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 AppleScripts. Script Debugger has advanced editing features including support for scripts larger than 32K and Drag and Drop editing. It is fully Projector aware, and allows you to make read-only projector files modifiable with the click of a button. It also provides you with a complete debugging environment. Script Debugger allows you to single-step through your scripts and examine the contents of variables while your script is executing.

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

About This Manual

This manual explains how to install and use the Script Debugger software. Chapter 1 gets you started with Script Debugger. It tells you what you need and walks you through the installation process.

Chapter 2 introduces you to Script Debugger's interface. You will learn about the different windows, the function of the various pop-up menus, and the basic Script Debugger features. Chapter 3 discusses each of Script Debugger's features in detail. It shows you how to get the most out of those features by walking you through creating and editing a sample script.

Chapter 4 describes Script Debugger's debugging capabilities. It begins by discussing some strategies for debugging scripts, and then steps you through debugging some example scripts. Chapter 5 shows you how to customize Script Debugger by configuring the preferences, adding scripts to the Extensions menu, and by attaching scripts to the other menu items. Chapter 6 describes some common problems that you might encounter while working with Script Debugger and offers some solutions.

You will also find several appendices at the end of the guide that contain documentation for the extension scripts that ship with Script Debugger. In addition, there is also discussion of Script Debugger's scripting interface and the documentation for Late Night Software Scripting Additions and Scheduler that are shipped with Script Debugger. You will also find an appendix explaining how to find out more about AppleScript and how to get help with your scripts.

This manual assumes that you are already familiar with the Macintosh desktop as well as with basic Macintosh skills, such as using the mouse and using the Chooser. This document also assumes you are somewhat familiar with AppleScript.

On-screen Help

Script Debugger includes on-screen information that you can consult when you need help. Balloon Help is a feature of System 7 that explains the function or significance of items you see on the Macintosh screen. To turn on Balloon Help, pull down the Help menu from the Help icon ( ) near the right end of the menu bar, and choose Show Balloons. 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.

What You Need to Get Started

To use Script Debugger, you must have a Macintosh II or later with Color QuickDraw. Your Macintosh must be running system software version 7.0 or later with at least two megabytes of free memory. If you install all of the components from the Script Debugger disk, you will need close to 3 megabytes of disk space. This includes AppleScript 1.1. If you already have AppleScript or System 7.5 installed, you will not need as much free space on your hard disk.

The Script Debugger package includes the following items:

- one 1.4mb disk, titled Script Debugger Install
- this manual, the *Script Debugger User's Guide*
- the Script Debugger Quick Reference card

Script Debugger has been accelerated for Power Macintosh, however, AppleScript 1.1 has not. You will see performance increases while editing scripts in Script Debugger, but executing and stepping through scripts will still be emulated on a Power Macintosh.

Installing Script Debugger

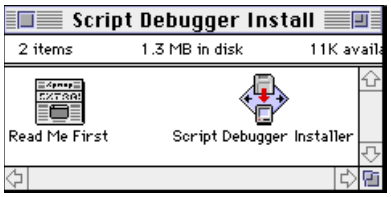
Follow the steps in this section to install the Script Debugger software on your computer.

1. Insert the Script Debugger Install diskette.

Before inserting the Script Debugger Install diskette, make sure it is write protected by moving the write protect tab on the back of the diskette to the upper position.

When you insert the Script Debugger Install diskette, the Script Debugger Installer disk window appears *(Figure 1-1)*.

Figure 1-1
The Script Debugger installer disk



2. Read the Read Me First file.

The Read Me First file contains late breaking information that may not appear in this manual. The Read Me First file also contains errata information and release information for minor releases of Script Debugger.

3. Run the Installer.

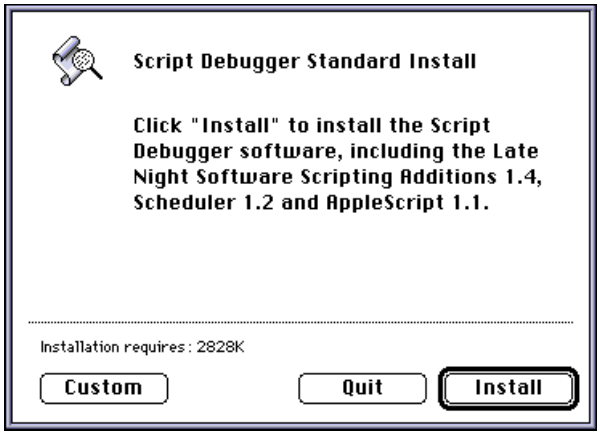
Install the Script Debugger software by running the Script Debugger Installer. To run the installer, double-click the Script Debugger Installer application. You will be greeted by an instruction dialog box *(Figure 1-2)*.

Figure 1-2
The Installer startup message

4. Click the Continue button to continue with the installation.
5. Select Easy or Custom installation.

When the installer is running, you are given the option of doing a standard or a custom installation *(Figure 1-3)*.

Figure 1-3
The Installer window



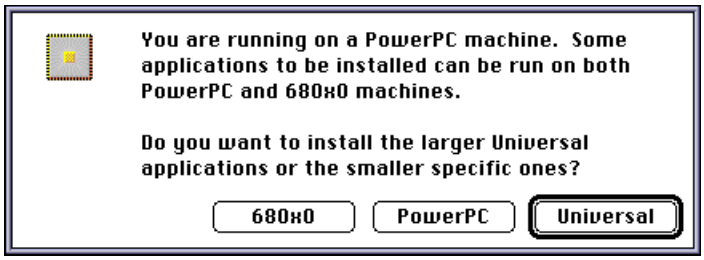
Clicking on the Install button installs AppleScript 1.1, Late Night Software Scripting Additions, and Scheduler, in your System Folder. It also installs a folder containing Script Debugger and the example scripts that come with it.

You may want to use the Custom installation option. If you are running System 7.5, AppleScript 1.1 is already installed. Clicking on the Custom button allows you to install any of the components in the Script Debugger package.

6. Select the version of Script Debugger that you want to install.

After you select the type of installation and confirm that you want to reboot your Macintosh after the installation is complete, the Installer then prompts you to select the 680x0, PowerPC, or Universal version of Script Debugger *(Figure 1-4)*.

Figure 1-4
The Installer prompts for the version of Script Debugger



Clicking the 680x0 button installs a version of Script Debugger that is only compatible with 680x0 Macintoshes.

Clicking the PowerPC button installs a version of Script Debugger that is only compatible with Power Macintoshes.

Clicking the Universal button installs a fat binary version of Script Debugger. This version of the program is larger than the other two, but is compatible with both 680x0 and PowerPC-based Macintoshes.

Registering Your Copy of Script Debugger

In order to provide you with service, we need to know who you are. Please take the time now to complete and mail the product registration card. Just fill in your name, address, and computer model. We also would appreciate your filling out the short questionnaire attached to the registration card. Thank you in advance for your prompt response.

CHAPTER 2

Getting Started

This chapter introduces you to Script Debugger's interface and features. It discusses all of Script Debugger's windows, shows you their different features, and explains how to use them. The chapter also describes the functions of the various pop-up menus in Script Debugger and gives you some tips for getting the most out of them.

The Script Debugger Interface

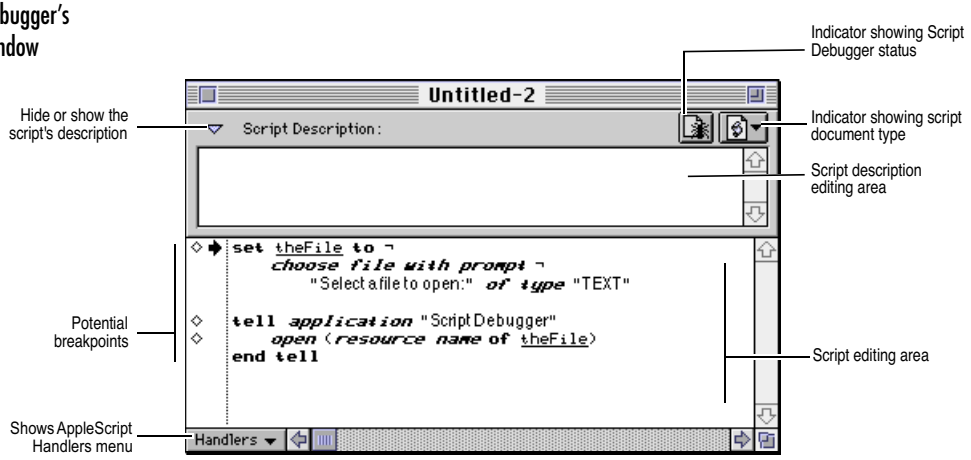
If you have used Apple's Script Editor, Script Debugger's Script window will seem very familiar *(Figure 2-1)*. It has a Script Description field that you can show or hide by clicking on the triangle next to "Script Description." You can change the font, style, size, and color of the text that you enter in the Script Description field using the hierarchical menus at the bottom of the Edit menu.

A Script Window

In addition to the Script Description label, you will also notice a Script Debugger status button and a document pop-up menu. The options in both of these items are also available in the Save As dialog, but the button and pop-up menu items can save you time.

Script Debugger can save the expressions in the Data window and breakpoint settings with your file. The Script Debugger status button is valid for all file types, except text files. It controls whether or not breakpoints, Data window expressions, and positions of the Data window, are saved. If the button has an "x" through it, Script Debugger does not save the debugging information.

Figure 2-1
Script Debugger's Script window

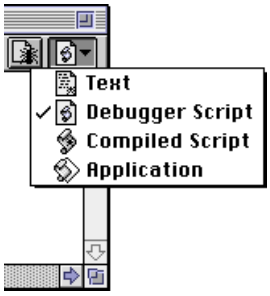


The black arrow is the current line indicator. When you are stepping through the script, it indicates which line Script Debugger will execute next.

NOTE: *If you save a script in Compiled or Application format with debugging information and then edit the file using the Script Editor, the breakpoint information will be lost when you open the file again using Script Debugger.*

The pop-up menu on the right side of the Script window lets you see at a glance whether the open script was saved in Text, Debugger, Compiled, or Application format *(Figure 2-2)*. The Script Type menu also allows you to change the document type without performing a Save As. You can also open the Save As dialog box by option-clicking on the script type pop-up menu.

Figure 2-2
The script type pop-up menu



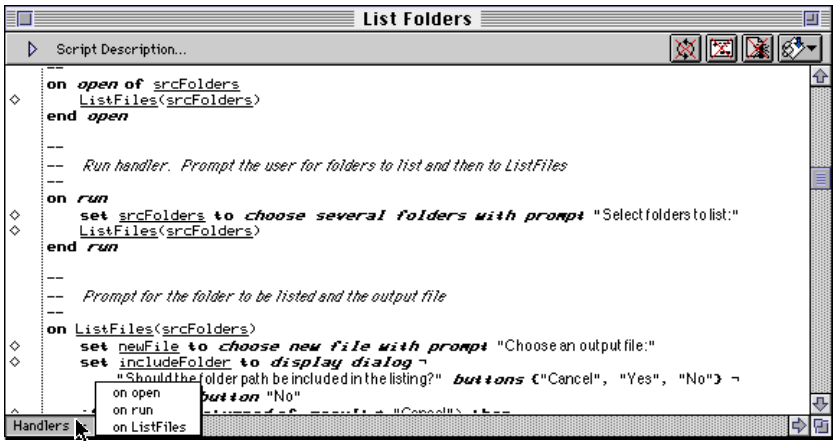
You edit your scripts in the area below the script description. If you have set the automatic indentation in the Preferences dialog box, your script is automatically indented as you type it in.

You can change the font, size, style, and color of your script using the AppleScript Formatting options in the Preferences dialog box. You will also notice that potential breakpoints appear in the left boundary of the Script window. Clicking on a diamond activates the breakpoint.

In the lower left corner of the Script window is a Handlers pop-up menu. If your script has any handlers in it, they are listed in this pop-up menu *(Figure 2-3)*. If you select one of the handlers from the pop-up menu, the Script window jumps to that handler. If you Option-click on the Handlers menu, it lists the global variables, properties, and script objects in your script along with the handlers.

If you are working on a script application or droplet script, you will see an additional set of script indicators above the Script Description field *(Figure 2-3)*.

Figure 2-3
Script indicators and the Handlers pop-up menu



In Figure 2-3, the icon with the two arrows forming a circle indicates whether or not the script is a Stay-Open script. In this case, the cross through the icon indicates that the script stops whenever the run handler finishes. You can change this setting by clicking the Stay Open indicator or selecting the Stay Open checkbox in the Save As dialog box.

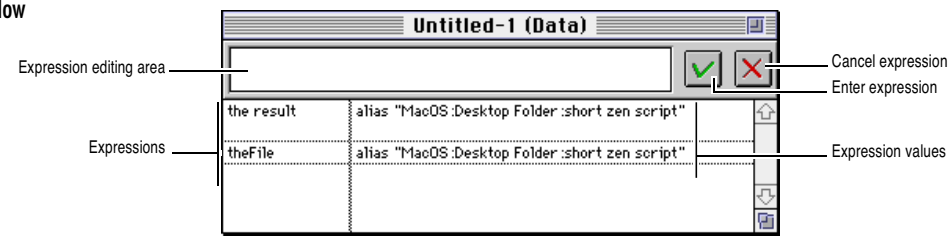
The next icon indicates whether or not the script's description is shown in a splash-screen whenever the script runs. In this case, a cross through the icon indicates that the description is not shown. You can change this setting by clicking on the Splash Screen indicator or selecting the Show Startup Screen checkbox in the Save As dialog box.

The Data Window

Every Script window has a Data window associated with it. The Data window contains the variables that you enter from the script. As you run or step through a script, the contents of the variables in this window are updated.

To add a variable to the Data window, enter it in the editing area, and then press the Return key or click the Enter Variable button to the right of the editing window. You will notice that the variable you entered is added to the list of variables in the bottom portion of the window. You can display global variables and properties, but not local variables.

Figure 2-4
A Script Data window

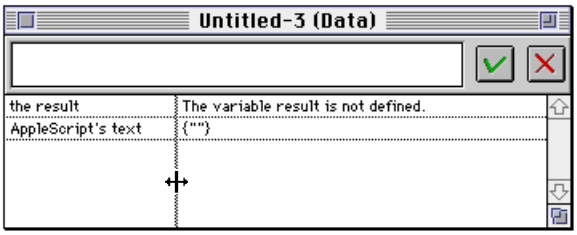


If you are defining a variable for the first time, Script Debugger places the comment "The variable <variable name> is not yet defined." in the values pane. If you have already run the script or have already stepped through the first occurrence of the variable, the current value of the variable is entered in the pane when you press the Return key *(see Figure 2-4)*.

You can control whether or not the information in the Data window is saved with your script. To save the information, use either the Save Debugging Information checkbox in the Save As dialog box or the Debugging Information icon situated in the upper right hand corner of the Script window.

You can resize the panes in the Data window by moving the pointer over the dividing lines until it becomes a double-headed arrow *(Figure 2-5)*. If you press and hold the mouse button, you can change the size of a pane by dragging it. Generally, a pane only displays a single line of text. While this is adequate for a variable name, you only see the first line of the contents of the variable. If the script returns a list of values for your variable, you may want to make the pane larger so you can see more of the list.

Figure 2-5
Resizing the panes in the Script Data window



The Controls Window

The Controls window floats above your script and Data windows *(Figure 2-6)*. It provides you with a way to compile and control the execution of your scripts as well as begin recording a script with a single click.

Figure 2-6
The Controls window

You can hide the Controls window by selecting the Hide Controls command from the Windows menu or by typing Command- – (Dash).

Even when the Controls window is hidden, you have access to all of its commands through the Controls menu and keyboard shortcuts.

The Controls window is automatically hidden when windows other than the Script and Data window, are activated. For instance, it is hidden when you bring the Result, Event Log, or a dictionary window to the front. It is hidden under these circumstances because the buttons in the Controls window are not active.

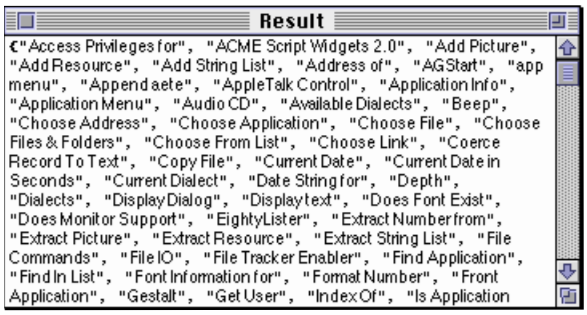
The Result and Event Log Windows

The Result window shows the results of your script. For example, the script

```
list folder "Macintosh HD:System Folder:Extensions:Scripting Additions"
```

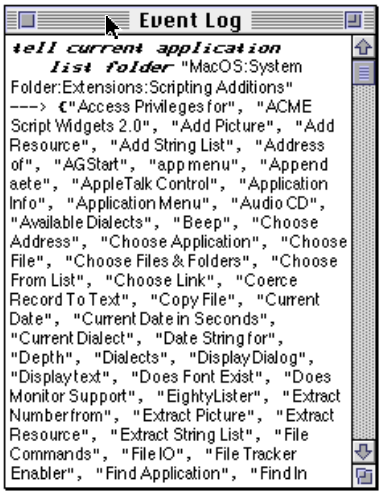
produces a list of the contents of the Scripting Additions folder. It would be returned as a list, and you would see the list in the Result window *(Figure 2-7)*.

Figure 2-7
The Result window



The Event Log window shows you the Apple events that Script Debugger sends to itself, to scripting additions, and to scriptable applications. If you executed the list folder script above, the Event Log window would show you the application receiving the Apple events, the script that is sent, and the results of the script *(Figure 2-8)*.

Figure 2-8
The Event Log window



Notice that the result of the script appears as a list following the arrow (—>).

A Dictionary Window

Every scriptable application and scripting addition has a dictionary which contains all of the AppleScript terms that it understands. Script Debugger's Open Dictionary command in the File menu allows you to open the dictionaries of scriptable applications and scripting additions.

The Additions menu item under the Open Dictionary command opens the dictionaries of your scripting additions. When you select the command, Script Debugger opens all of the additions installed in your Scripting Additions folder and builds an Additions Dictionary window *(Figure 2-9)*.

Figure 2-9
The Additions Dictionary window

Script Debugger groups events in the additions together by suite. For instance, all of the events in the Late Night Software Scripting Additions are grouped together in the Late Night Software suite.

You can use the Find command to search for Apple events and classes in the Dictionary window.

