option-choosing a clipping reveals it in the Finder. These are the same items that appear in the Clippings palette. See "Clippings palette" on page 55.

 menu

Lists the folders and scripts from the Scripts folder. Choosing a script runs it. Option-choosing a script opens it for editing. These are the same items that appear in the Scripts palette. See "Scripts palette" on page 55.

Help menu

(Under Mac OS X, there is no Help menu. There is presently no Balloon Help, so the Show/Hide Balloons menu item is absent; the other two menu items discussed here appear in the Script Debugger menu.)

Show/Hide Balloons

Toggles visibility of help balloons as you hover the mouse over an interface item.

Send Us Email

Creates a new message to Late Night Software technical support, using your email application.

Visit Our Web Site

Tells your Web browser to visit the Late Night Software Web site. You should be connected to the Internet before choosing this item.

Issues With Scriptable Applications

This appendix describes certain unpleasant or surprising interactions between Script Debugger and some scriptable applications, and offers workarounds for many of them.

The Explorer Panel

The Dictionary window's Explorer panel does not work equally well with every scriptable application, for the simple reason that not all scripting interfaces are created equal. Some applications provide incomplete dictionary information; that is to say, their internal dictionary simply lies about the application's supported classes or properties, or about what you are and are not allowed to say to the application when scripting it. Also, the Explorer probes more or less the entirety of a scriptable application's scripting interface, which may expose problems in the application's scripting implementation which would otherwise come to light only rarely.

Script Debugger's Explorer includes a facility called Terminology Plugins to adapt to certain kinds of deficiencies in dictionary information. You can visit our Web site, http://www.latenightsw.com, for new Terminology Plugins and for updates to this document. Observe that Terminology Plugins won't work if you've turned off terminology extensions in the More Dictionary Settings preference panel.

Here are some issues between the Explorer and particular applications.

Mac OS Finder Before 8.5

The Tools & Goodies folder contains a Terminology Plugin for the Finder. If you are using a version of the Finder *before Mac OS 8.5*, you should move this into the Plugins folder and restart Script Debugger. This plugin extends the Finder dictionary to reflect more accurately the true structure of the Finder scripting interface. **Do *not* use this plugin with Mac OS 8.5 or later!** With the release of Mac OS 8.5, Apple reworked the Finder's dictionary. These corrections obviated the need for the Finder Terminology Plugin.

For example, the `application file` class inherits all the properties of the `file` class, which in turn inherits all the properties of the `item` class. Similarly, the `folder` class inherits all the properties of the `container` class. The Finder's dictionary resource, before Mac OS 8.5, fails to reveal correctly these facts, and many others like them. The Finder Terminology Plugin fixes this problem.

While the Finder Terminology Plugin corrects many problems in the Finder's dictionary information, it does not correct omissions in the key forms supported by the various Finder classes. Specifically, the Finder dictionary indicates that ranges of elements cannot be specified, and that `whose` clauses cannot be used. In fact, however, it is possible to use ranges and `whose` clauses with most Finder classes. To build range and `whose` object specifiers in the Dictionary Explorer, hold down the Option key when clicking on the object specifier popup menu icon (this trick was mentioned under "Explorer panel" on page 53).

Claris Emailer 2.0

We have found that Emailer 2.0 hangs when it is asked to count the number of attachments of an outgoing message that has no attachments.

FileMaker Pro

Included with Script Debugger is a Terminology Plugin for FileMaker Pro. This plugin is already present in the Plugins folder. It extends the FileMaker Pro dictionary to reflect more accurately the true structure of the FileMaker Pro scripting interface.

In particular, the plugin reveals that it is possible to access records, fields and cells from the application class. (These elements are taken to refer to the frontmost window.) It also extends the definition of the `database` and `document` classes so that all the properties accessible through these classes are properly listed.

Microsoft Excel and Word

Late Night Software recommends that you *not* use the Explorer with Microsoft Excel or Microsoft Word. You can sometimes get away with this, but in our testing we have found that this can lead to crashes. For example, Excel crashes whenever it is asked to count the number of Add-in elements in its application class (and of course this is something that the Explorer does ask Excel to do, since it inquires about all elements of every class).