To open an application's dictionary, follow these steps:

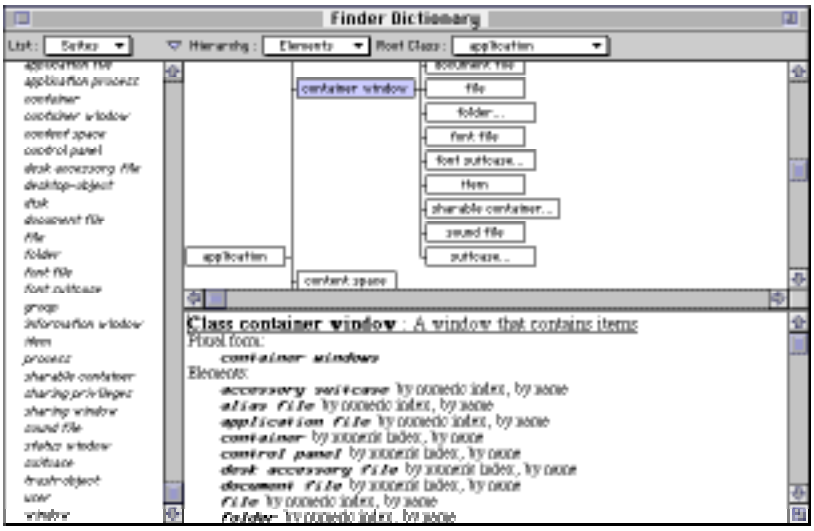
1. Choose Other from the Open Dictionary hierarchical menu under the File menu.

A dialog box opens where you can select an application. Notice that only scriptable applications appear in the dialog box.

2. Select an application and click the Open button.

The Dictionary window opens *(Figure 2-10)*.

Figure 2-10
The Dictionary window for the Scriptable Finder

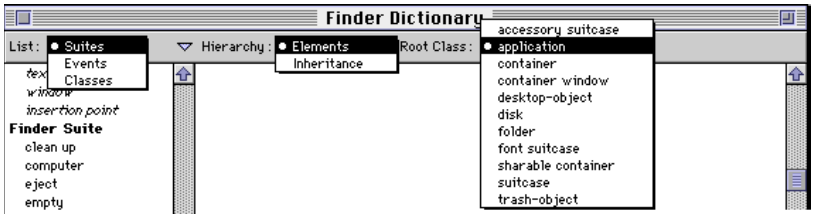


As in the Additions Dictionary, the scrolling list along the left side of the window shows you the Apple events that the application understands and the objects in the application. This information is grouped by suites in the application. The Finder, for instance, understands events from the Required and Standard suites as well as its own Finder suite.

The right side of the window contains the specific information about the event or events that you select on the left side of the window. When you select an event, its description, forms, and syntax appear on the right side of the window along with a description of the parameters. You will also notice that you can select and copy the syntax examples from the window. If you have Macintosh Drag and Drop installed, you can drag the examples from the Dictionary window and drop them in a Script window.

The List pop-up menu on the top left side of the window lets you change the display of the suites and events. Selecting Events from the pop-up hides the suite information and sorts the events alphabetically by event name. This is a quick way to locate an event when you cannot remember which suite it is in. It also lets you check for more than one event with the same name in your scripting additions since events will appear in the list for each scripting addition they are in. Selecting Classes from the List pop-up hides all of the suite and event information and just displays the classes available.

Figure 2-11
The Dictionary pop-up menus

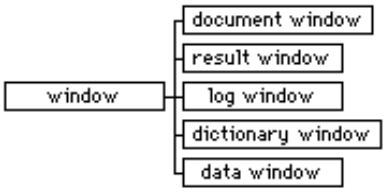


The Hierarchy and Root Class pop-up menus control the display of the objects in the hierarchical diagram at the top of the Dictionary window *(see Figure 2-11)*. You can display the diagram by clicking on the triangle next to the Hierarchy label. You can resize the diagram pane by clicking below the horizontal scroll bar and dragging. If you select an object in the diagram, its description is displayed in the pane below it.

NOTE: *Not all scriptable applications support object heirarchies. In these cases, Script Debugger is not able to show a diagram.*

When you select Elements from the Hierarchy menu, the diagram shows the objects in the application according to their hierarchy. The ellipses at the end of some class names in the diagram indicate that the sub-tree has been truncated. This is done if the sub-tree is already displayed in another area of the diagram. If you select inheritance from the Hierarchy menu, the diagram shows the object selected in the Root Class and the other objects in its inheritance hierarchy.

Figure 2-12
The window object and its sub-classes



Not all applications use inheritance. If an application does not use inheritance, the Root Class menu will be empty. If an application uses inheritance, the Root Class menu will contain all of the objects in the application which have sub-classes. Selecting an object from the menu changes the hierarchical diagram to show the object you have selected and all of the objects which inherit from it. For instance, in Script Debugger, the window object has a number of sub-classes like the Dictionary window *(Figure 2-12)*.

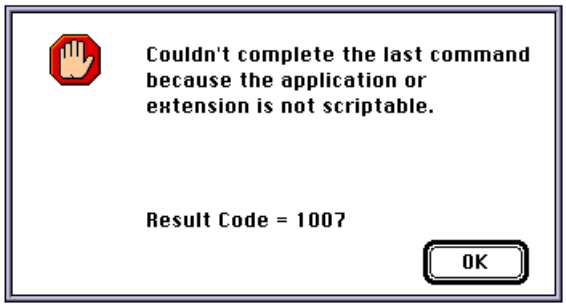
You can option-click on an item in the diagram to display the description of the item and make it the root class in the diagram.

Dictionary Shortcuts

If you open a program's dictionary regularly, you can add it to the Open Dictionary menu by placing an alias in the Dictionary Items folder. You can add the alias from Script Debugger by using the Add to Open Dictionary Menu extension script under the Extensions menu. For more information about this extension script, refer to Appendix A, *Extension Scripts*.

You may notice that Script Debugger appears in the Open Dictionary dialog box. However, you can only open the Script Debugger's dictionary if you have turned on the Enable Script Debugger Dictionary checkbox in the Preferences dialog box. If you have not checked the Enable Script Debugger Dictionary checkbox, and you try to open the dictionary, you will get an error message saying that the application is not scriptable *(Figure 2-13)*.

Figure 2-13
The not scriptable error message



You may get this error message if an application or scripting addition has a dictionary, but there are no commands in it.

You can also open an application's or scripting addition's dictionary by dropping it on Script Debugger's icon. If an application does not have a dictionary, you will see the error message in Figure 2-13.

CHAPTER 3

Creating and Editing Scripts

This chapter explains how you can use Script Debugger to create and edit scripts. It provides information about the script formats that Script Debugger can open and save, as well as instructions for editing and printing scripts. This chapter also discusses compiling and recording scripts.

Creating and Editing a Script

In this portion of the tutorial, we create a simple script to list the contents of a folder using the Choose File and the List Folder scripting additions that are provided with AppleScript. Afterwards, we modify the script to include a test for the Scriptable Finder.

To create a new script document,

1. Choose New Default Script from the File menu.

A new, untitled Script window opens. You will also notice that a Data window for the script opens below it.

2. Enter a script. For the purposes of this tutorial, enter the following script:

```
choose folder with prompt "Select a folder to list..."
copy the result to theFolder
list folder theFolder
```

This script uses the Choose File scripting addition to open a dialog box where you can select a folder for a listing of its contents. The "with prompt" lets you add the text "Select a folder to list..." to the dialog box. Once you select a folder in the dialog box, your selection is placed in AppleScript's result variable. The second line of the script places the contents of the result variable in the variable theFolder. The third line of the script uses the List Folder scripting addition to get a listing of the folder referred to by the theFolder variable.

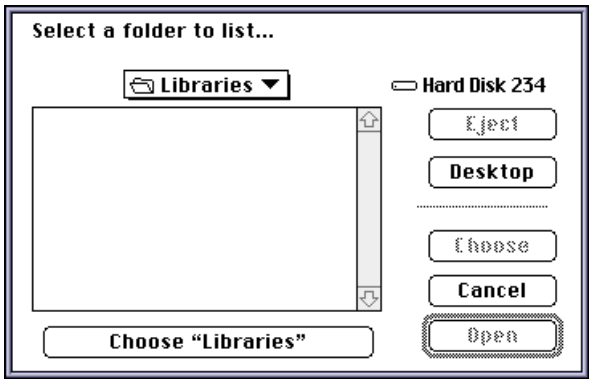
3. Select run from the Control menu or click on the Run button on the Control window. When the script begins to run, the cursor changes to two gears.

Script Debugger uses three cursors to indicate that it is busy with an action: the gears, the beachball, and the watch. The gears indicate that a script is executing. You should be able to switch to other applications, select from the Controls menu, or click on the buttons in the Controls window while the gears are turning. The spinning beachball indicates that Script Debugger is executing an attachment or extension script. You should be able to switch to other applications while the beachball is spinning. The watch indicates that Script Debugger is busy. It appears when Script Debugger is responding to an Apple event or performing a lengthy operation like compiling or opening a Dictionary window.

When you select Run, Script Debugger will first compile your script and then run it. You will know that the script has compiled because the format will change to reflect your formatting choices in the Preferences dialog box. If you have made any typing mistakes, you may find them at this point in the form of compile errors.

Once Script Debugger begins to run your script, a dialog box opens and you are prompted to select a folder *(Figure 3-1)*.

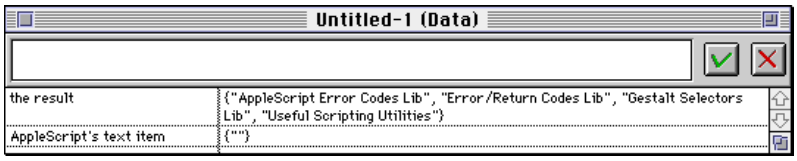
Figure 3-1
Choose Folder dialog box



4. Select a folder by navigating to it and then clicking on the Choose button at the bottom of the dialog box.

A moment later the script finishes running, and you will notice a listing of the folder contents in the result pane of your script's Data window *(Figure 3-2)*.

Figure 3-2
The Data window with the listing results



Cursor Movement

While you have the script open, you may want to use the command key equivalents to move the cursor around in your script. You can use the arrow keys to move the cursor a character or a line at a time. You can also use the Option and Command modifier keys in conjunction with the arrow keys to move a word and a line at a time.

1. Press Option-Up arrow to move the cursor to the beginning of the script.
2. Press Option-Right arrow a few times. You will notice that it moves the cursor to the beginning of the next word.
3. Press Option-Left arrow a few times. You will notice that it moves the cursor to the end of the preceding word.
4. Press Command-Right arrow to move the cursor to the end of the current line.
5. Press Command-Left arrow to move the cursor to the beginning of the current line.
6. Press Option-Down arrow to move the cursor to the end of the script.

You can also select words and lines in your script by holding down the Shift key in combination with any of the cursor movement keys. For instance, if your cursor is still at the end of the script, you can select the entire script.

7. Press Shift-Option-Up arrow to select to the beginning of the script.

Editing Scripts

In addition to the cursor movement keyboard shortcuts, Script Debugger supports the standard Cut, Copy, and Paste commands with a couple of noteworthy differences. If you still have the Script window open for the script you were working on above, you can test these commands.

1. Select several lines in the script and then select the Cut command from Edit menu.
2. Select the Undo command from the Edit menu to replace the lines that you just cut from the script.
3. Select the Clear command from the Edit menu to remove all of the lines that you have selected.
4. Select the Undo command from the Edit menu again to replace the lines you just cleared.
5. Now open a new Script window by selecting Default Script from the New menu.
6. When the Script window opens, select the Paste Reference command from the Edit menu.

This pastes a reference to the text that you copied from in the Script window. You could close your original window and the pasted reference would still refer to it.

7. Bring your tutorial Script window to the front and add parentheses to the last line of your script so that it looks like this. Yes, it is nonsensical.

```
list (folder theFolder())
```

8. Place your cursor between the two parentheses after theFolder and select the Balance command from the Edit menu.

The command will select the two parentheses that the cursor is between. Select the Balance command a second time and Script Debugger highlights from the last parenthesis to the first one in the line.

Copying and Pasting Text

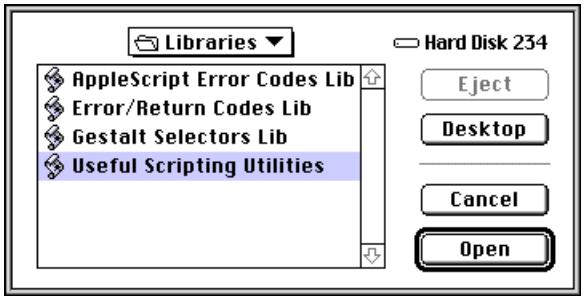
While your script uses scripting additions to get a listing of a folder, you could take advantage of the Scriptable Finder. Unfortunately, it might not be available in every situation, so you could add a test to your script to check for the Scriptable Finder. The Libraries folder has a Useful Scripting Utilities script that contains a handler to check for the presence of the Scriptable Finder.

In this portion of the tutorial, we will copy the handler into the script that we are working on and make some modifications.

1. **Select Open Script from the File menu.**

   When the Open dialog box appears, navigate to the Libraries folder and then open the Useful Scripting Utilities script *(Figure 3-3)*. The Libraries folder is inside of the Examples folder in your Script Debugger folder.

Figure 3-3
Open Script dialog box



2. **Click on the Handlers pop-up menu at the bottom of the Script window. Select the CheckForScriptableFinder handler at the bottom of the menu.**

Script Debugger scrolls the window to the bottom of the script and places the cursor at the beginning of the handler.

3. Select the two properties and the handler.

This is the portion of the script that you will be copying:

```
property gestaltFinderAttr : "fndr"
property gestaltOLSCompliantFinder : 3

on CheckForScriptableFinder()
   —
   — This handler allows you to check for the existence of the Scriptable Finder
   — before you begin using it.
   —

   try
      return (get gestalt gestaltFinderAttr bit gestaltOLSCompliantFinder)
   on error -1708
      error "Gestalt Scripting Addition is not installed." & return & return & ¬
         "Please refer to the Script Debugger User's Guide for information on " & ¬
         "how to install Script Tools"
   end try
end CheckForScriptableFinder
```

4. Select Copy from the Edit menu.
5. Bring your Script window to the front and place the cursor at the beginning of the script.
6. Select Paste from the Edit menu to add the properties and handler at the top of the script.

NOTE: *If you have Macintosh Drag and Drop installed, you could accomplish this in fewer steps. You would select the properties and handler in the Useful Scripting Utilities script. Then you would click on it a second time and drag it to your Script window.*

Now we need to modify the script to take advantage of the handler.

7. Add a call for the handler above the top line of the original script.

```
CheckForScriptableFinder()
```

This will return whether or not the Finder is scriptable. If it is, the handler will return true. If not, the handler will return false.

8. Add an "if then" statement to the original script.

```
if the result = false then
   choose folder with prompt "Select a folder to list..."
   copy the result to theFolder
   list folder theFolder
else
```

If the Scriptable Finder is not present and the handler returns false, this part of the script uses the scripting addition to get a listing of the folder.

9. Add another "if statement" and a "tell statement" for the Finder.

```
   if the result = true then
      choose folder with prompt "Select a folder to list..."
      copy the result to finderFolder
      tell application "Finder"
         list folder finderFolder
      end tell
   end if
end if
```

This "if then" statement is used if the Scriptable Finder is present and the handler returns true. This portion of the script uses the same Choose File scripting addition to select the folder whose contents will be listed, but the tell statement lets the Finder generate the listing instead of the List Folder addition.

Compiling Scripts

Before it can run a script, Script Debugger must compile it. This involves checking the syntax of the script for any AppleScript errors or any errors in the portions of the scripts that use applications or scripting additions. A script must be re-compiled after you make any changes to it.

Aside from checking the syntax, the main reason for compiling a script is to make it run faster. Once a script is compiled, it does not have to be checked before it is run.

1. You can compile the script that we have been working on. If you have already closed it, enter or open a script that you want to compile.
2. Select Compile from the Controls menu or click on the Compile button on the Controls window. Script Debugger displays a watch icon to let you know that it is working.

If Script Debugger encounters any errors while it is compiling the script, it stops the compile and displays a dialog box with an error message in it.

3. When Script Debugger finishes compiling, the formatting of the script changes to let you know that it is compiled.

For instance, if you have specified that Language Keywords should be bold (or red, if you are using a color display), the language keywords in your script will become bold.

4. Save your script when Script Debugger has finished compiling it.

Once you have checked the syntax of your script, you might want to check the variables to see that your script is running properly. In the case of our tutorial script, you will be able to see which if statement the script uses to get the listing of the folder. In order to check the variables, you will need to add them to the Data window.

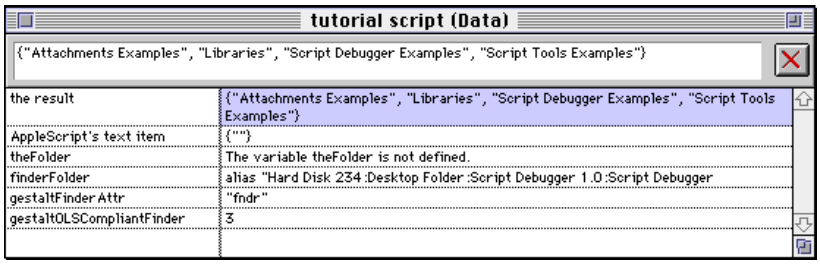
1. Select Bring Data to Front from the Windows menu or press command-; to bring the Data window to the front.
2. Place the cursor in the expression editing area and enter the variable used in the first if statement: theFolder.
3. Press the Return key to add the variable to the expression panes.

You will notice that the value portion of the Data window says that "The variable theFolder is not defined" and that the variable is selected. Script Debugger shows this "error" and selects the variable so you can correct it.

4. Press the Return key to add the variable to the expression panes. Click below the variable to deselect it.
5. Enter the variable used in the second if statement in the expression editing area: finderFolder.
6. Click the Run button in the Control window or select Run from the Controls menu.

As before, you will be prompted to choose a folder as the script runs. When the script finishes, you will be able to see that the variables in the Script window have been updated *(Figure 3-4)*. If you have the Scriptable Finder installed, the finderFolder variable will contain the path to the folder that you choose. Otherwise, theFolder variable will contain the path. You might also notice that the result variable will contain the listing of the folder. If you bring the Event Log window to the front, you will be able to see each step as your script runs. We will talk more about the Event Log and Data windows in Chapter 4, *Debugging and Stepping Through Scripts*.

Figure 3-4
The Data window after running the script



Recording Scripts

Script Debugger supports recording scripts. While not all applications that are scriptable are recordable, a growing number of them are. In general, recording a script is faster than writing it from scratch. You will probably need to edit the recorded script, but you can let the application do most of the work. You can also learn more about how to write scripts for an application by recording a common action in it.

For this tutorial, you may want to move the Script window so you can watch Script Debugger create the script. We will be recording a script using Scheduler. Scheduler is a combination of an Extension and a

Control Panel that allows you to launch programs and open documents at specified times.

If you chose the Easy Install option when you installed Script Debugger, the appropriate version of Scheduler was automatically installed in your Control Panels folder. If you chose Custom Install and did not install Scheduler, you will need to install it for this section of the tutorial.

1. Open the Scheduler Setup from your Control Panels folder and then switch to Script Debugger.
2. Open a new Script window and select Record from the Controls menu or click on the Record icon in the Controls window. A flashing tape icon appears on the Apple menu.
3. Switch to Scheduler Setup.
4. Click on the Add button. This opens a standard Open dialog box where you can select a document or application.
5. When the Open dialog box appears, select SimpleText or TeachText, or any other application.
6. Select When a Folder Changes from the Launch menu.
7. Click on the Set Folder button and select the Control Panel folder.
8. Now switch back to Script Debugger and select Stop from the Controls menu or click the Stop button on the Controls window. Script Debugger may complete the script after you have stopped recording.

Your script may look something like this:

```
tell application "Scheduler Setup"
   activate
   make new periodic occurrence with properties {scheduled file:alias "Hard Disk
234:apps:SimpleText"}
   set selection to occurrence 1
   set class of occurrence 1 to folder change occurrence
   set folder of occurrence 1 to alias "Hard Disk 234:System Folder:Control
Panels:"
end tell
```

While this script is just an example, it does show a useful function of Scheduler that you could use for running scripts.

Searching For Text

Script Debugger lets you search for text in your scripts. If you still have the script open that we were working with earlier, you can search for text in it.

To begin a search for text in your script,

1. Select Find from the Search menu or type Command-F.

The Find dialog box appears *(Figure 3-5)*.

2. Enter the text you are searching for in the Search For area of the dialog box.

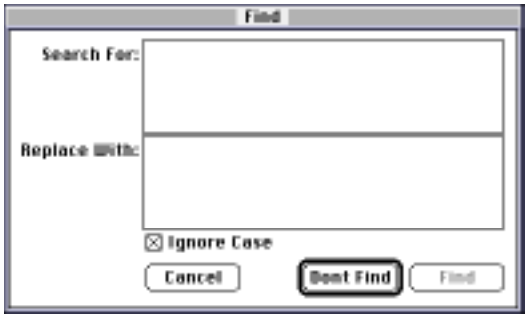
If you want Script Debugger to ignore the case of the text that it is searching for, enable the "Ignore Case" checkbox. Once you do this, Script Debugger will treat "TheValue", "theValue", and "thevalue" as the same search term.

3. Click the Find button or press Return to begin the search.

Script Debugger remembers the text in the Search For and Replace With fields between searches. If you have entered text in either field, but you click the Don't Find button, Script Debugger cancels the search but remembers the text. If you enter text in either field, but you click the Cancel button, Script Debugger forgets the text you entered.

You can find the next occurrence of the text by selecting Find Next from the Search menu or you can press Command-G. You can find a previous occurrence of the text by pressing Shift-Command-G.

Figure 3-5
The Find dialog box



Script Debugger provides you with a shortcut for this process in the Enter Selection command under the Search menu. You can select a text string in your script and then select the Enter Selection command. The text is inserted in the Search For field, and you can select Find Next from the Search menu to find the next occurrence of the text.

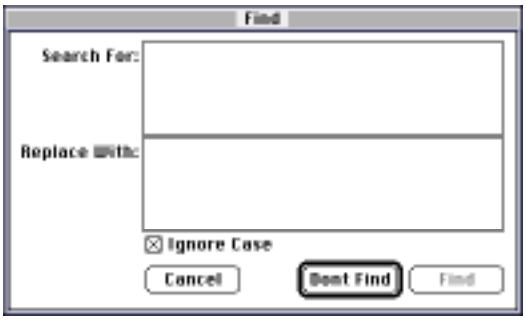
Replacing Text

Script Debugger lets you search for and replace text in your scripts. To begin a search for text in your script,

1. Select Find from the Search menu or type Command-F.

The Find dialog box appears *(Figure 3-6)*.

Figure 3-6
The Find dialog box



2. Enter the text you are searching for in the Search For area of the dialog box.

If you want Script Debugger to ignore the case of the text that it is searching for, enable the Ignore Case checkbox.

3. Enter the replacement text in the Replace With area of the dialog box.
4. Click the Find button or press Return to begin the search. Click the Cancel button to cancel the search.
5. When the Script Debugger finds the text, select Replace from the Search menu or press Command-H to replace the text that you found.

Script Debugger supports Undo and Redo for single replace operations. However, Undo and Redo are not supported for Replace All operations.

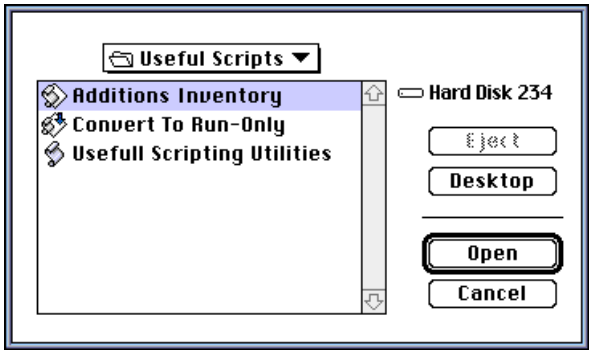
You can replace the text and find the next occurrence by selecting Replace and Find Again from the Search menu or by pressing Command-T. You can replace the text and find a previous occurrence by pressing Shift-Command-T. Replace All will replace all of the occurrences of the search string after the current selection; any occurrences of the string before the current selection are not replaced.

Opening Scripts

In the previous tutorials, we have used the Open command, but we did not look at all of the file formats that Script Debugger can open.

To open a script saved on disk, select the Open Script command from the File menu. When you choose this command, you are prompted to locate the file you want to open *(Figure 3-7)*.

Figure 3-7
Script Debugger's Open dialog box



If you are running System 7.5 or have System Utilities 3.0 installed, you will be able to see the different document icons in the Open dialog.

Alternatively, you can use the Finder to choose a file and open it. If the file was saved using Script Debugger, you can simply open it using the Finder by double-clicking the file or by selecting it and choosing Open from the File menu. You can also drag script files onto the Script Debugger icon.

NOTE: *If you double-click on a Script Editor file in the Finder, it will open the Script Editor. To open Script Editor files, you must use the Open command in Script Debugger or drag the file icon onto the Script Debugger icon.*

Script Debugger can open the following types of files:

Text Scripts

Script Debugger can open any text file. Any script that has not been compiled is saved in text format. If you are working on a script that has errors in it, you can save it in text format if you cannot save it in compiled format.

Debugger Scripts

Debugger scripts are created only by Script Debugger. Scripts saved in this format load more quickly than compiled scripts and don't need to be re-compiled each time they are opened. You can open a debugger script from the Finder and also with the Open command.

Compiled Scripts

Compiled scripts are produced by Apple's Script Editor and other script editing utilities. You can open a Script Debugger compiled script from the Finder, with the Open command, or by dropping them on the Script Debugger icon.

Script Applications

When a script has been saved as an application, you can double-click the script to run it. You can open script applications in Script Debugger with the Open command or by dropping them on the Script Debugger icon.

Droplets

When a script has been saved as a droplet, you can drop files on it to process them. You can open droplet files in Script Debugger with the Open command or by dropping them on the Script Debugger icon.

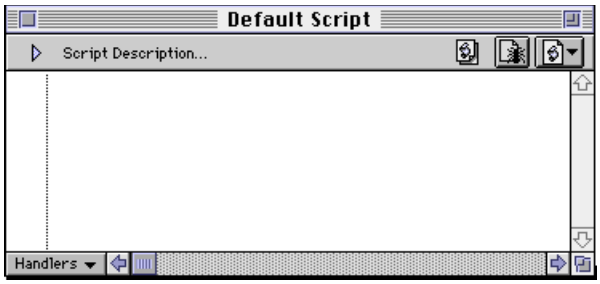
Stationary Pads

Script Debugger supports Stationary Pad files. These are files which, when opened, create a new untitled document containing pre-defined settings and contents. Stationary files also store the default location of Script Debugger's windows. To use a Stationary Pad file, open it as you would a normal script file.

Whenever you choose New Default Script from the File menu, Script Debugger automatically opens the Stationary Pad named "Default Script" located in the Script Debugger folder.

You can hold down the Option key while selecting Default Script from the New menu to open the Default Script directly and retain the stationary pad settings. When you do this, you will see a stationary pad icon in the Script window *(Figure 3-8)*.

Figure 3-8
The Default Script window with the stationary pad icon



If you want to create a stationary file with your own default settings for all new scripts, open a new file. Position the windows and enter the contents and pre-defined settings that you want to use every time you open a script. Select Save As... from the File menu. When the Save As... dialog box appears, enter the name "Default Script" in the name field and click on the Stationary Pad checkbox at the bottom of the dialog box. If you want to create a separate template with your own settings, you can name it anything you like. You can make your templates/ stationary pads handier by saving them in the Templates folder. Then they will always be accessible from the New hierarchical menu under the File menu.

TIP: *If you want a stationary pad that does not save window positions, you can create one using the following steps.*

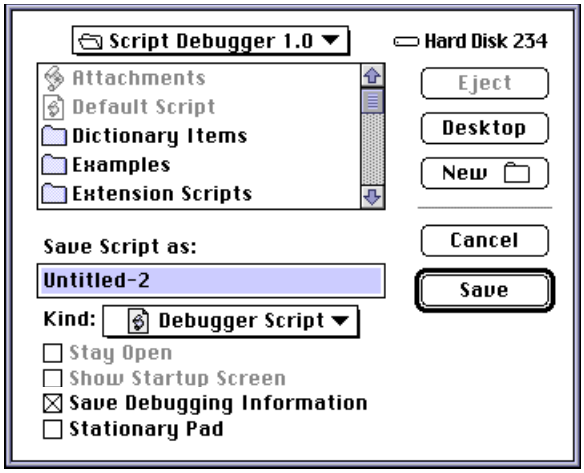
1. Move the Default Script stationary pad out of the Script Debugger folder.
2. Create a new script. Do not move or resize the Script or Data window.
3. Make whatever other changes you like to the script.
4. Save the stationary pad.
5. Move the Default Script back into the Script Debugger folder.

Whenever you open this stationary pad, Script Debugger automatically places the window for you.

Saving Scripts

To save a script that you have created, use the Save command in the File menu. If you have not saved your script before, you will be asked to name your script, choose the type of script document, and a location on your hard disk where the script is to be saved *(Figure 3-9)*.

Figure 3-9
Save As
dialog box



Replace the default "Untitled" name with the name you would like for your script. Use the Kind pop-up menu to choose the type of script document. The Kind pop-up menu has four choices *(Figure 3-10)*.

Figure 3-10
Save As
pop-up menu

Text

The Text format stores your script in a format that other script editors and word processors can read. Text scripts must be compiled before they can be run, but they do not need to be compiled before they are saved.

TIP: *If your script has compilation errors and you are in a hurry to save your work, you can save it as a text script or a debugger script.*

Debugger Scripts

The Debugger Script format is the default kind. This type of file records scripts in a format which is unique to Script Debugger. Using this format, Script Debugger can store your script, the script description, your breakpoints, and all the expressions, in the Data window.

While this format cannot be used by other applications, such as Apple's Script Editor, it does have distinct advantages. Script Debugger scripts open and close faster. Unlike compiled scripts and script applications, Debugger Scripts are not recompiled when they are saved.

Compiled Scripts

The Compiled script format stores your script and script description. If you have chosen the Save Debugging Information option, it also saves information about breakpoints you may have set or expressions which have been entered in the Data window.

Compiled scripts (and script applications) are re-compiled when saved. If the script has property definitions which prompt for things, or if Apple Script needs to locate an application, you will have to respond to these prompts before Script Debugger can complete the save operation. Because of this, property definitions are reset when saving in this format.

This file format is compatible with other applications, such as Apple's Script Editor, which can read compiled AppleScript script files.

Application

The Script Application format stores your script in the form of an application. Scripts stored in this format can be executed as stand-alone applications without the aid of Script Debugger or any other script editing utility. If your script has an "on open" handler, it is saved as a droplet.

Files saved as script applications are compatible with other script editing utilities, such as Apple's Script Editor, which can open script applications.

When saving a script as a script application, you can choose from the following options:

Stay Open

A Stay Open script continues to execute after the run handler finishes. This is useful if your script contains an idle handler or accepts Apple events from other scripts or applications.

Show Startup Screen

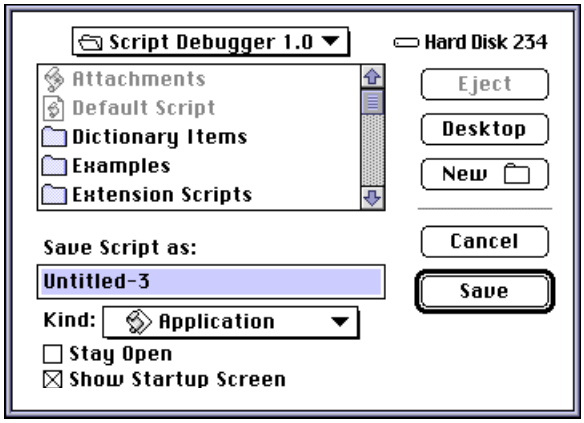
The Show Startup Screen option causes your script's description to be displayed before it executes.

Saving As Run-Only

You can save a script in Run-Only format. The only difference between Run-Only format and the standard compiled and script application formats is that the scripts cannot be examined.

To save a script in Run-Only format, select Run-Only from the File menu. A Save As dialog box appears *(Figure 3-11)*.

Figure 3-11
Run-Only Save As dialog box



Enter a name for the script in the "Save Script as" text box and select the kind of script from the pop-up menu. You can save the script as a compiled script or a script application as you can with the standard Save As dialog box.

NOTE: *When you save a script as Run-Only, you will never again be able to edit the script . If you are saving a script in a Run-Only version to distribute to users or clients, you should keep a backup copy of the script in text or compiled form.*

Printing Scripts

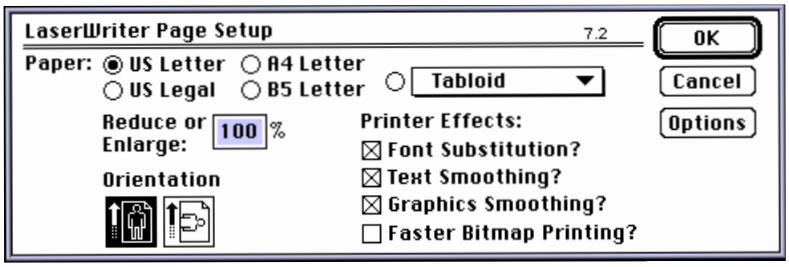
You can print your scripts using Script Debugger's Print command. When you print a script, a header is added to the top of each page which contains the name of the script, the day, date, and time the script was printed, and the page number.

To print a script,

1. Select Print Options from the File menu.

A Print Options dialog box appears *(Figure 3-12)*.

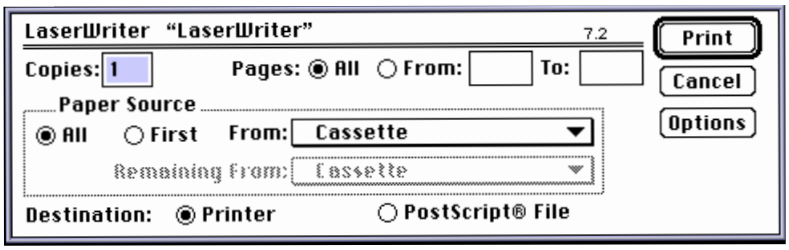
Figure 3-12
The Page Setup dialog box



2. Set the options in the Print Options dialog box and click OK.
3. Select Print from the File menu.

A Print dialog box opens *(Figure 3-13)*.

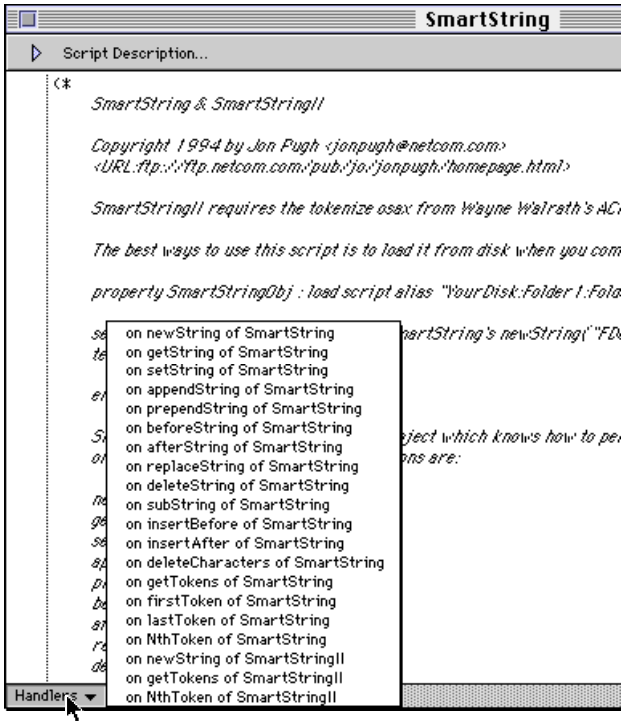
Figure 3-13
The Print dialog box



Locating Handlers, Script Objects, Global Variables and Properties

The Handlers menu at the bottom of the script window allows you to jump quickly to handlers in your script *(Figure 3-14)*.

Figure 3-14
The Handlers pop-up menu

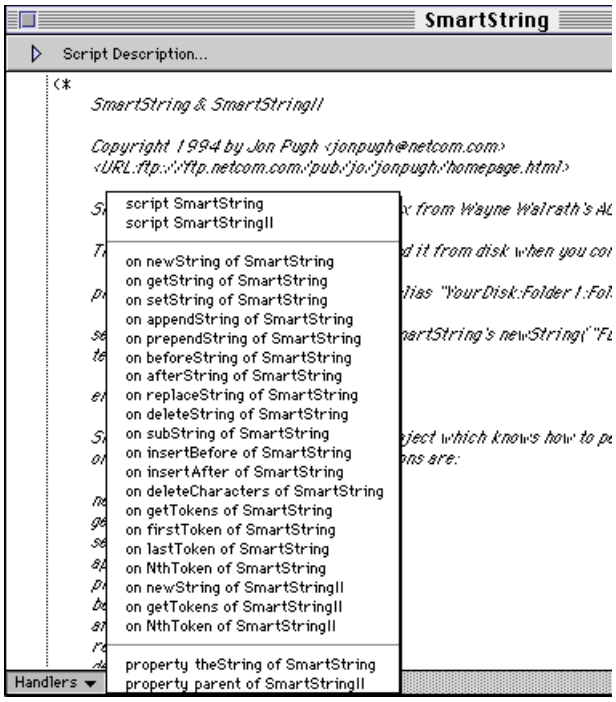


In addition to locating handlers, you can use the Handlers pop-up menu to locate script objects, global variables and properties in your scripts.