The Dictionary and Object Map panels are not affected by these bugs in Excel and Word, and may be used without risk.

Observe that this problem is Microsoft's, not Script Debugger's. You can just as easily crash Excel or Word by trying to script them with any other application, such as the Script Editor or Frontier.

Eudora

Eudora's scripting is not implemented in a way that supports tools like the Dictionary Explorer. The trouble is that Eudora does not support counting of elements, and does not support indexed access to many element hierarchies. This makes it impossible for the Explorer to display the Eudora scripting interface automatically. (This problem was discussed on page 24, and a workaround was demonstrated on page 25.)

Applications That Aren't Running

When you use Script Debugger to work on scripts that speak to other scriptable applications, it's best if those applications are already running. Otherwise, some mildly annoying things can happen when you compile your script, or when you open a compiled script that was saved earlier. This section discusses these mildly annoying things.

Where Is...?

When you compile a script containing a `tell` block for an application that isn't running, the "Where Is...?" dialog sometimes appears, asking you to locate the application on your hard drive. This is particularly galling when the application appears in your Applications palette. You know that Script Debugger knows where the application is, because its presence in the Applications palette means that Script Debugger has an alias to it; so why is Script Debugger asking you where it is?

The answer is that it isn't Script Debugger that's asking you this question; it's AppleScript. AppleScript maintains its own list of known applications, and if you refer to an application that isn't on this list, AppleScript doesn't turn to Script Debugger and ask where it is — it turns to you, the user, instead.

If this drives you crazy, there's a workaround: copy aliases of frequently used applications into your Scripting Additions folder (in the System Folder). AppleScript looks there before giving up and consulting the user directly.

Automatic Launch

When you use Script Debugger to compile a script, or to open a compiled script, and that script contains a `tell` block for an application that isn't running, it may happen that the application launches all by itself. This makes sense if you actually run the script — obviously an application must be running before you can talk to it — but it's clear that in this case the application is launching merely because Script Debugger is consulting its dictionary. This seems wrong, especially since Script Editor can consult an application's dictionary without launching the application.

The reason for this behavior, if you encounter it, is complicated and technical. It turns out that AppleScript allows applications to provide their dictionary data in two ways. *Static dictionaries* can simply be specified in an `'aete'` resource, which AppleScript reads directly. Applications that have *dynamic dictionaries* (because, for example, they allow plugins to extend their scripting interface) can request that AppleScript read their dictionary by sending an Apple event. In this latter case, the application must obviously be running in order to respond to the Apple event.

Now, all of these applications have static dictionaries; so what, you may ask, does the distinction between static and dynamic dictionaries have to do with the problem we're discussing? The trouble is that Script Debugger itself has a dynamic dictionary — and there's a bug in AppleScript, such that this fact causes it to mistakenly consider *all* applications referred to by Script Debugger to have dynamic dictionaries too. In the case of AppleScript 1.6, we have not found a way to work around this bug in AppleScript.

To be sure, the problem is not a very serious one, since you were probably going to launch the application anyway. Plus, the problem shouldn't arise in versions of AppleScript earlier than 1.6, and may well be fixed in later versions. Nevertheless, it's something to be aware of if you're wondering how all these applications suddenly got running on your computer without your starting them up.

APPENDIX B

Projector Support

MPW (Macintosh Programmer's Workshop) is a free development environment available from Apple Computer. One aspect of MPW is Projector, an integrated collection of tools and scripts for managing project source files; Projector regulates users' access to source files by requiring that they check the files in and out of the Projector database, and it maintains revisions and comments to the source files. If you like, you can manage your scripts through Projector. This Appendix explains how Script Debugger reflects your use of Projector.

To obtain MPW or to learn more about it, see http://developer.apple.com/tools/mpw-tools/. Note that this discussion does not explain how to check files in and out of Projector! For that information, you should refer to the documentation that came with your copy of CodeWarrior or MPW, or see http://developer.apple.com/tools/mpw-tools/commandref/index.html.

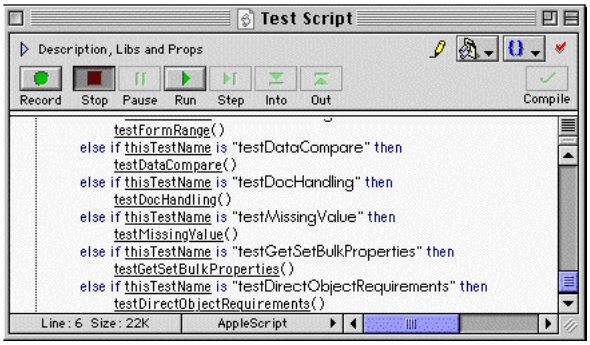
Observe that Script Debugger won't respond to your use of Projector if you've turned off the Projector-aware option in the Editor Settings preference panel.

Script Debugger indicates the Projector state of the script with one of three icons in the script's header. A script can be either checked out for modification, checked out as read-only, or checked out as modifiable read-only.

Modification

If your script file is managed by Projector and you have checked it out for modification, the script header contains an additional icon; it is a pencil drawing a line (Figure B-1).

Figure B-1
Script Checked Out for Modification

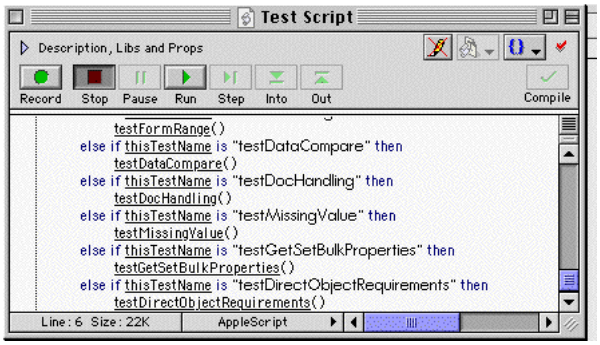


After you have made modifications to the script, you will check it back into Projector's database.

Read-Only

If your script file is managed by Projector and you have checked it out as Read-Only, the Pencil icon in the header has an X through it (Figure B-2).

Figure B-2
Read-Only Script



In this case, Script Debugger treats the file as though it were locked. You cannot make changes to the contents of the file. For instance, the Save menu is disabled to prevent you from accidentally making any changes to the file.

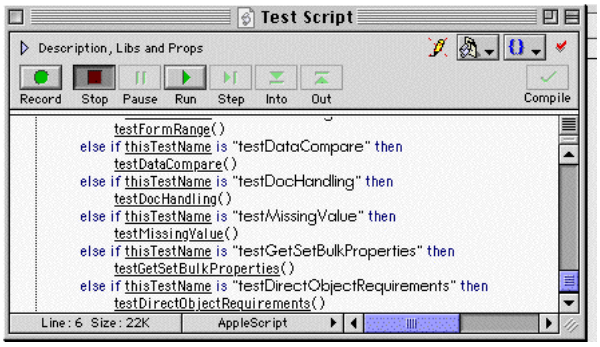
If you decide to make changes to the file, you can change its state to Modifiable Read-Only. To do so, click on the Pencil button in the script header, or choose Modify Read-Only from the File menu. If a file is Modifiable Read-Only, Projector allows you to make changes to it without checking it out of the Projector database. You do have to check the changes into Projector at a later time.

Script Debugger makes permanent the change in the file's status only when you save it. If you have not yet saved the file, you can reverse the change to Modify Read-Only status, by choosing Revert To Saved from the File menu.

Modifiable Read-Only

If a script file has been checked out of Projector as Modifiable Read-Only, the Pencil icon in the script header has a dimmed X through it (Figure B-3).

Figure B-3
Modifiable Read-Only Script



In this case, you can make changes to the file and then check them into Projector's database when you are finished.

APPENDIX C

AppleScript Information

This appendix provides pointers to sources of information about AppleScript — Apple's manuals, other books, and various online resources.