1. Hold down the Option key and click on the Handler pop-up menu at the bottom of the Script window.

The pop-up menu now shows script objects, global variables, and script properties, in addition to script handers *(Figure 3-15)*.

Figure 3-15
Script properties and objects



2. Select a script object, property, or global variable and release the mouse button. The Script window scrolls to the object you have selected and places the cursor before it.
3. Press Command-Up arrow if you have a script open with a handler in it. The cursor jumps to the beginning of the previous handler.
4. Press Command-Down arrow if you have a script open with a handler in it. The cursor jumps to the beginning of the next handler.

Extensions Scripts

Because Script Debugger is scriptable and attachable, you can add features or functionality to it using AppleScript. Twelve scripts which extend Script Debugger are supplied with it. You can see these scripts by selecting the Extensions menu *(Figure 3-16)*. In particular, several of the extension scripts make editing scripts easier: the Paste File Path, Paste Folder Path, Hide All Descriptions, and Show All Descriptions scripts.

Figure 3-16
The 12 Extension scripts supplied with Script Debugger

You can add your own scripts to the Extensions menu by placing your compiled scripts in the Extension Scripts folder. If you place files of any other type in the Extensions folder, Script Debugger ignores them.
To find out more about these Extension scripts, refer to Appendix A, *Extension Scripts*. If you want to write your own Extension Scripts, refer to Appendix B, *Script Debugger Scripting Interface*. It provides more information about the Script Debugger objects and the events that you can use to control them.

CHAPTER 4

Debugging and Stepping through Scripts

Script Debugger provides you with all the tools you need to debug your AppleScripts. This chapter begins by giving you some general strategies for debugging your scripts. It then walks you through the process of debugging scripts using Script Debugger's Data and Event Log windows. In addition to scripts, the chapter also demonstrates debugging AppleScript handlers and Script Debugger extensions.

Throughout this tutorial, we will work with the Debugging Tutorial script that you can find in the Tutorial folder inside the Examples folder of the Script Debugger folder. For purposes of this tutorial, we will step through a script in Script Debugger that copies portions of a script to an empty Script window.

To set your Script Debugger to follow through this tutorial, you need to do the following:

1. Launch the Script Debugger, open the Preferences dialog box and select the Enable Script Debugger Dictionary checkbox. You will need to quit Script Debugger after making the change.

If you have already enabled Script Debugger's dictionary, you can skip this step.

2. Launch Script Debugger again.
3. Open the Debugging Tutorial script.
4. Leaving the Untitled-1 Script window open, open the Execute Quit Handler script that you will find in the Extension Scripts folder inside the Script Debugger folder.

Depending on the size of your display, you may want to adjust the windows so you can watch the two Script windows in Script Debugger as you step through the script.

Debugging Strategies

Finding the errors or bugs in your AppleScripts can be time-consuming and tedious. Since you cannot always tell the status of your script while it is running, you must either embed debugging code or step through a script.

Script Debugger provides you with a sophisticated development environment for AppleScripts. You can set breakpoints and step through scripts a line at a time. While the previous chapters have introduced you to some of Script Debugger's features, this chapter will show you how to take advantage of those features as you develop scripts.

Common Errors

Some programming errors do not require sophisticated tools to locate. Probably the most common programming error that you can make in any language is a simple typing error. Typing errors can be particularly hard to debug since they can lead you away from the real error. For instance, if the word "line" in the following line of the Debugging Tutorial script

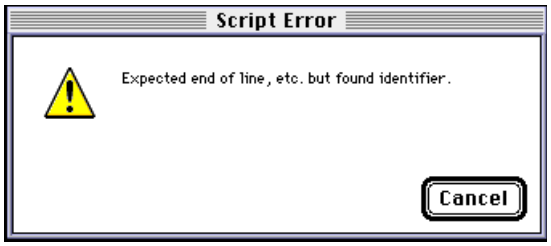
```
get style of word j of line i of theDoc
```

is misspelled as "lin"

```
get style of word j of lin i of theDoc
```

Script Debugger will highlight the word "lin" and signal a script error that it is expecting the end of a line *(Figure 4-1)*.

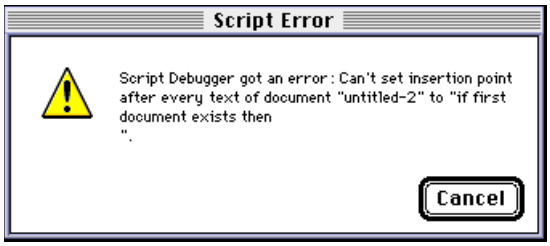
Figure 4-1
An error caused by a typo



Carefully checking the line that a script error is signaled on is the best way to find a typo in your script. Whenever you encounter a script error, you should not immediately suspect your AppleScript code. Scan the line the error is in to make certain that you have not made the simplest of programming mistakes.

Other easy mistakes to make are inserting the incorrect path to a file or folder in your script and referring to windows or objects that are not open or do not exist. In the Debugging Tutorial script, for instance, you would get a script error if the window named "Untitled-1" was not open. The script would run up to the point of copying the text to that non-existent window, but would fail when it tried to write to that window *(Figure 4-2)*.

Figure 4-2
Referencing a paragraph that does not exist



In this case, the reference to the window "untitled-2" should tip you off that something is amiss.

Simple Strategies

When you do have an error in your script, you can apply some simple strategies with Script Debugger to locate the problem. Begin by asking yourself some of the same questions that Scott Knaster does in his book, *How to Write Macintosh Software*.

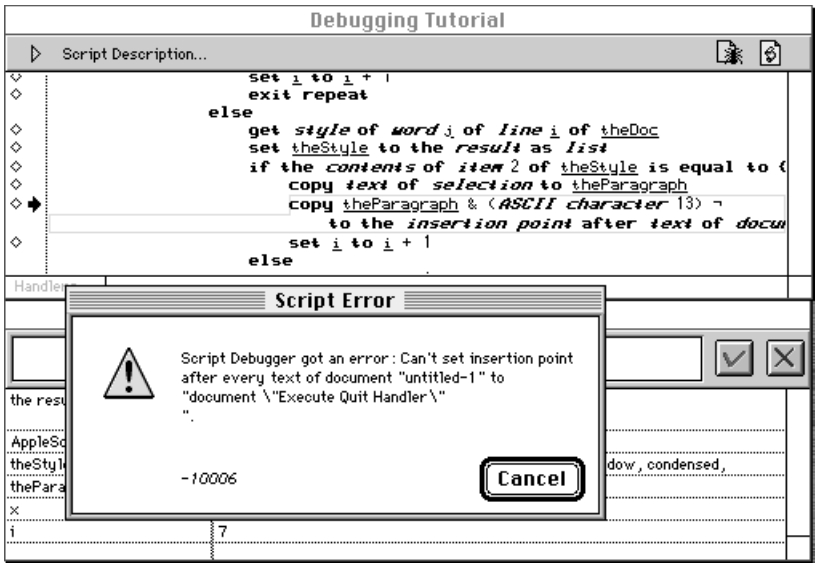
- Where did the error occur?
- What caused the error?
- What was the last line of the script to execute before the error occurred?

When you are trying to debug your script, try to isolate where the error occurred. For example, did the error occur in a tell statement directed to the Scriptable Text Editor or when the Finder was trying to write to a file? Answering this question helps you focus on the handler or control statement in the script that contains the error. You can use Script Debugger's breakpoint feature to speed the debugging process. If you need to run through the script to test the problem, you can click the Run button and the script will run up to the breakpoint and stop. You can then single-step through the problem portion of your script.

If you are debugging an open handler, setting a breakpoint allows you to stop the execution of the handler and step through the script. This is discussed in more detail later in this chapter.

Once you have determined where the error occurred, you can begin looking for the cause of the error. Did the Scriptable Text Editor look for paragraphs in a window that did not have any? Was the Finder trying to write to a file that another application already had open? In the example we used above, you could determine that the Script Debugger was looking in the wrong window by checking the name of the window that your script references *(Figure 4-3)*.

Figure 4-3
A highlighted line in the script



You can use the Data window with single-stepping or with the breakpoint feature. For instance, you can select the variables that you want to watch and then single-step through your script, watching how the variables change. You could also set a breakpoint that stops script execution just after a variable is updated. This allows you to execute your script quickly, but still examine the variables to make certain that they are being updated correctly.

You should also ask yourself which was the last line to execute. Script Debugger can be very helpful since it shows you exactly which line was executing when the error occurred and, in most cases, will flag a variable or word within the line *(Figure 4-3)*. Examine that line to make certain that your references are correct and the terms are not misspelled. You should be able to solve most of your problems by re-reading the line carefully.

Controlling Script Execution

While Apple's Script Editor allows you to execute the scripts that you create, it only allows you to stop the execution of a script. You cannot step through your script a line at a time or set breakpoints in your script. With Script Debugger, you can control the execution of a script and watch the variables and expressions change as you step through the script.

We ran a script in Chapter 3, so in this chapter we will concentrate on the debugging features of Script Debugger.

Preparing to Step Through a Script

For the next few sections, we will be using one of the example scripts that came with Script Debugger.

1. Open the Debugging Tutorial script in the Tutorial Examples folder if you have not already done so. This is the script:

```
tell application "Script Debugger"
   set theDoc to document "Execute Quit Handler"
   set x to count of lines of theDoc
   set i to 1
   repeat while i < x

      select line i of theDoc
      if length of line i of theDoc is greater than 0 then
         set j to 1
         get word j of line i of theDoc
         set theWord to the result
         repeat
            if theWord begins with "—" then
               set i to i + 1
               exit repeat
            else
               get style of word j of line i of theDoc
               set theStyle to the result as list
               if the contents of item 2 of theStyle is equal to {bold} then
                  copy text of selection of theDoc to theParagraph
                  copy theParagraph & (ASCII character 13) ¬
                     to the insertion point after text of document "untitled-1"
                  set i to i + 1
               else
                  set i to i + 1
                  exit repeat
               end if
               exit repeat
            end if
         end repeat
      else
         set i to i + 1
      end if
   end repeat
end tell
```

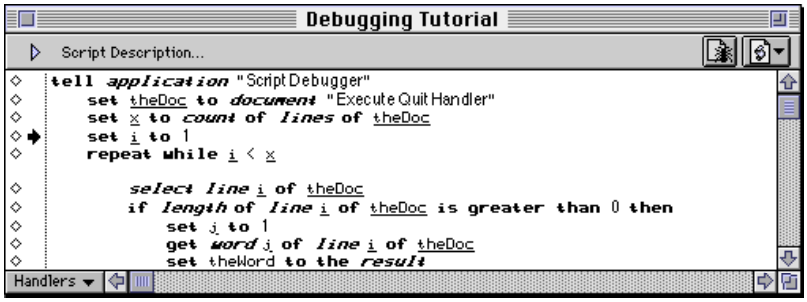
2. If you have not already done so, create a new script and open the Execute Quit Handler script that you will find in the Extensions folder inside the Script Debugger folder.

The Debugging Tutorial script checks the style of each line in the Execute Quit Handler looking for bold text in the line. When it finds one, it copies the paragraph to the Untitled-1 document. Now that you have the script open, let's step through it.

Stepping Through a Script

When you step through a script, Script Debugger moves an indicator along the left side of the Script window so you can track your progress *(Figure 4-4)*. Each time you select the Step command, Script Debugger executes the next line in your script. If your script requires interaction by selecting files or entering information, the dialog boxes will open for the information as you step through the script. If your script runs another program, as our example script does, the commands in your script will be executed in the background unless you specifically activate the program or an error brings it to the front.

Figure 4-4
A Script window with the step indicator in the left margin



As you step through this script, you might want to adjust your windows so you can watch Script Debugger's windows. If you shrink your windows, you will see that Script Debugger scrolls the window as you step through the script. Even the script Execute Quit Handler window scrolls in the background so that the selected line is always in view.

1. **Click on the Step button in the floating palette or select the Step command.**

Script Debugger takes a moment to compile the script and then the step indicator appears next to the first line of the script.

2. **Single-step through the script until you have seen the step indicator jump over the "if then" statement, which tests for the comment characters.**

The indicator returns to the "if then" statement that tests for the length of the line.

3. **Click the Stop button on the palette or select the Stop command to stop single- stepping through the script. You will not be able to edit the script until you stop the single-step execution.**

Now that you see how stepping though a script works, let's set a break-point into the script so that you can see how Script Debugger steps to breakpoints.

Setting Breakpoints

As you have probably noticed from the previous section, single-stepping through a script can take time. Often you may suspect that your script has a problem in one particular statement, subroutine, or handler, near the end of the script. Instead of single-stepping through the whole script, you can set a breakpoint just before the problem section. Then you can run the script until it reaches the breakpoint.

Another possible use of breakpoints is that you want to examine a variable to make certain that it is being updated properly. When you run a script, the values of a variable are updated and when the script stops, it will contain the very last value of the variable. To check a variable as it changes, you can set a breakpoint before the line which changes the variable. When you run the script, Script Debugger stops at the breakpoint and you can step through the line to see how the variable changes. You can then continue running the script. As with the previous example, setting the breakpoint allows you to check for problems in your scripts without single-stepping through the whole script.

If you look at the left side of your Script window, you will see a column of hollow diamonds. These are all of the potential breakpoints in your script. You can set a breakpoint by clicking on a diamond. Let's set a breakpoint at the second "if" statement.

1. Click on the diamond next to the line "if theWord begins with '—' then."

You will notice that it is filled when you click on it; that's your visual clue that the breakpoint is set.

2. Select Run or click the Run button on the palette. The pointer changes to the two gears while your script is running.

When it reaches the breakpoint, the step indicator is placed next to the breakpoint, and Script Debugger stops running the script.

3. Step until the "exit repeat" statement.
4. Now click on the Run button or select the Run command.

Script Debugger runs the script until it reaches the breakpoint again.

NOTE: *Script Debugger stores breakpoint and Data window information in Compile Script and Script Application documents. If you save one of these documents with Script Debugger, edit it with Script Editor and then try and open it with Script Debugger, your breakpoints will not be loaded since it cannot be certain that the lines match up with the breakpoints.*

NOTE: *As you step through this script, you will notice that the "i" variable does not get updated. That is because loop variables in AppleScript are local to the repeat loop. Declaring a global variable will not fix this problem. Due to limitations of AppleScript, Script Debugger cannot display the value of local variables.*

Clearing Breakpoints

After you have stepped through a script using breakpoints, you may want to remove them from the script. If your script is short and you have only set a few, you can probably click on them individually to disable them. If you have a longer script with breakpoints scattered throughout, you may want to use the Clear All Breakpoints command. Let's experiment with the example script to see how easy it is to clear breakpoints.

1. Set several breakpoints in the script.
2. Click on a couple of them so that you can see how easy it is to disable them.
3. Select Clear All Breakpoints from the Debug menu to remove the remainder of the breakpoints.

Unless you have disabled the Script Debugger icon in the Script window, it retains all of the breakpoints when you select the Save command. While you are developing a script, you will probably want to keep the breakpoints, but you should remove them when you finish the script.

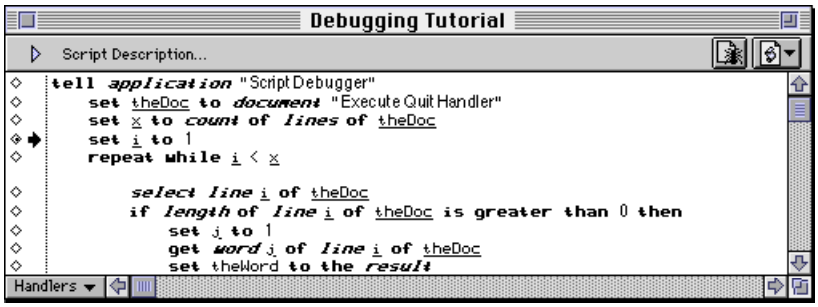
Setting Temporary Breakpoints

Occasionally, you may want to set a temporary breakpoint that is cleared as Script Debugger comes to it during the execution of the script. Temporary breakpoints are useful for skipping over repeat loops and handler calls when you run a script. You can create a temporary breakpoint by holding down the option key while you click on the breakpoint diamond. If you want to set a breakpoint without running the script, press the Option and Shift keys while clicking on the hollow diamond.

If you are still in single-step mode, click the Stop button on the Control window or type Command-. so that we can start fresh for this section. You may also want to open the Event Log window if you do not already have it open.

1. Press the Option key and click on the hollow diamond next to "set theStyle to the result as list" line. A dot appears inside of the hollow diamond (Figure 4-5).

Figure 4-5
Setting a temporary breakpoint in Script Debugger



Notice that Script Debugger compiles the script if it is not already compiled and then begins to run it. Script Debugger will run the script until it comes to the temporary breakpoint. Notice that the breakpoint is cleared when Script Debugger comes to it.

In our tutorial script, this runs quickly through all of the lines containing comments and comes to the section of the script that examines the line to see if it contains any bold formatting. The advantage of this is that we are able to jump to the portion of the script that does the real work.

2. Move the Event Log window so that you can watch while the script is running.

NOTE: *If the Event Log window is not open, it does not record the events while a script runs.*

3. Press the Option key and click on a diamond below the first temporary breakpoint to set another.

Notice as the script runs that the events begin to run up from the first breakpoint.

Now that you have been introduced to the Event Log window, let's see how you can use it for debugging your scripts.

Logging Apple Events

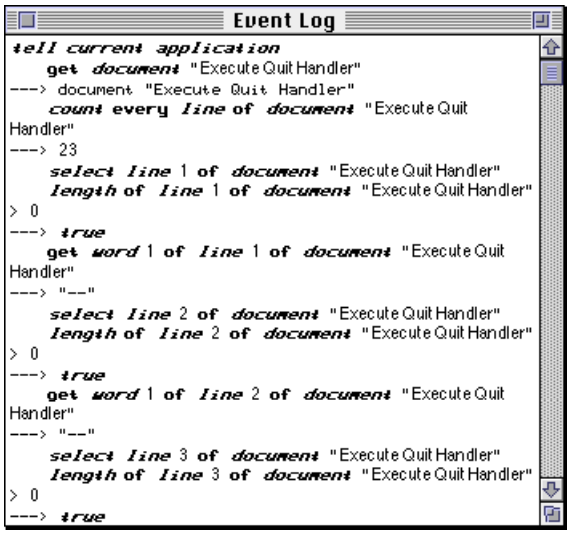
The Event Log window accumulates a running list of the Apple events executed by your script and the results that they return. The Event Log is cleared each time you run a script, and you can see it at any time by selecting Bring Event Log to Front from the Windows menu or by pressing Command-'.