Apple Documentation

The *AppleScript Language Guide* is Apple Computer's authoritative document describing the AppleScript language. It is quite readable and highly recommended. Apple Computer makes this document available online in two forms: HTML and PDF.

HTML: http://developer.apple.com/techpubs/macos8/InterproCom/AppleScriptScripters/AppleScriptLangGuide/index.html

PDF: http://developer.apple.com/techpubs/macos8/pdf/AppleScriptLanguageGuide.pdf

The following technical notes document recent changes in AppleScript:
http://til.info.apple.com/techinfo.nsf/artnum/n88018
http://til.info.apple.com/techinfo.nsf/artnum/n120001

See also Apple's guides to scripting the Finder and the Scripting Additions that come with AppleScript:

http://developer.apple.com/techpubs/mac/AppleScriptFind/AppleScriptFind-2.html

http://developer.apple.com/techpubs/mac/scriptingadditions/ScriptAdditions-2.html

Books

Perry, Bruce W. *AppleScript in a Nutshell*. O'Reilly & Associates, 2001 (ISBN 1-565-92841-5). $29.95
http://www.oreilly.com/catalog/aplscptian/

Wilde, Ethan. *AppleScript for the Internet*. Peachpit Press, 1998 (ISBN 0-201-35359-8). $17.95
Wilde, Ethan. *AppleScript for Applications*. Forthcoming. Peachpit Press, 2001 (ISBN 0-201-71613-5). $19.99
http://www.peachpit.com

Trinko, Tom. *AppleScript for Dummies*. IDG Books, 1995 (ISBN 1-568-84975-3). $19.99.
http://www.hungryminds.com/

Goodman, Danny. *The Complete AppleScript Handbook*. 2nd ed. Random House, 1995; reprint (ISBN 0-966-55141-9). $34.95
http://www.iuniverse.com/marketplace/bookstore/book_detail.asp?isbn=0%2D96655%2D141%2D9

The following books are now out of print, but may be available at second hand:

Michel, Steve. *Scripting the Scriptable Finder*. Heizer Software, 1995. $49.00

Schneider, Derrick. *The Tao of AppleScript*. 2nd ed., 1994 (ISBN 1-56830-115-4). $24.95

Trinko, Tom. *Applied Mac Scripting*. MIS:Press, 1995 (ISBN 1-55828-330-7). $34.95

Web Sites

Apple Computer provides an extensive Web site devoted to AppleScript, with sample scripts, tutorials, and a range of other AppleScript-related information:
http://www.apple.com/applescript/

Late Night Software maintains a series of pages on its Web site containing references to online information about AppleScript:
http://www.latenightsw.com/scripting.html

The ScriptWeb pages provide valuable links and archives:
http://www.scriptweb.com/

MacScripter.net is a major source of AppleScript news, scripting additions, scripts, and useful links:
http://www.MacScripter.net/

The AppleScript Sourcebook is another valuable and instructive source of links and information:
http://www.AppleScriptSourcebook.com/frames.html

Mailing Lists

The MacScripting mailing list is devoted to the discussion of AppleScript, Frontier, and other OSA scripting languages. You can subscribe by addressing a message to listserv@dartmouth.edu. The message should contain the line

```
sub macscrpt <your name>
```

Replace the `<your name>` with your real name. You will be added to the mailing list and will receive information about changing your mail options and unsubscribing.

Apple Computer also maintains an AppleScript Mailing list:
http://www.lists.apple.com/mailman/listinfo/applescript-users

Applications Mentioned in This Manual

BBEdit
http://www.bbedit.com/products/bbedit.html

Clip2Gif
ftp://mathforum.com/software/workshops/mac/clip2gif.sea.hqx

Eudora
http://www.eudora.com/

Excel
http://www.microsoft.com/mac/products/office/2001/excel/

FileMaker Pro
http://www.filemaker.com/products/fm_home.html

Frontier
http://frontier.userland.com/

HyperCard
http://www.apple.com/hypercard/

OneClick
http://www.westcodesoft.com/oneclick/ocinfo.html

OSAMenu
http://www.lazerware.com/software.html

QuarkXPress
http://www.quark.com/products/xpress/

REALbasic
http://www.realbasic.com/realbasic/about/index.html

Script Editor
included as part of the AppleScript installation

StuffIt Expander
http://www.aladdinsys.com/expander/index.html

Tex-Edit
http://www.nearside.com/trans-tex/

Word
http://www.microsoft.com/mac/products/office/2001/word/

APPENDIX D

Scripting Script Debugger

Script Debugger is recordable, scriptable, and attachable. This means that Script Debugger can be very heavily automated, customized, and extended. This appendix provides a brief introduction to these aspects of Script Debugger.

Observe that in order to develop a script that drives Script Debugger, it will be useful to have access to Script Debugger's dictionary. This dictionary is dynamic, and by default isn't loaded, meaning that AppleScript won't be able to produce the dictionary, and that scripts that drive Script Debugger will appear in terms of raw Apple events, without translation into English-like AppleScript terms. So you might want to start by turning on Script Debugger's dictionary, and probably its recordability as well. Use the first two Script Options in the Scripting Settings preference panel. You'll have to quit and restart Script Debugger to get AppleScript to register your changes.

Recordable

The fact that Script Debugger is recordable means that actions you perform manually can be translated for you into the AppleScript commands you would have had to give in order to drive Script Debugger to perform those same actions. Obviously this can be very useful if you're developing a script to automate Script Debugger.

To record Script Debugger, press the Record button in Script Debugger, the Script Editor, or any other application that knows how to register the actions of recordable applications, and perform some actions in Script Debugger. When you're done, stop recording in the application where you started recording.

Scriptable

Script Debugger is very heavily scriptable, as you'll see from examining its dictionary. To get a look at some example scripts that drive Script Debugger, just option-choose any of the scripts in the Scripts palette.

If you do this, you'll discover that although scripting Script Debugger is not difficult, there's a lot more to think about than you might have suspected. Consider, for instance, the first of the Editing Tools scripts, Comment Selection. You might have thought that nothing could be simpler than to insert comment delimiters at the start and end of the current selection — just substitute for the currently selected text the same text preceded and followed by comment delimiters. And indeed, the action of inserting comment delimiters is itself simple. We do have to worry about what language the script is in, though (it wouldn't do to insert AppleScript comment delimiters into a JavaScript script). There is also a slightly tricky bit about whether we need to insert a return character.

But more important, this entire action is wrapped in a number of tests corresponding to various states in which Script Debugger might find itself at the moment the user runs this script. Is the frontmost window a non-script window, such as a dictionary? If it is a script, is it in the middle of debugging? In either case, we can't proceed. This sort of consideration can call for some careful planning. You may find it useful to model your script on the existing scripts provided.

Another thing to notice is that, when developing a script, if you're working within Script Debugger itself, you can assume Script Debugger as your context. For example, suppose we wish to tell Script Debugger to open the Apple Event Log window. From a different program, such as Script Editor, we can say:

```
tell application "Script Debugger"
   open AppleEvent log
end tell
```

And we can do the same thing from within Script Debugger. But you're already in Script Debugger, so you don't really need a `tell` block; Script Debugger is the parent of the script. You'll notice that the scripts in the Scripts palette don't have a `tell` block targeting Script Debugger; that's because Script Debugger is already the target. So, you can remove the surrounding `tell` block, and run the following script from within Script Debugger:

```
open AppleEvent log
```

As an example of a rather more significant use of scripting Script Debugger, here's a little script that lets the user pick a folder of scripts and saves each of those scripts as run-only:

```
set whatFolder to choose folder ¬
   with prompt "Folder of scripts to convert:"
tell application "Finder" to ¬
   set whatFiles to every file of whatFolder whose file type is "osas"
if length of whatFiles = 0 then return
repeat with aFile in whatFiles
   set aDoc to open (aFile as alias)
   save aDoc in file ((aFile as string) & "RO") with run only
   close aDoc
end repeat
```

As you develop a script, if you're working within Script Debugger itself, watch out for circumstances that might adversely affect the environment. Returning, for instance, to the Comment Selection example, you'll see that it operates on the first document. But if you were developing this script yourself, and you pressed the Run button in the Comment Selection script window, the Comment Selection script window *is* the first document, and a running script can't modify itself; so you can't test in this way. You may wish to wrap your script in some lines that juggle the environment for purposes of testing.