The Event Log window has the advantage of being a history of a script's execution. After you have run a script, you can look at the Event Log window to see how each expression was evaluated. In addition to keeping a history of a script's execution, it is a real-time indicator of its progress. When you run a script in Script Debugger, the variables in the Data window are not updated until the script finishes running or is stopped. As a result, you only see the last changes to the variables. You can see the variables updated in the Event Log window as the script runs. To see how this works, let's run the Debugging Tutorial script that we were looking at.

1. Bring the Event Log window to the front.
2. Click on the Run button or select the Run command.

As the script executes, notice that after each event sent to Script Debugger, an arrow (—>) indicates what the event returns *(Figure 4-6)*. If you have specified different colors or fonts for the language keywords, references, variables, or any of the other AppleScript formatting items in the Preferences dialog box, you will see that Script Debugger uses those in the Event Log window.

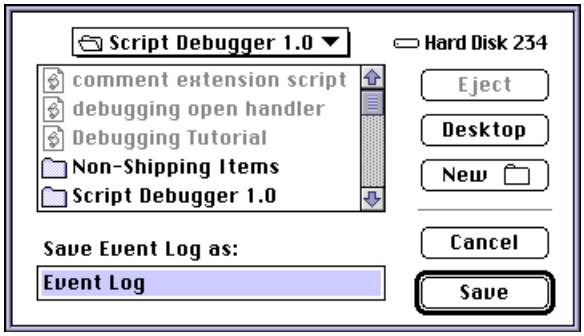
Figure 4-6
The Event Log window after running a script



When the script has completed, you can save the contents of the Event Log to a file. This makes it possible for you to compare the results of running a script to the script itself. To save your Event Log, use the Save As command.

3. Bring the Event Log window to the front.
4. Select the Save Event Log As command from the File menu. A standard Save As dialog box appears (Figure 4-7).

Figure 4-7
Saving the Event Log



5. Enter a name for the Event Log and click on the Save button. The log will be saved to disk as a Script Debugger text file. You can open the Event Log with any text editor.

Using the Data Window

The Event Log gives you continuous feedback while a script is running, but the Data window allows you to interactively examine expressions and watch them as your script runs. While Script Debugger does not save the contents of the Event Log window, it does save the contents of the Data window with your script. This gives a snapshot of the state of the variables and expressions as you stepped through your script.

At any point while you are stepping through your script, you can add an expression from your script to the Data window. If the expression in the script has not been evaluated when you add it, Script Debugger reports an error—"The variable i has not been defined yet", for instance. Once Script Debugger evaluates the expression, the evaluation appears next to the expression in the Data window.

Adding Expressions to the Data Window

Let's add expressions to the Data window for the Debugging Tutorial script.

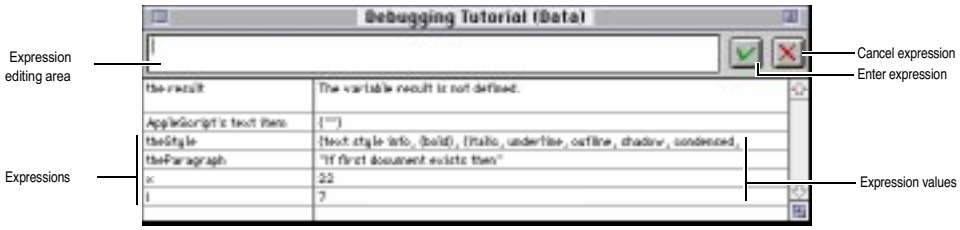
1. **Bring the Debugging Tutorial Data window to the front if you have not already done so.**

If you cannot see the window, you can bring it to the front by selecting Debugging Tutorial (Data) from the Windows menu.

2. **To enter an expression from the script, click in the expression editing field (Figure 4-8), enter the expression, and press the Return key or click on the Checkmark in the Data window.**

If you make a mistake while you are adding an expression, you can remove the expression from the editing area by clicking on the X button in the Data window.

Figure 4-8
The Data window



If you have not yet run the script, the expression value area will contain "The variable theParagraph is not defined." If you have run the script and you enter one of the expressions that has been evaluated, the current value of that expression appears in the value area.

You can also drag a variable and drop it into the expression editing area.

3. In the Script window, double-click on one of the variables that you have not entered to select it.

Click on it again and drag it over to the expression editing area and release the mouse button.

4. Now that the variable is in the expression area, press the return key to add it to the list of expressions.

Add the rest of the variables to the Data window.

5. After you have entered several of the variables from the script, run the script so that you will have values for all of the expressions in the Data window.

Drag and Drop Support for Values

In the same way that you can drag and drop a variable from the Script window into the Data window, you can also drag the values from the Data window to another application's window.

1. Open another Script window and select one of the values in the Debugging Tutorial script's Data window.
2. Click on the value a second time and then drag it to the new Script window that you opened.

Notice as you drag the value from the Data window that an outline follows the pointer. When you get over the new Script window, it is highlighted to show that you can drop the value in it.

3. Release the mouse button to drop the value into the Script window.

TIP: *You may want to drop values from the Data window into your script. For instance, if a value in the Data window contains a folder path or object reference, this is a quick way to get it in your script.*

If you have a text editor or word processor that supports Drag and Drop, you can drag the variables and their values into them. More importantly, you can drag a variable from your script into the expression editing field of the Data window. This saves you the time it takes to type the variables into the window.

Adding Properties to the Data Window

If your script has properties in it, you can quickly add them to the Data window using the Add Properties to Data Window extension script. When you run the script, it collects all of the properties in your script window and adds them into the Data window of the current script.

Removing Expressions from the Data Window

You can remove variables and expressions from the Data window individually or you can remove all of them at once.

1. Select the variable that you want to remove from the window.
2. Select Clear Expression from the Debug menu. You can also press Command-/.

You can also clear all of the expressions from the Data window at once.

3. Select Clear All Expressions from the Debug menu.

Selecting the Clear All Expressions command removes all of the expressions from the window including any that have been inserted by the Default Script. For instance, "the result" is inserted by the Default Script. After selecting the Clear All Expressions command, you may need to re-enter some of your expressions.

Examining Script Variables

If you removed all of the expressions in the previous section, you should add them back into the Data window for this portion of the tutorial.

Whenever you step through a script, you can examine the script's variables. As you have already noticed, when you add a variable to the Data window and then step through the script, you can watch as the variable is updated dynamically. You can also examine these variables at any time while you are stepping through the script. Let's step through the Debugging Tutorial script to see how you can watch and examine variables.

1. Select the Debugging Tutorial script window from the Windows menu if it is not the frontmost window.
2. Click on the Step button in the Control window or select Step from the Controls menu.
3. Step through the script, watching as the values of the variables change.

You will also notice as you step through the script that any line containing bold lines are being copied to the Untitled-1 window.

Examining AppleScript Expressions

You can enter expressions and one line scripts that you want to evaluate directly into the Data window.

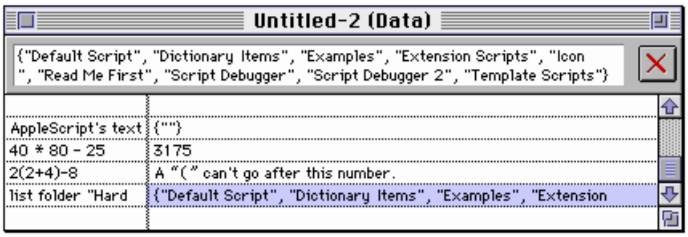
1. In the expression editing field, enter the expression:

```
40 * 80 - 25
```

2. Press Return or click the Checkmark to evaluate the expression.

Script Debugger enters the expression in the values area and places the value after it *(Figure 4-9)*.

Figure 4-9
Evaluating AppleScript expressions in the Data window



If the expression cannot be evaluated, Script Debugger enters the expression in the list of expressions and places an error message in the values area next to it.

3. Enter the expression:

```
2(2+4)-8
```

4. Press Return or click the Checkmark to evaluate the expression.

You will see an explanation of the error in the values area. You can also enter single line expressions in the expression editing area.

5. Enter the path to a folder in the expression editing area:

```
list folder "Hard Disk 234:Desktop Folder:Script Debugger
1.0:"
```

6. Press Return or click the Checkmark to evaluate the expression.

This expression requires the List Folder scripting addition. Script Debugger evaluates the expression and places a list with the contents of the folder that you selected into the value. If you click on the value, its contents are displayed in the expression editing area.

Restrictions

The Data window has a couple of restrictions. Any expressions you enter cannot reference local variables, and "the result" must appear as the first item in the Data window.

Due to the nature of AppleScript and the way Script Debugger operates, the expression "the result" must be the first item in the Data window. If you place "the result" after another expression in the Data window, the value shown for "the result" will be the value returned from the prior expression in the Data window. The Default Script supplied with Script Debugger has "the result" placed as the first item in the Data window. We recommend that you do not change this in the Default Script file.

You should avoid displaying long lists or record structures. You are not limited to the size of the value that you can display in the Data window, but long lists and record structures may slow down AppleScript and this will cause single-stepping to slow down. When you display large values in the Data window, Script Debugger only displays the first few thousand characters. However, when you copy or drag the value, you will get the entire value.

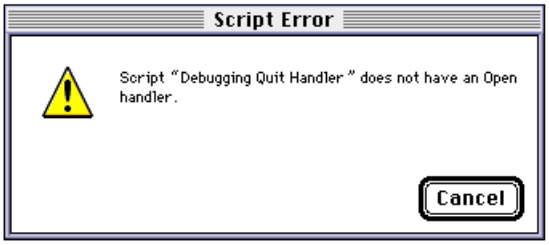
Debugging Open, Idle and Quit Handlers

Script Debugger normally executes the Run handler when you use the Run and Step commands. AppleScript defines three other standard handlers (open, idle and quit) which scripts can contain. Normally, you would have to write and test the scripts contained in the handlers before adding them to the handlers. Three of the extension scripts for Script Debugger allow you to step through open, idle, and quit handlers to debug them.

Executing an Open Handler

Open handlers are generally used for script applications or droplets. Usually, you would write and test all but the handler of the script in a script editor. Then you would add the handler and save the script as a script application. The Execute Open Handler (Folders...) and Execute Open Handler (Files...) extension scripts allow you to test the handler and script together by executing the open handler in the frontmost window. If the script does not contain an open handler, Script Debugger returns an error *(Figure 4-10)*.

Figure 4-10
The error returned if the script does not have an open handler



To see how this feature works in Script Debugger, let's look at a sample script.

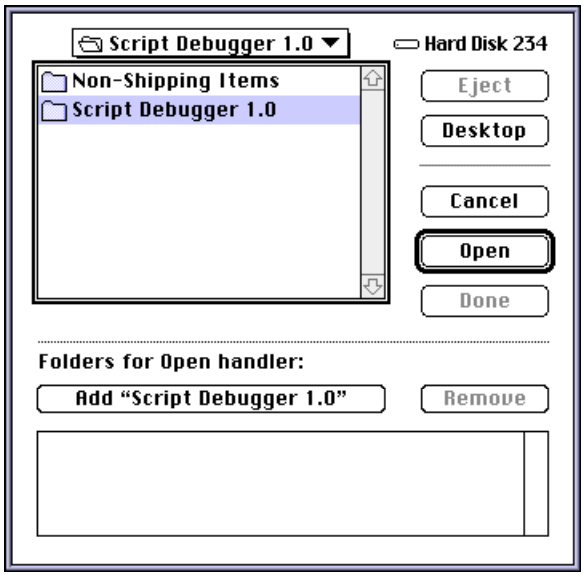
1. Open the Debugging Open Handler script. This is the script:

```
global theFiles

on open (theFolders)
   set theFiles to {}
   repeat with i from 1 to count theFolders
      set aFolder to item i of theFolders
      list folder (contents of aFolder)
      set theFiles to theFiles & the result
   end repeat
end open

on run
   open (choose folder with prompt "Select a folder to list...")
end run
```

2. Set a breakpoint on the "list folder (contents of aFolder)" line.
3. Add the expression theFiles to the Data window.
4. Select Execute Open Handler (Folders...) from Script Debugger's Extensions menu.

Script Debugger opens the Choose Folders dialog box where you can select the folder that you want to list (Figure 4-11).

Figure 4-11
The dialog box from the Execute Open Handler extension script



5. Choose a folder, click the Add button, and then click the Done button. Script Debugger executes the script up to the breakpoint.
6. Click the Step button in the Control window or select step from the Controls menu. Notice as you step through the next two lines of the script that the expression theFiles lists the contents of the folder you selected.

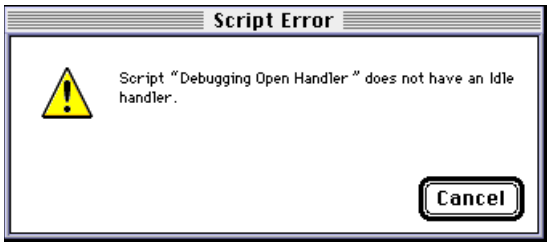
You can also use the Execute Open Handler (Files...) extension script in the same way if you have a script that collects information about a file or modifies the contents or attributes of a file.

Executing an Idle Handler

Typically, idle handlers are in Stay Open script applications. AppleScript periodically sends idle commands to script applications when a script application is not responding to incoming events. The Script Editor does not provide you with any way to debug an idle handler, but Script Debugger does in the Execute Idle Handler extension. It allows you to execute the handler without the need of adding to a larger script.

Like the Execute Open Handler extension scripts, the Execute Idle Handler script allows you to execute the idle handler in the frontmost window. If the script does not have an idle handler in it, Script Debugger signals an error *(Figure 4-12)*.

Figure 4-12
The error returned if the script does not have an idle handler



To see how you would test an idle handler script, let's look at a sample script.

1. Open the Debugging Idle Handler script. The script looks like this:

```
on idle
   set i to 1
   repeat with i from 1 to 10
      if i = 1 then
         beep 1
      else
         if i = 10 then
            beep 2
         end if
      end if
      set i to i + 1
   end repeat
end idle
```

This script does not do any work, but it shows you how you can debug an idle handler in the Script Debugger.

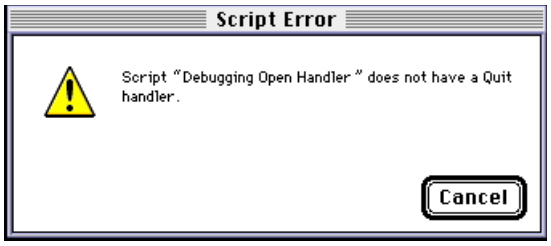
2. Set a breakpoint at the "if i = 1 then" line.
3. Add the variable i to the Data window. This allows you to see the variable incremented as Script Debugger loops through the script.
4. Select Execute Idle Handler from Script Debugger's Extensions menu. Script Debugger executes the script up to the breakpoint.
5. Click the Step button or select Step from the Controls menu. Step through the statements of the repeat loop several times to see the value of i incremented.
6. Click the Run button to finish executing the handler.

Executing a Quit Handler

Typically, quit handlers are in stay open script applications. AppleScript sends the script application a quit command whenever the user chooses Quit or presses Command-Q. Script Debugger provides you with a way to easily debug a quit handler in the Execute Quit Handler extension. It allows you to execute the handler without the need of adding to a larger script.

Like the other execute extension scripts, the Execute Quit Handler script allows you to execute the quit handler in the frontmost window. If the script does not have a quit handler in it, Script Debugger signals an error *(Figure 4-13)*.

Figure 4-13
The error returned if the script does not have a quit handler



To see how you would test a quit handler script, let's look at a sample script.

1. Open the Debugging Quit Handler script. The script looks like this:

```
on quit
   display dialog "Do you really want to quit?" ¬
      buttons {"Not really", "Quit"} default button "Quit"
   if the result is "Quit" then
      continue quit
   end if
end quit
```

2. Set a breakpoint at the "display dialog" line.
3. Select Execute Quit Handler from Script Debugger's Extensions menu. Script Debugger executes the script up to the breakpoint.
4. Click the Step button or select Step from the Controls menu. Step through the line with the dialog box and click on the Quit button to see the value returned as the result.

NOTE: *You always need to include the* `continue quit` *statement in your handler, otherwise you will have trouble quitting your script application. If you have failed to include the statement, you can always quit by pressing Command-Shift-Q. This will save any changes to the script's properties and quit immediately.*

Script Debugger Extensions

In addition to the debugging tools that are built into Script Debugger, you can also write your own extensions to add to the Extensions menu. Chapter 5 discusses Extension scripts, and you will find all of the documentation for Extension scripts in Appendix A.

Perhaps the best way to test Extension scripts that you are writing for Script Debugger is to run them from a second copy of Script Debugger or from Apple's Script Editor. This method would allow you to control Script Debugger and determine if your script is behaving as it should.

CHAPTER 5

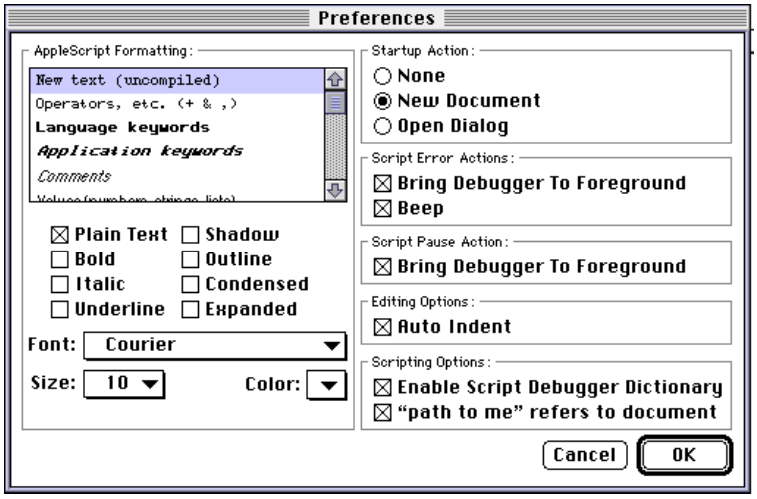
Customizing Script Debugger

Script Debugger is an attachable application. In other words, through scripts, you can customize it in a number of ways to make it better suit your needs. This chapter describes how you can customize the way Script Debugger formats your scripts and reacts to certain events. The chapter shows you how to create custom script templates. It shows you how to add items to the Extensions menu, and how to change Script Debugger commands through an Attachments script.

Changing Script Debugger Default Settings

Script Debugger allows you to alter how it displays AppleScript scripts and how the debugger responds to certain events. To change any of these settings, select the Preferences command from the Edit menu. Selecting this menu item opens the Preferences dialog box *(Figure 5-1)*.

Figure 5-1
Script Debugger's Preferences dialog box



AppleScript Formatting

This group of settings lets you change the way AppleScript formats your scripts. The scrolling list shows the different elements of a script.

NOTE: *The formatting changes that you make in Script Debugger's Preferences dialog change the formatting of the English AppleScript system and will carry over to the Script Editor and any other application which displays AppleScript scripts.*

New text is anything that you have entered into your script but have not yet compiled. In other words, it is the script before you click on the Compile button.

Operators that you might use in your scripts are the plus (+) and minus (-) signs, the less than (<) and greater than (>) symbols, and the equal sign (=).

Language keywords are the AppleScript keywords that appear in your script.

Application keywords are the keywords from the dictionary of the application or scripting addition that you are scripting.

Comments are the portions of the script set off by dashes (—) or by blocks. For example, (* this is a block comment*).

Values are the dates, numbers, strings, and lists that AppleScript returns whenever expressions are evaluated.

Variables and subroutine names are the names that you enter in your scripts.

References are the phrases that you use to specify objects within applications.

You can adjust the text style, font, font size, and color for each of these items. To change the style of one of the elements:

1. Select the element that you want to change from the scrolling list.
2. Adjust the style of the font by clicking on the checkboxes.
3. Adjust the font and the font size using the pop-up menus.
4. Adjust the color of the element using the color pop-up menu.

You can change another element by selecting it from the list. Closing the Preferences dialog box saves your changes.

While color scripting may seem frivolous, it can help you understand scripts and find problems quickly. For instance, you might want to use color for the language and application keywords, the values, and the variables in your scripts. While the colors do not carry over to the Data window, they do show up in the Event Log and the Result windows. If you are tracing the progress of a variable, you can spot it quickly in the Event Log by watching for the color you have assigned to variables.

NOTE: *After you change AppleScript formatting settings, Script Debugger will reformat all open scripts and dictionaries. There may be a brief pause while this takes place.*

Startup Action

The Startup Action group of radio buttons allows you to control what Script Debugger does when you double-click on its icon. The options are:

None When this radio button is selected, Script Debugger does not perform another action when it starts up. In other words, Script Debugger does not open a new, untitled script nor open a file selection dialog.

New Document
When this radio button is selected, Script Debugger creates a new, untitled script. Script Debugger uses the Default Script as the template for the new script. This is the default option setting when Script Debugger is installed.

Open Dialog
When this radio button is selected, Script Debugger presents a file selection dialog box when it starts up. This allows you to choose a script when starting Script Debugger.

Script Error Actions

This group of items lets you control how Script Debugger operates when there is an error in your script.

Bring Debugger to Foreground
When this item is selected, Script Debugger comes to the foreground whenever there is a script error.

Beep When this item is selected, Script Debugger plays the beep sound whenever there is a script error.

Script Pause Action

The Script Pause Action allows you to control how Script Debugger operates when a script pauses or reaches a breakpoint.

Bring Debugger to Foreground

When this item is selected, Script Debugger comes to the foreground whenever a script is paused or Script Debugger encounters a breakpoint.

For example, if you are stepping through a script which activates the Scriptable Finder, selecting this option brings Script Debugger to the foreground each time you pause. (If you are single-stepping through the script, you pause at each line.) As you step, Script Debugger activates the Scriptable Finder to handle the next event and then Script Debugger comes back to the foreground so you can step again.

This option relies on an auto-activate feature. Script Debugger tries very hard to track which application is the foreground application. It does this so that it can re-activate the foreground application whenever you step over a statement which sends an event. The idea is to ensure, as much as possible, that the application environment is correct.

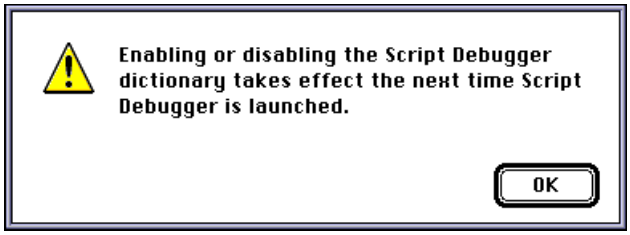
Editing Options

When the Auto Indent checkbox is checked, Script Debugger automatically indents your script as you enter it.

Scripting Options

If you want to write scripts for the Script Debugger, you need to enable the Script Debugger Dictionary checkbox in the Preferences dialog box. After enabling the dictionary, you must quit Script Debugger and restart it to make the changes take effect. Script Debugger warns you that you need to do this *(Figure 5-2)*.

Figure 5-2
Enabling the Script Debugger dictionary



Selecting the Enable Script Debugger Dictionary option makes the Script Debugger's dictionary available. When this option is not selected, nothing—not even Script Debugger—can display the dictionary. When it is selected, Script Debugger intercepts the Get AETE event and returns the appropriate aete data. Script Debugger must take this approach to allow you to write non-Script Debugger scripts which use terms that might conflict with the terms in the Script Debugger dictionary.

NOTE: *At present, the only known conflict is with the offset property.*

Enabling the Script Debugger Dictionary also enables recording. This allows you to record operations done in Script Debugger, just as you can record operations performed in other applications. If you do not want Script Debugger to record its own functions, disable the Script Debugger Dictionary.

Select the "path to me" checkbox if you want Script Debugger to intercept the "path to me" command. Doing this causes Script Debugger to return the path to the document for the command. This is useful if you save a script as a script application or droplet. In this case, the "path to me" command returns the same value when the script is executed within Script Debugger, and when it's executed as a stand alone application.

If you do not select this checkbox, Script Debugger returns the path to itself for the command. This option only has an effect if you save the script. If the script is new and unsaved, this setting has no effect—the path to Script Debugger is returned in both cases.

Changing Defaults for New Scripts

The Default Script stationary pad in the Script Debugger folder supplies all defaults for new scripts. This stationary pad file can supply defaults for every aspect of a new script. When you save a new script document as a stationary pad, it retains the window settings and locations, font and color settings, and script text.

You can hold down the Option key while selecting Default Script from the New menu to open the Default Script directly and retain the stationary pad settings.

To change defaults for new scripts, follow these steps:

1. Open a new document.

Use the New command from the File menu to create a new untitled script document.

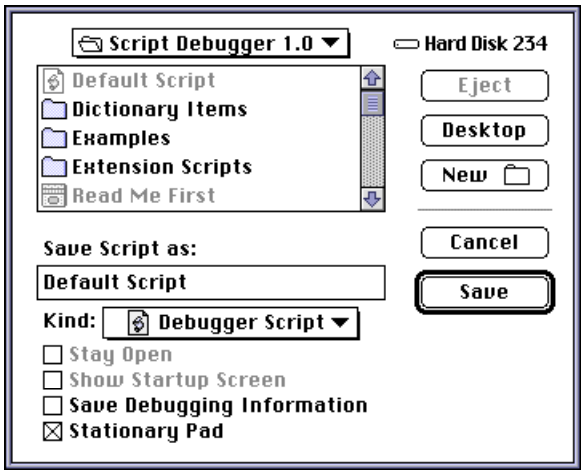
2. Alter the new document.

Modify the new script so that it has the settings you prefer. You can enter a default script description, default script text and default data expressions. You can also hide or show the script description and save the window location. You can even set breakpoints if you have default script text.

3. Save the new document.

Select Save As from the File menu. When the Save As dialog box appears *(Figure 5-3)*, name your script "Default Script" and select the Stationary Pad checkbox. Save the script in the Script Debugger folder.

Figure 5-3
Creating a new default script



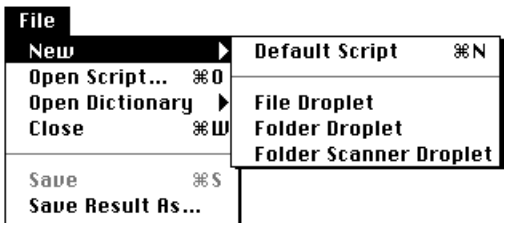
4. Replace the old Default Script.

When you click on the Save button, Script Debugger asks if the old Default Script file should be replaced. Answer Yes.

Adding Templates to the Templates Menu

Script Debugger's Template Scripts folder contains three script templates: Folder Droplet; File Droplet; and Folder Scanner Droplet. You can see these templates under the New menu *(Figure 5-4)*. These three template scripts are the beginnings of general purpose droplet scripts. They each provide handlers with comments that tell you where to place your script for processing files or folders.

Figure 5-4
The template scripts in the File menu



You may add your own template scripts to the New menu by placing scripts saved as stationary pads in the Template Scripts folder.

To add a template script to the New menu, follow these steps:

1. Open a new document.

Use the New command from the File menu to create a new, untitled script document.

2. Alter the new document.

Create a specialized template script with the handlers and script text you need. You can enter a script description, text, and data expressions. You can also hide or show the script description and save the window location. You can even set breakpoints if you have entered script text.

3. Save the new document.

Select Save As from the File menu. When the Save As dialog box appears *(Figure 5-5)*, give your template script a name and select the Stationary Pad checkbox. Save the script in the Template Scripts folder.

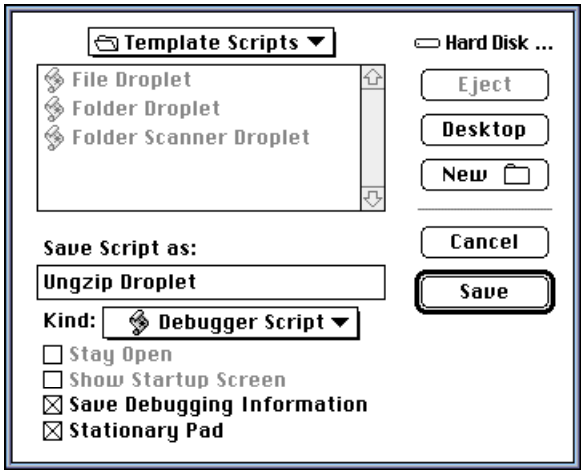


Figure 5-5
Creating a template script

NOTE: *Non-stationary pad files stored in the Template Scripts folder are listed in the Templates menu. However, if your template is not stored as a stationary pad, Script Debugger will open the template as a normal document. In other words, it will not use the template to create a new, untitled document, and you may overwrite your template script.*

Adding Applications to the Open Dictionary Menu

The Open Dictionary menu allows you to quickly display dictionary information for applications and scripting additions without having to use a file selection dialog box. You can add the names of the scriptable applications and scripting additions you use frequently to the Open Dictionary menu.

To add an item to the Open Dictionary menu, do the following:

1. Make an alias for the application or scripting addition.

Use the Finder to make an alias for the application or scripting addition you want to add to the Open Dictionary menu.

2. Move the alias to the Dictionary Items folder.

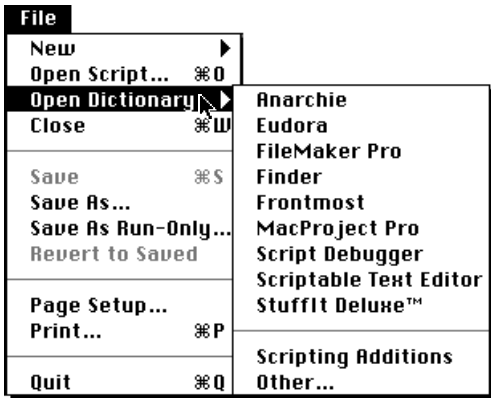
Use the Finder to move the alias you created in step 1 to the Dictionary Items folder. The Dictionary Items folder is located within the Script Debugger folder.

Figure 5-6 gives an example of a Dictionary Items folder containing a series of aliases to commonly used scriptable applications.

Figure 5-6
The Dictionary Items folder

The resulting Open Dictionary menu is shown in Figure 5-7.

Figure 5-7
The Open Dictionary menu



You can also use the Add to Open Dictionary Menu extension script to add an application's dictionary to the menu. If you have a Dictionary window open, you can select the Add to Open Dictionary Menu script from the Extensions menu. The script adds an alias of the application to the Dictionary Items folder.

NOTE: *If you add an application to the Dictionary Items folder (in other words, the program and not the alias), it will not be listed in the Open Dictionary menu.*

Adding Commands to the Extensions Menu

The Extensions menu allows you to add new commands to Script Debugger. Each item listed in the Extensions menu corresponds to an AppleScript script application file or compiled script stored in the Extension Scripts folder inside the Script Debugger folder.

To add a new item to this menu, follow these steps:

1. Create a script.

Using Script Debugger, create a script which you want to include in the Extensions menu.

2. Save the script in the Extension Scripts folder.

Save your script in the Extension Scripts folder within the Script Debugger folder as a script application (applet or droplet) or as a compiled script. If you save your script in any other format, such as a Debugger Script or Text File, Script Debugger will not include your file in the Extensions menu.

You can add a keyboard command to your Extension script by adding "|<the key>" to the Extension script's name. For instance, if you wanted to add Command-H to the Hide Descriptions extension, you would rename it Hide Descriptions|H. The Extensions menu would change accordingly *(Figure 5-8)*.

Figure 5-8
A keyboard command added to an Extension script

Attaching Scripts to the Menu Items

We have already pointed out elsewhere in this manual that Script Debugger is scriptable, recordable, and attachable. Like other scriptable applications, Script Debugger can be scripted and recorded. More importantly, you can write scripts that extend and add to Script Debugger's functionality.

You have already seen how you can add commands to Script Debugger by adding scripts to the Extensions folder. Script Debugger is also attachable; you can write scripts and attach them to commands in Script Debugger's menus. This allows you to modify the way in which some of Script Debugger's commands work.

The rest of this section explains how attachments work, outlines which commands can be modified, and describes how to write attachment handlers. For more information about Script Debugger's dictionary, refer to Appendix B.

How Attachments Work

When Script Debugger starts up, it looks for a compiled script called "Attachments" in the Script Debugger folder. If the Attachments script exists, Script Debugger loads it and uses it to handle attachable commands. When you issue one of the commands listed in *Table 5-1*, Script Debugger uses the handler in the Attachments script for the command. If the Attachments script does not contain a handler for the command, or if an Attachments script is not present, Script Debugger handles the command normally.

Table 5-1: ATTACHABLE COMMANDS		
Command	**Menu Command**	**AppleEvent/Handler**
Create a new script	File/New/Default Script	make new document
Open a script document	File/Open	open
Close an open script	File/Close	close
Compile a script	Controls/Compile	compile
Run a script	Controls/Run	execute, execute open, execute idle & execute quit
Step a script	Controls/Step	step
Pause a script	Controls/Pause	pause
When a script pauses (following a step, or when a breakpoint is encountered)	n/a	pause
Stop a script	Controls/Stop	stop
When a script completes (following a stop, following an error or when the script completes)	n/a	stop
Start recording	Controls/Record	record
Save a script	File/Save, File/Save As, File/Save As Run-Only	save
Revert to saved	File/Revert To Saved	revert
Open an application dictionary	Open Dictionary/Other	open dictionary
Open the Scripting Additions dictionary	Open Dictionary/Scripting Additions	open additions dictionary
Close an open dictionary	File/Close	close

NOTE: *Before executing an attachable command, Script Debugger checks the modification date of the Attachments script file. If the file has been changed since Script Debugger last loaded it, it reloads the script. If the Attachments script has been removed or renamed, Script Debugger stops using the attachment handlers.*

Creating Attachment Handlers

To attach a script to a Script Debugger command, you must create a handler for the command's Apple event and add it to the Attachments script. For examples of attachment handers, look at the Attachment scripts in the Auto-Add to Dictionary and Every Possible Handler folders in the Attachments Examples folder. In particular, the Every Possible Handler script has example handlers for all of the commands that support attachment.

For instance, if you wanted to attach a script to Script Debugger's Open Script command, you would add statements inside the open handler.

```
on open of theFile
   tell application "Finder"
      set date1 to modification date of theFile as string
      set date2 to creation date of theFile as string
   end tell
   display dialog "Created:  " & date2 & return & ¬
        "Last modified:  " & date1 buttons {"Cancel", "OK"} default button 2
   continue open theFile
end open
```

This particular script uses the Scriptable Finder to get the file information about the file that you are about to open. It puts the creation and modification date into variables and then displays a dialog box with that information before the document is opened. Notice that we do not have to insert a line to choose the file; Script Debugger takes care of that before calling the handler. Another important item to notice in this handler is the `continue open theFile` line before the end of the handler. If this is not included, Script Debugger does not open the file. You should also notice that the display dialog line is outside of the tell Finder block so the dialog is displayed in Script Debugger. You could easily modify this script to use scripting additions to get the file information and reformat it for display in the dialog box.

Debugging Extension and Attachment Scripts

It is sometimes difficult to debug Script Debugger extension scripts and attachment handlers directly within Script Debugger. One way to solve this problem is to use a second copy of Script Debugger. When you use two copies of Script Debugger, one copy can host the script, and the other can respond to Apple events.

When you run two copies of Script Debugger, you should make sure to send the events to the correct copy. Renaming the second copy of Script Debugger makes this a little easier.

CHAPTER 6

Common Problems and Troubleshooting

While every effort has been taken to make Script Debugger a solid and bug-free program, you may still experience some problems while you are using it. Some of these problems stem from the interaction of AppleScript and scriptable programs. This section lists some common problems and offers suggestions for working around them.

page 225

Use the Include Enclosed Folders checkbox to indicate whether or not you want Scheduler to include the files stored in enclosed folders when it calculates the size of the folder.

front matter

page 121

atObject	The name of an AppleTalk network object
atType	The type name of an AppleTalk network object
atZone	The zone name of an AppleTalk network object

PROBLEM I can't compile Script Debugger scripts.

ANSWER The Script Debugger dictionary is disabled. You can enable it by clicking in the Enable Script Debugger Dictionary checkbox in the Preferences dialog. See the *Scripting Options* section of Chapter 5*: Customizing Script Debugger,* for more information.

PROBLEM I can't record Script Debugger actions.

ANSWER Recording is only available when the Script Debugger dictionary is enabled. To record Script Debugger's actions, you must click on the Enable Script Debugger Dictionary checkbox in the Preferences dialog. See the *Scripting Options* section of Chapter 5*: Customizing Script Debugger,* for more information.

PROBLEM I can't display the Script Debugger dictionary.

ANSWER The Script Debugger dictionary is disabled. You can enable it by clicking in the Enable Script Debugger Dictionary checkbox in the Preferences dialog. See the *Scripting Options* section of Chapter 5*: Customizing Script Debugger,* for more information.

PROBLEM I can't open Script Debugger script files with Apple's Script Editor.

ANSWER Your Script Debugger scripts must be saved in Text, Compiled Script, or Script Application format before they can be opened by Apple's Script Editor.

PROBLEM The breakpoint information in my compiled script is missing.

ANSWER If you save a compiled script or a script application with debugging information and then edit the file using Apple's Script Editor, all of the breakpoint information will be lost when you open the script again using Script Debugger.

PROBLEM Script Debugger's performance is poor when single-stepping through a script.

ANSWER If you have an excessive number of expressions in the Data window, Script Debugger slows down when single-stepping. Expressions that result in long text values, record structures, or object specifiers are particularly time consuming to update.

PROBLEM I can't debug open, idle or quit handlers.

ANSWER 1 Confirm that your script contains an open, idle or quit handler and that it's compiled.

ANSWER 2 Make sure that you have a breakpoint set in your handler, and that you are executing it using the appropriate Extensions script.