For example, suppose we intend to append "Hi there" to the frontmost script's text:

```
copy "Hi there" to after text of window 1
```

The script from which this command is given is the frontmost script, so we can wrap the script in lines that prevent this from being the case:

```
move script document 1 to end
copy the result to myself -- get a reference to this window
copy "Hi there" to after text of window 1
move myself to beginning
```

However, there's a cooler and much less onerous way to accomplish the same thing — namely, to take advantage of the fact that Script Debugger is attachable. That's the subject of the next section.

Attachable

Users of AppleScript are accustomed to applications being scriptable and, to a lesser extent, recordable; but attachability is so rarely seen that it may be new even to longtime scripters. An application is attachable if it gives the user an opportunity to interfere with its native actions — to customize them, to extend them, even to prevent them. Script Debugger is completely attachable, and then some: all of its scriptable commands can be customized, and so can some extra functions that are not represented by any scriptable command.

It's important to understand that what you're customizing when you "attach" a script to Script Debugger is not merely its behavior when it's being driven through AppleScript; you're customizing all of its behavior. That's because Script Debugger performs all of its actions, including when you're working with it manually through its interface of windows and menus, by sending itself AppleScript commands (this, in fact, is what makes Script Debugger recordable).

To attach a script to Script Debugger, open the Attachments file in the same folder as Script Debugger and insert a handler. The name of the handler is, generally speaking, the command from Script Debugger's dictionary that you wish to customize. Once the Attachments file is compiled and saved, the attachment is made; you don't have to close it.

The trick is that in order to obtain Script Debugger's default functionality for a command that you're modifying, you must at some point during the handler `continue` the command. As an example, we'll change what Script Debugger does when it runs a script. This, we see from the dictionary, corresponds to its `execute` command, which takes one parameter, a reference to the code window whose contents are to be executed. So we open the Attachments file and insert this handler:

```
on execute whatDoc
   display dialog "I am the attachment handler!"
   continue execute whatDoc
end execute
```

Compile and save. Now open a new code window, type a short script, and run it. The dialog appears, proving that we have extended the default functionality of running a script; but the script also does run, proving that we have maintained that default functionality through the `continue` statement.

You will see from experimentation that our attachment handler is called when the user presses the Run button, or chooses Run from the Script menu, or presses the Step button or does anything else which causes debugging to proceed, or gives the `execute` command or the `run script` command in AppleScript. (However, the Attachments file itself does not implicitly recurse into itself, since if it did, that recursion would be infinite; in other words, AppleScript commands given to Script Debugger from within the Attachments file are not themselves attachable.)

Now let's make our script more practical. Remember that we had a problem testing scripts intended to drive Script Debugger? We would like the window containing such a script to retire to the background while running, to simulate the situation under which it will be run eventually, without being represented by a window at all. Let's give any such window a property, Testing, which will be a boolean set to true. Then our `execute` attachment handler can go like this:

```
on execute what
   local isTesting
   copy false to isTesting
   try
      if script document 1's Testing then
         copy true to isTesting
      end if
   end try
   if isTesting then
      copy contents of script document 1 to whatToDo
      move script document 1 to end
      copy result to whatDoc
      do script whatToDo
      move whatDoc to beginning
   else
      continue execute what
   end if
end execute
```

To try this out, have two code windows. In the frontmost one, put this:

```
 contents of window 1
```

and run the script. You'll see that the result is "contents of window 1", proving that the window you are in is window 1, and that our `execute` attachment handler is not interfering with normal operations — it merely passes the `execute` command on up the line, with the `continue` statement. But now put this in the frontmost code window:

```
property Testing : true
```

```
copy "Hi there" to after text of window 1
```

You'll see a momentary flicker, and you'll find that "Hi there" has appeared in the *second* window. That's because our `execute` attachment handler, upon seeing the presence of the Testing property, sent the frontmost window to the back before running its contents, which is just what we wanted. We now have created a milieu in which we can easily test a script which is intended ultimately to be run as a script file from the Scripts palette, without its presence as a code window interfering with its own normal operation.