PROBLEM When I try to open a script, Script Debugger tells me that it can't open the script because it was created by a later version of Script Debugger *(Figure 6-1)*.

ANSWER You will need to upgrade to the latest version of Script Debugger.

NOTE: *You can still open the file using Apple's Script Editor. You will lose any debugging and data saved with the script.*

Figure 6-1
Script Debugger can't open a file created by a later version

PROBLEM I can't use Drag and Drop to invoke open handler.

ANSWER 1 Make sure the script is compiled.

ANSWER 2 Make sure your script has an open handler.

ANSWER 3 Make sure you have a breakpoint set in the open handler.

ANSWER 4 Make sure your Macintosh has the Drag and Drop extension installed or you are running System 7.5.

ANSWER 5 Make sure you are dragging a file (application, document, etc.) or a folder. Script Debugger will not invoke the open handler if you drag text or a text clipping.

PROBLEM I get an out of memory (-108) error when displaying the Scripting Additions dictionary.

ANSWER 1 If you have a large number of third party scripting additions installed, you may need to expand Script Debugger's memory partition before you can display the Scripting Additions Dictionary window.

ANSWER 2 Script Debugger stores some of the information it needs to display the dictionary using "Temporary Memory." If your system is low on free memory, you do not have enough space available for Script Debugger to display the dictionary. Try closing other applications that you may have running.

PROBLEM I've written an attachment script for Script Debugger, but now I can't quit the program.

ANSWER When you are writing scripts for the attachability features in Script Debugger, you may get into a state where you can't quit the program. This is caused when an attachment for the Close command generates an error. The solution is to simply move the Attachments file out of the Script Debugger folder and try the operation again.

This solution can be used to solve any problems caused by attachments. It works because Script Debugger checks the Attachments file and executes any handlers it finds there each time you select a command.

PROBLEM The identifiers in my script are in the |ident| style. (For more information about identifiers, refer to the *AppleScript Language Guide*, p. 28.)

ANSWER If you open a script using Script Debugger which contains identifiers that conflict with the identifiers in the Script Debugger dictionary, the identifiers in the script are converted to the |ident| style. To avoid this, uncheck the Enable Script Debugger Dictionary checkbox in the Preferences dialog box.

APPENDIX A

Extension Scripts

You can add to Script Debugger's features by writing Extension scripts and placing them in the Extensions folder. This appendix describes the Extension scripts that are shipped with Script Debugger. It explains how to use them and provides some tips that might not be obvious. If you are interested in writing Extension scripts for Script Debugger, be sure to examine the way these scripts are written.

Script Debugger Extensions

Script Debugger provides a special Extensions menu *(Figure A-1)* where you can add additional commands to the program. These commands invoke AppleScript scripts that use the Script Debugger scripting interface to add functionality to the program.

Script Debugger comes with twelve Extension scripts that are documented in this section. If you want to add Extension scripts to this menu, see Chapter 5, *Customizing Script Debugger*.

Figure A-1
Script Debugger's Extensions menu

Add Properties To Data Window

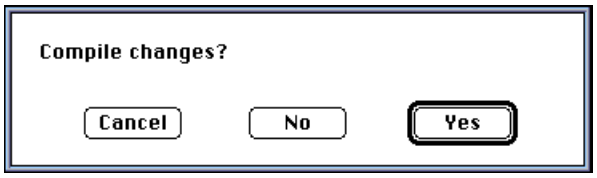
The Add Properties to Data Window extension script allows you to set up the Data window with the object properties from the Script window. This can save time since you do not have to add each property in the script individually.

To add the properties to the Data window,

1. Bring the Script window to the front.
2. Select the Add Properties to Data Window command from the Extensions menu.

If you have made any changes to the script, the script extension prompts you to compile the changes *(Figure A-2)*. This allows Script Debugger to add any additional globals you have added to the script since the last time you compiled it. Script Debugger then adds the object properties to the Data window.

Figure A-2
Compile changes prompt



If the script has been compiled, this dialog box does not appear.

Add To Open Dictionary Menu

The Add to Open Dictionary Menu extension script adds an application or scripting addition to the Open Dictionary menu. When you execute the script, it gets the name of the frontmost application or scripting addition dictionary window. It then creates an alias of the application or scripting addition in the Dictionary Items folder. To add an application's dictionary to the menu, follow these steps:

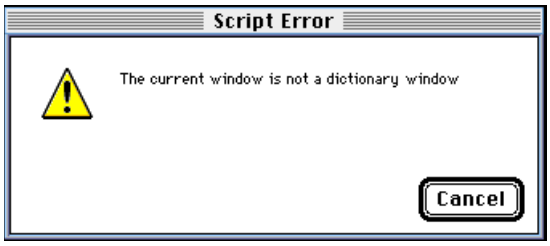
NOTE: *This script uses the Scriptable Finder.*

1. Open an application's or scripting addition's dictionary using the Other sub-menu under the Open Dictionary menu.
2. Choose Add to Open Dictionary Menu from the Extensions menu.

The script creates an alias in the Dictionary Items folder for the application.

NOTE: *Make certain that you have a Dictionary window in the foreground when you choose Add to Open Dictionary Menu. If you do not, Script Debugger informs you that the current window is not a Dictionary window (Figure A-3).*

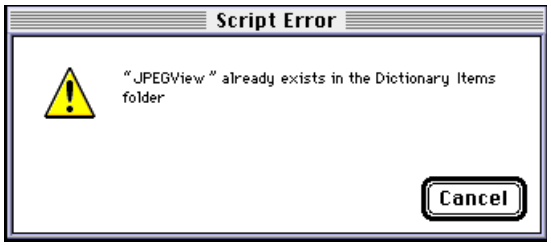
Figure A-3
Not a Dictionary window error message



This script uses the Gestalt scripting addition from Late Night Software Scripting Additions and the Scriptable Finder. If you do not have the Scriptable Finder installed, an error dialog box appears with the message that the Scriptable Finder is not installed. See Appendix D, *More Information about AppleScript*, for information about getting the scriptable Finder.

If you try to add a Dictionary window that is already in the Open Dictionary menu, you will see an error message *(Figure A-4)*.

Figure A-4
An alias already exists in the Dictionary Items folder



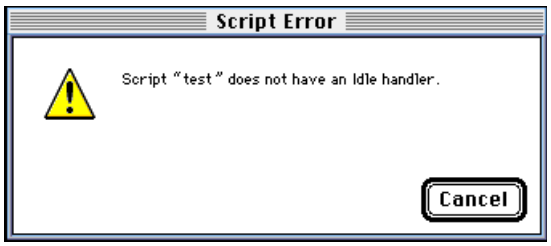
Execute Idle Handler

The Execute Idle Handler extension script invokes the idle handler in the current script. Like the Execute Open and Execute Quit scripts, this script allows you to test and debug idle handlers without the need to modify your script after you have finished testing it.

The best way to test an idle handler is to set a breakpoint in the handler, and then select the Execute Idle Handler script. The breakpoint stops the script in the handler and you can then single-step through it.

NOTE: *If the current window does not have an idle handler in it, Script Debugger returns an error message (Figure A-5).*

Figure A-5
No Idle handler error message



Script Debugger also returns an error message if the script in the current window is running or if you are recording a script in the current window when you invoke the Execute Idle Handler script *(Figure A-6)*.

Figure A-6
Script running or recording error message

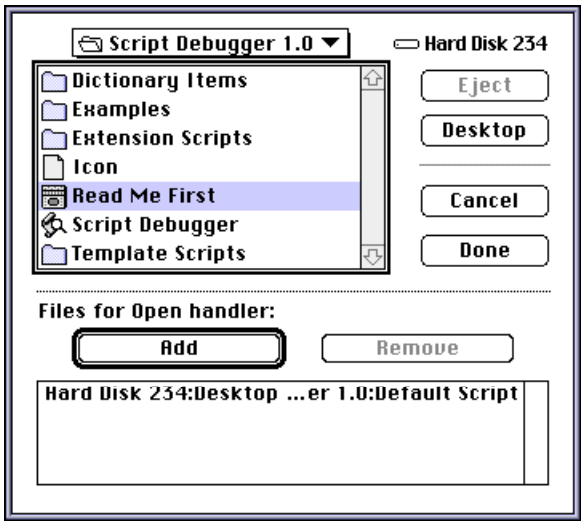
Execute Open Handler (Files). . .

The Execute Open Handler (Files) extension script invokes the open handler of the current Script window. If you want to take control of the script once the handler has been invoked, you should set a breakpoint in the Open handler. This stops the script in the handler so that you can single-step through it.

To execute the script,

1. Choose Execute Open Handler (Files) from the Extensions menu.
2. Select the files for the Open handler in the dialog box (Figure A-7) and click the Add button.

Figure A-7
The Add Files dialog box



As you select files, they are added to the scrolling list at the bottom of the dialog box.

3. Click the Done button and Script Debugger passes the selected files to the Open handler as a parameter, and executes it.

If you have Macintosh Drag and Drop installed, you can use the Drag and Drop feature to execute your Open handler. You do this by dropping a file or files on the open Script window. For more information about this feature, refer to Chapter 3, *Creating and Editing Scripts*.

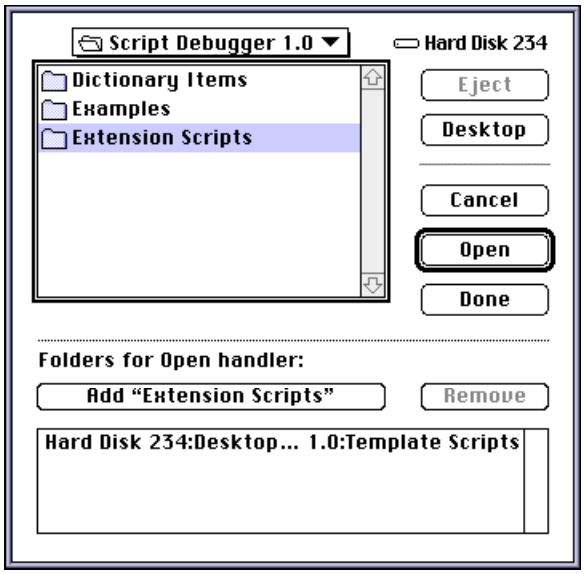
Execute Open Handler (Folders). . .

The Execute Open Handler (Folders) extension script invokes the open handler of the current Script window. If you want to take control of the script once the handler has been invoked, you should set a breakpoint in the Open handler. This stops the script in the handler so that you can single-step through it.

To execute the script,

1. Choose Execute Open Handler (Folders) from the Extensions menu.
2. Select the folders for the Open handler in the dialog box (Figure A-8) and click the Add button.

Figure A-8
The Add Files dialog box



3. Click the Done button and Script Debugger passes the selected folders to the open handler as a parameter, and executes it.

If you have Macintosh Drag and Drop installed, you can use the Drag and Drop feature to execute your Open handler. You do this by dropping a folder or folders on the Open Script window. For more information about this feature, refer to Chapter 3, *Creating and Editing Scripts*.

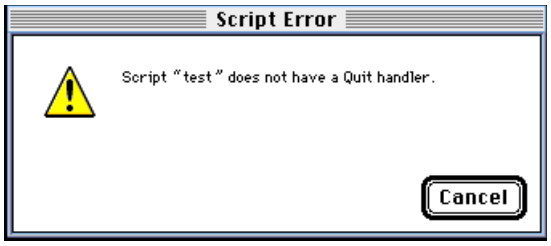
Execute Quit Handler

The Execute Quit Handler extension script invokes the quit handler in the current script. As with the other Execute scripts, this script provides you with a way to debug a Quit handler without having to edit the script.

If your script needs to be compiled, this script will compile it before the handler is executed.

NOTE: *If the current window does not have a quit handler in it when you choose Execute Quit Handler from the Extensions menu, Script Debugger returns an error message (Figure A-9).*

Figure A-9
No Quit Handler error message



Hide and Show Descriptions

The Hide Descriptions extension script lets you hide the description areas in all currently open scripts. This is quicker than bringing each window to the front and clicking on its Script Description triangle. To hide all descriptions, choose Hide Descriptions from the Extensions menu.

You can reverse the effect of the Hide Descriptions script by choosing Show Descriptions from the Extensions menu. The Show Descriptions extension script lets you show the description areas in any currently opened scripts. This is quicker than bringing each window to the front and clicking on its Script Description triangle. To show all descriptions, choose Show Descriptions from the Extensions menu.

If you do not have any Script windows open, Script Debugger returns an error *(Figure A-10).*

Figure A-10
No open documents error message

Lock and Unlock All Expressions

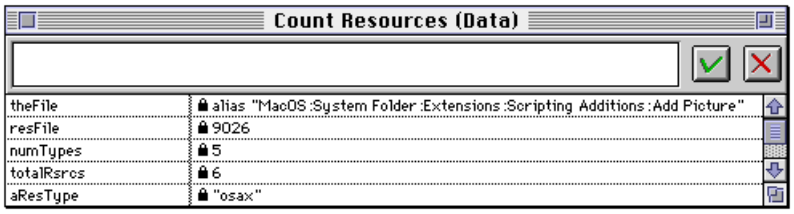
The Lock All Expressions extensions script allows you to lock the contents of the expressions in your Data window. This preserves the values of the expressions in the Data window. If you run your script again after you have locked it, the expression values are not changed. Another advantage of locking expressions is that it speeds up the performance of your script when you are single-stepping.

To lock the expressions in your script,

1. Bring the Script or Data window to the front.
2. Choose Lock All Expressions from the Extensions menu.

You will notice that locks appear next to the expressions in the Data window *(Figure A-11)*.

Figure A-11
A Data window with Locked Expressions



Now, if you run your script again, the values of the expressions will not change. If you watch the progress of your script in the Event Log window, you can see that the expressions are being updated, but their values in the Data window are not updated.

The Unlock All Expressions script reverses the effects of the Lock All Expressions script. To unlock the expressions in your script,

1. Bring the Script or Data window to the front.

2. Choose Unlock All Expressions from the Extensions menu.

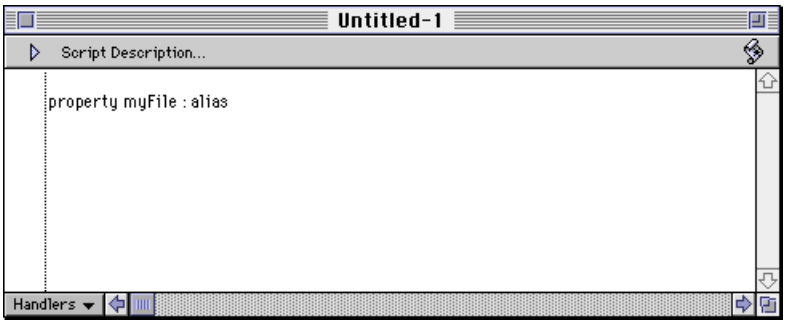
Now, if you run your script again, the values of the expressions will be updated as they normally would.

Paste File Path…

The Paste File Path extension script pastes the path to a file into a script. You can use this to quickly place the location of files in the scripts you write. For instance, if you want to define an AppleScript property containing an alias to a particular file, follow these steps:

1. Enter the beginning of a property definition into your script (Figure A-12).

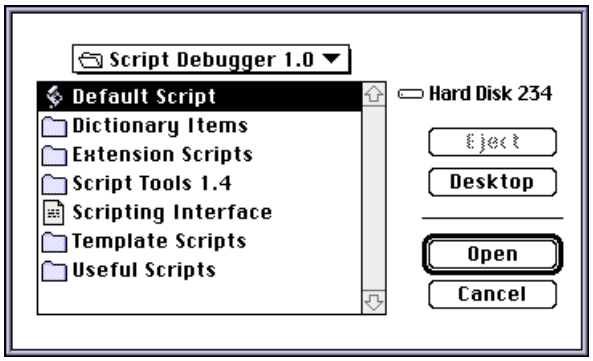
Figure A-12
Property definition for a file alias



2. Choose the Paste File Path command in the Extensions menu.

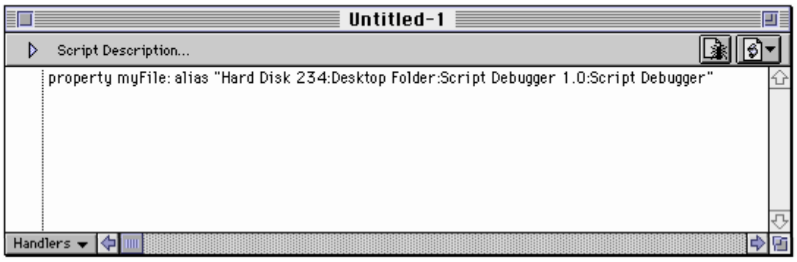
When the file selection dialog box opens *(Figure A-13)*, locate the file whose path you want to place in the script.

Figure A-13
The File Selection dialog box



Once the Open button is pressed, the file's path is added to your script *(Figure A-14)*.

Figure A-14
A file's pathname pasted into a script



Paste Folder Path...

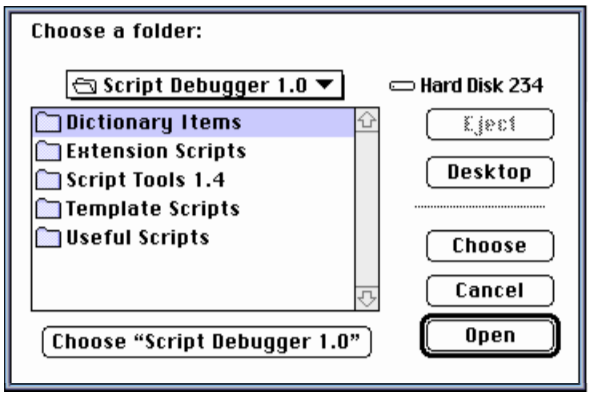
The Paste Folder Path extension script is similar to the Paste File Path extension script. Instead of pasting a file pathname into your script, however, it pastes a folder's pathname into your script.

To paste a folder path into your script,

1. Choose Paste Folder Path from the Extensions menu.

A folder selection dialog box opens *(Figure A-15)*.

Figure A-15
The Folder Selection dialog box



2. Select the folder whose path you want to add to your script and click the Choose button.

When you have selected a folder, the dialog box closes and the path is pasted into your script window *(Figure A-16)*.

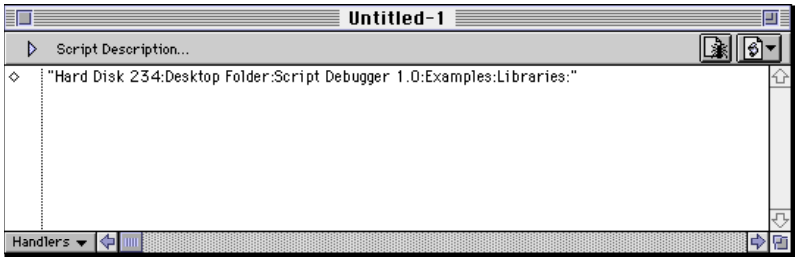


Figure A-16
A folder's pathname pasted into a script

APPENDIX B

Script Debugger Scripting Interface

This appendix describes the objects available in Script Debugger's dictionary and the commands that you can use to manipulate these objects and their contents. The purpose of the discussion in this appendix is to make it easier for you to write scripts extending and modifying Script Debugger's behavior. You can find information about all of Script Debugger's objects and commands by opening its dictionary using the Open Dictionary Other command in the File menu.