Now, please note that I'm not saying that the implementation presented here is the ultimately correct implementation of this attachment handler! In real life, you'd probably want the script to test for lots of other conditions before deciding just what to do. Writing an attachment handler is tricky business, and writing an attachment that interferes with fundamental operations like `compile` and `run` is the trickiest of all. Nevertheless, the example suffices to demonstrate the nature and power of attachability.

What commands can be attached? Any Apple event listed in the Script Debugger dictionary, except `get` and `set`; plus, any Apple event in the Standard Additions scripting addition; plus, the `Open URL` event. Furthermore, you can define several special handlers corresponding to events in the life of Script Debugger but not to any particular scriptable command. These handlers are:

- `SDStartup` — called when Script Debugger is launched
- `SDShutdown` — called just before Script Debugger shuts down
- `SDPasteTell` — called when you press the Paste button in the Applications palette or use drag-and-drop to create a `tell` block
- `SDPasteClipping` — called when you press the Paste button in the Clippings palette, but not when you use drag-and-drop to insert a clipping
- `SDDebuggerEnabled` — called when debugging is enabled
- `SDDebuggerDisabled` — called when debugging is disabled

If you look in the Attachments file, you'll see that both `SDPasteTell` and `SDPasteClipping` are implemented. For example, `SDPasteTell` is how Script Debugger works its magic where, if you drag from the Applications palette onto a selection in a code window, the resulting `tell` block wraps the selection.

A major complication in writing an attachment handler is the variability of parameters. First of all, you might not know in advance the type of a parameter, and this can make a difference. You can test this by asking for the class of the parameter. For example, in our `execute` attachment handler we might have reason to ask for `class of what` (the parameter) and do something special if this is not `cSCR` (a script document).

Second, an Apple event can have optional parameters, but AppleScript is really bad at dealing with these. Actually, AppleScript is really bad at dealing with these twice. It's bad when the event arrives, because it provides no way to know how many parameters were actually included. And it's bad when you try to `continue` the event, because you have to specify the parameters exactly; you can't just wave your hands and say, "Whatever parameters arrived here, pass those on up the line."

To help you deal with this, Script Debugger supplies the `retrieve parameter` command. In a `try` block, you specify as `retrieve parameter`'s parameter the four-letter code that designates the parameter you're interested in. If it's there, then you now have the parameter value; if it's not there, then you'll get an error, which you can catch because you're in a `try` block. For example:

```
on save theObj
   try -- was the Run Only parameter included?
      set pRunOnly to retrieve parameter «class ASro»
      set pRunOnlySpecified to true
```

```
   on error
      set pRunOnlySpecified to false
   end try
   -- and so forth...
```

But it is still up to you, when the moment comes to `continue` the command, to provide all the different possible variations that the command allows. This is absolutely dreadful and tedious, but at present there's just no way around it. For example, if a command allows two optional parameters, you'll need four versions of the `continue` statement: with neither optional parameter, with the first one, with the second one, and with both. You'll use a massive `if-then` structure to pick the right one to perform, based on what you found out earlier (using `retrieve parameter`) about what parameters were supplied.

For example, suppose that, in addition to the information about the `run only` parameter, we've captured the `in` parameter to our `save` handler (if there is one) in `pIn`, and that we've recorded whether there is such a parameter in `pInSpecified`. And let's pretend, for the sake of simplicity, that these are the only possible parameters for `save`. Then there are four ways to continue the `save` command, and we'd have to say:

```
if pRunOnlySpecifed then
   if pInSpecified then
      continue save theObj in pIn with run only
   else
      continue save theObj with run only
   end if
else
   if pInSpecified then
      continue save theObj in pIn
   else
      continue save theObj
   end if
end if
```

In reality, of course, there are not four ways to `continue` the `save` command; there are lots more than that! You'd have to test for, and `continue`, every possible combination of parameters.