Scripting Script Debugger

As we discussed in Chapter 5, *Customizing Script Debugger*, you can write scripts that add commands to Script Debugger's Extensions menu and modify the behavior of Script Debugger's commands. Toward that end, this section of the manual briefly elaborates on the description of the commands that you find in Script Debugger's dictionary.

NOTE: *To use the Script Debugger scripting interface effectively, you must enable the Script Debugger dictionary using the Preferences command. When you no longer need the Script Debugger dictionary, disable it to avoid terminology conflicts with Scripting Additions and other applications. Please refer to Chapter 5 for more details about the Preferences command.*

Certain Restrictions

When you are scripting Script Debugger, you should be aware of certain restrictions that affect any event which causes a script to execute. These events are the step, execute, execute open, and other commands.

Script Debugger only allows one script to be executing at any given time. If you have a script paused in one window, you cannot begin executing or recording a script in another window. Whenever you cannot run a script, check the other open windows to make certain that you have not paused a script. One exception to this is that you can compile a script in another window while a script is paused.

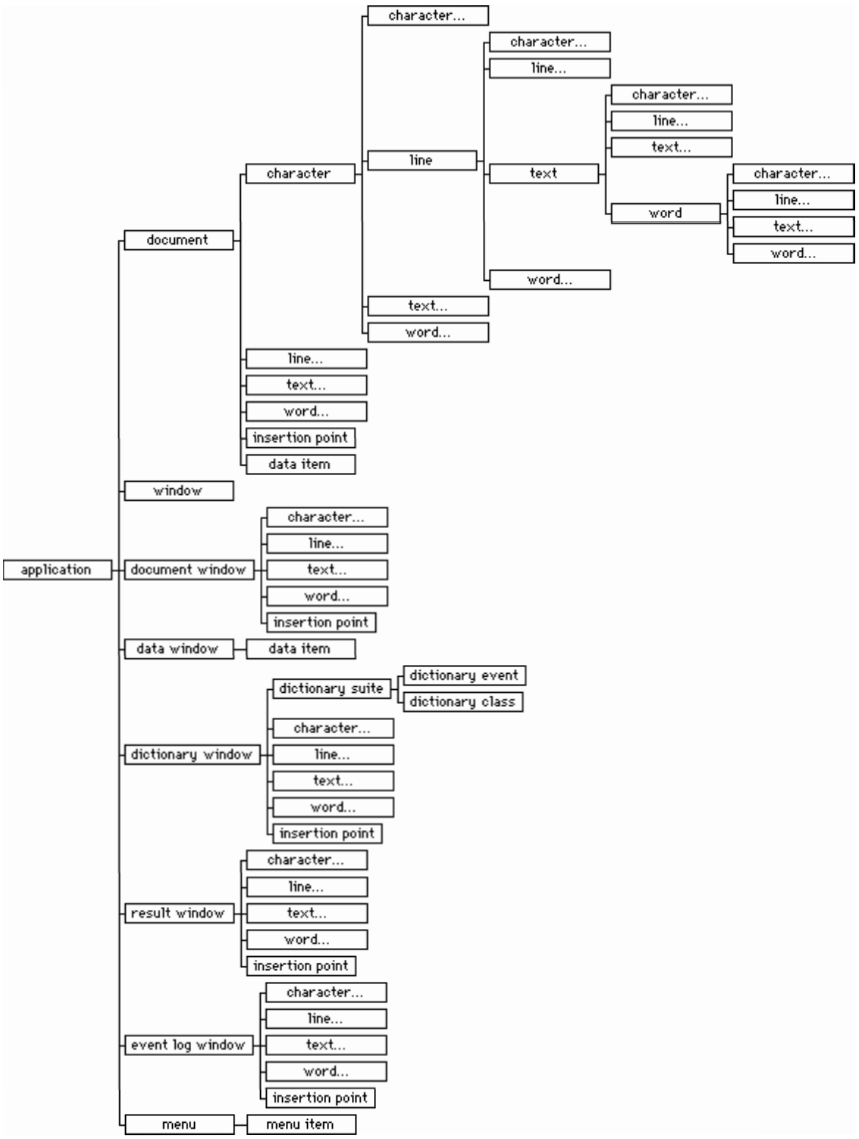
Another restriction in Script Debugger is that the event which initiates the execution of a script (the first step or execute command) will not complete until the script finishes executing. If you want to be able to issue additional step events, you must send the first step event using an AppleScript "ignoring application response" block.

In addition to these restrictions, you will also notice that object references containing "whose" tests involving strings are limited to 255 characters and not 32,767 as the Apple Event Registry suggests.

Element Hierarchy

Figure B-1 shows the hierarchy of objects within Script Debugger. Whenever you see an element in the hierarchy followed by an ellipse, the element has a sub-tree that is hidden to save space in the graphic. For more information about objects and inheritance in AppleScript, refer to the *Apple Event Object Support Library Developer Note* and the *Apple Event Registry*.

Figure B-1
Script Debugger's object hierarchy



Perhaps the best way to think of Script Debugger's element hierarchy is as windows and the information they contain. For example,

Script Debugger
  has open documents
    which have text
      which have data items

  has open dictionaries
    which have suites
      which contain events
      which contain classes

This is the core structure for the program. The window classes are built around this core and provide access to non-document information such as the dictionary, event log, and result windows. The window classes also provide alternate access paths to document information such as the script text and data items. The alternate access paths are made possible by additions to the Core, Text, and Miscellaneous suites. You can see these suites in the Script Debugger dictionary.

Augmented Suites

If you look in Script Debugger's Dictionary window, you will see the core, text, and miscellaneous suites represented along with the Script Debugger suite. These suites are included in the dictionary because they have additional commands, classes, and properties that make scripting Script Debugger easier. We will look at a few of these commands in this section, and you can find out more about the additions by examining the dictionary.

Text Suite

The additions to the Text suite allow you to get text from Script Debugger's document windows. These extensions to the suite are limited to some additional properties for the text, line, character, and word classes. For example, the properties "breakpoint" and "breakpoint set" let you get the breakpoint information about a line in your script. The extensions to the text suite are modeled on the Scriptable Text Editor.

In the Text suite, you can ask for the default value of an object in a few forms:

1. As text.

```
get lines of first document whose breakpoint set = true as plain text
```

2. As styled text. This is the default.

```
get first line of document as styled text
```

3. As an object reference.

```
get lines of first document whose breakpoint=true as reference
```

Object references are important because they allow you to retain references in your script. This can be very useful if you need to apply a number of events to the references:

```
set bpLines to lines of first document whose breakpoint set = true as reference
set bpLineText to contents of bpLines
set breakpoint set of bpLines to false
```

This is generally faster because the returned object references are character-based, which is the fastest method. Line-based selections are the next fastest. Word-based selections are the slowest because Script Debugger has to find all of the words along the way to resolve the specification.

Core Suite

The only addition to the Core suite is the Save command. It accepts some additional parameters which let you specify the file type and whether or not it is run-only. For instance, you can save a window into a debugger script.

```
save first window in (new file) as debugger script
```

Or you could save a compiled script in read only format.

```
save first window in (new file) as script application with run only
```

Miscellaneous Suite

The additions to the Miscellaneous suite allow you to script the menus and menu items in Script Debugger. As you will see in a script later in this appendix, you can use the menu commands to bring windows to the front in your scripts. You can also use menu commands to speed some otherwise tedious-to-script tasks.

For example, if you want your script to quit Script Debugger and save all of the windows as Script Debugger is quitting, you could write a script which cycles through all of the windows and saves each one. However, the menu commands allow you to accomplish the same task in fewer lines.

```
select menu item "Save All" of menu "Windows"
select menu item "Quit" of menu "File"
```

Granted, such scripts are not always optimal, but they can save you time and let you script objects in Script Debugger that you might not otherwise be able to script.

The Miscellaneous suite also contains an addition to the Select command. It has a "scroll to selection" parameter which only applies when selecting objects from the text suite. It allows you to scroll the newly selected text into view.

Script Debugger Suite

Aside from the additions to the other suites that we have just looked at, all of the other unique commands are limited to the Script Debugger suite. The commands in the Script Debugger suite allow you to perform all of the tasks in scripts that you can perform in the interface. This extends to the way that scripts are executed and that windows are opened, in scripts.

Script Window

When you are executing a script interactively in Script Debugger, the Script window must be on top. The same holds true of any script that you write to control Script Debugger. If your script issues events to Script Debugger which change window order, the window of the running script remains on top. If your script opens a Dictionary window, for example, it will be opened below the running script.

Note also that when a script is executing or paused, its text is locked. Any Apple events which try to alter the text of a running or paused script, are not handled.

With those restrictions in mind, let's look at some ways that you can script Script Debugger.

Create a Script Application

This example script creates a script application from the script on the clipboard. While trivial in itself, this script demonstrates that you can control Script Debugger through scripts.

Due to Script Debugger's restriction that the executing window must be on top, you would need to run this script from another script editor or a second copy of Script Debugger.

```
tell application "Script Debugger"
   activate
   paste
   save first window in file "Script App" as script application
end tell
```

Before you run this script, you would need to copy the script for a script application to your clipboard. When the example script is executed, activating Script Debugger converts the clipboard. The paste command inserts the contents of the clipboard into the front-most window. The save command allows you to save the script as an application. If you step through the script, you may notice a longer delay while the save line is executing. This is a result of Script Debugger compiling the script before it saves it as an application. Of course, any errors in the script that you are pasting in will cause the script to fail.

Save as Run-Only

This example script has its basis in the File Droplet template. It takes a compiled script that you drop on it and saves it as a run-only script. The script application opens the script dropped on it in Script Debugger and then saves it.

```
on ProcessAFile(aFile)
   tell application "Script Debugger"
      open aFile
      save first window in (new file) as script¬
      application  with run only
   end tell
end ProcessAFile

on open of fileList
   repeat with aFile in fileList
      ProcessAFile(contents of aFile)
   end repeat
end open

on run
   set theFile to choose file
   ProcessAFile(theFile)
end run
```

The script prompts you for a name, but it could just as easily be re-written to take the name of the original script.

Comment Lines

While the previous scripts show how easy it is to script Script Debugger, the real power of Script Debugger is in its extendability. Through Extension scripts, you can add features and power to Script Debugger. This example script places a block comment around the selected text and can be run as a script extension. It performs a couple of checks to make certain that the front window is a script and that it is not paused or running. It then gets the range of the selected text and places a block comment mark at the beginning and end of the selection.

```
if first document exists then
   if class of first window = document window then
      if debugger state of first document = stopped then
         set theSel to selection of first document
         if class of theSel ≠ insertion point then
            set startOffset to character offset of first character of theSel
            set endOffset to character offset of last character of theSel

            make new text at after character endOffset of first document ¬
                 with data "*)"
            make new text at before character startOffset of first document ¬
                 with data "(*"
            select text startOffset thru (endOffset + 4) of first document
         else
            error "Can't comment empty selection"
         end if
      else
         error "Script "" & (name of first document) & "" is running or recording."
      end if
   else
      error "The current window is not a document window"
   end if
else
   error "There are no open documents"
end if
```

You can extend this script to place either block comment marks or single line comment marks based upon the length of the selection. For example, if there are more than six lines selected, the script would place block comment marks. If six lines or less are selected, the script would place two dashes at the beginning of each line.

Scripting Pointers

When you are writing scripts for Script Debugger's Script window, it is a good idea to use "whose" expressions as much as possible. This speeds up script execution considerably. For instance, you could locate all of the lines containing breakpoints by stepping through each of the lines and checking to see if the breakpoint is set.

```
set bpLines to {}
repeat with i from 1 to count lines of first document
   if breakpoint set of line i of first document then
      set bpLines to bpLines & { i }
   end
end
```

While thorough, this is the slow way to find the breakpoints. A faster way uses a “whose” expression.

```
set bpLines to lines of first document whose breakpoint set = true
```

Whose expressions rely on object references. As noted earlier in this appendix, they are faster than line and word references.

Data Window

In addition to manipulating the contents of script windows and the windows themselves, you can also work with the contents and other Script Debugger windows. For example, you can save the contents of the Data window. If you save a script that you are working on as a Debugger script, it retains the expressions and their values between runs. You might also find it useful to save the contents of the Data window across several runs of your script for the purpose of comparison. The following script saves the contents of the Data window to a text file.

```
script dataItemObj
   property theExprs : { }
end

(*Save the expressions in the data window to a file *)

set theExprs of dataItemObj to (expression of data items of second document)
set theObjFile to new file
store script dataItemObj in theObjFile replacing yes

(*Add the expressions saved in a file to the data window *)

delete data items of second document

set newDataItemObj to load script theObjFile
repeat with anExpr in theExprs of newDataItemObj
   make new data item at end of second document with data (contents of anExpr)
end repeat
```

Dictionary Window

The dictionary commands in the Script Debugger suite allow you to open any Dictionary window or the Scripting Additions Dictionary window. The scripting interface also allows you to retrieve information from the Dictionary window. If you have selected text in a Dictionary window, you can retrieve the text with the script

```
get text from the first dictionary window
```

You can also specify the text for Script Debugger to retrieve. For example, the script

```
get lines 1 through 5 of the first dictionary window
```

returns the specified lines. The full text suite is supported so you can perform "whose" tests or any operation you can do in the other windows.

NOTE: *The text in Dictionary windows is not modifiable.*

For a better idea of the capabilities in Script Debugger's implementation of the dictionary window, open the Additions Inventory script application that you will find in the Script Debugger Examples folder of the Examples folder. This script uses Script Debugger to open the dictionary window for each addition in the Scripting Additions folder, to copy the class and event information, and to write it to a Scriptable Text Editor file. The business end of the script begins after the "tell application "Script Debugger"" statement which is located toward the end of the file.

```
tell application "Script Debugger"
   set theDict to open dictionary additionPath
        tell me to OutputText("Addition "" & name of additionInfo & """)
        set numSuites to count every dictionary suite of theDict
        repeat with i from 1 to numSuites
          tell me to OutputText("  Suite: " & name of ¬
             dictionary suite i of theDict)
          if dictionary event 1 of dictionary suite i of theDict exists then
          tell me to OutputText("  Events: " & ¬
                  (name of every dictionary event of ¬
                    dictionary suite i of theDict) as string)
        else
          tell me to OutputText("  Events: none")
          end if
          if dictionary class 1 of dictionary suite i of theDict exists then
          tell me to OutputText("  Classes: " & ¬
                  (name of every dictionary class of ¬
                    dictionary suite i of theDict) as string)
        else
          tell me to OutputText("  Classes: none")
          end if
        end repeat
     tell me to OutputText(" ")
     close theDict
   end tell
```

In this portion of the script, Script Debugger counts the suites in the scripting additions window. It then checks for and lists the events and classes in each of the suites in the Dictionary window. Notice that it is retrieving the information as a string and then concatenating the string with a label to paste in the Scriptable Text Editor window.

Event and Result Windows

The Script Debugger suite contains classes that allow you to refer to the Event and Result windows. You can use the save command to write the contents of the window to a file.

While you are testing a script, you might want to save the contents of the Event Log or Result windows during different runs of a script. This would provide you with a source of comparison as you make changes to your script and help you isolate problems. It is easy enough to bring the Event Log or Result window to the front and perform a save, but it is even simpler to incorporate the save command into your script.

For example, you can add a line to the end of your script that prompts you for a name and saves the contents of the Result window to disk.

```
tell application "Script Debugger"
   list folder "Hard Disk 245:System Folder:Extensions:Scripting Additions:"
   save "Result" window (new file with default¬
      "Additions Listing")
end tell
```

You can script the Event Log window in the same way. You should remember that the events are not recorded in the window if it is not open. To make sure that the window is open to catch events, you can add a line to the beginning of your script that opens the window, and then one at the end that saves the contents of the window.

```
tell application "Script Debugger"
   select menu item "Show Event Log" of menu "Windows"
   list folder (choose folder)
   save event log window in (new file)
end tell
```

Note that this rather simplistic script will fail if the Event Log window is already open since its menu will be disabled. You can check for the status of a menu with this line of script.

```
get enabled of menu item "Enter Selection" of menu "Search"
```

This will return a boolean on which you can base the behavior of your script .

APPENDIX C

Scripting Additions

This appendix describes each of the AppleScript commands in the Late Night Software Scripting Additions package. Scripting additions are a special type of software that add new features to the AppleScript language.

The Late Night Software Scripting Additions are automatically installed when you select Easy Install when installing Script Debugger (*see Chapter 1*). You will find the example scripts in the Examples folder which is inside the Examples folder of the Script Debugger folder.

Additions Examples

The Additions Examples folder inside the Examples folder contains a series of example AppleScript scripts showing how to use the new commands provided by the Late Night Software Scripting Additions. All of these examples are stored as Script Debugger scripts or script applications.

AppleTalk Folder

AppleTalk State

This example displays the current state of the AppleTalk network.

List File Servers

This example lists the file servers available on your AppleTalk network.

Files & Folders Folder

Choose File In Prefs Folder

This example shows how to use the Set Default Folder and Get Default Folder commands to control the starting folder presented by the Choose File command.

Choose New File Example

This short example shows the Choose New File command in use.

Choose Several Files Example

This short example shows the Choose Several Files command in use.

Choose Several Folders Example

This short example shows the Choose Several Folders command in use.

File IO Example

This example creates a text file and writes a short message to it using the File IO commands.

File IO Example II

This example opens a text file and displays the contents of the file, line by line.

List Folders

This example uses the Choose Several Folders and the File IO commands to produce a listing of the files stored in folders.

Misc. Folder

Quit All Applications

This example illustrates how to use the List Processes, Get Process and Get Current Process commands to quit all the non-essential applications running on your Macintosh.

Shutdown

This example illustrates the use of the Shutdown command.

Regular Expressions Folder

Regular Expression Example

This example uses the Regular Expression commands to modify the names of all the files in a folder (note the file names are not actually changed).

Regular Expression Example II

This example uses Regular Expression and File IO commands to read and parse a simple text file.

Resources

Count Resource

This example counts the number of resources in a resource file.

Enumerate All Resource Types

This example displays all of the resource types in the current resource file chain.

Get Chosen Printer

This example displays the name of the currently selected printer.

Remove "ckid" Resources

This droplet script scans a directory tree and removes 'ckid' resources from the files it finds in the directory tree. 'ckid' resources are added to files by Projector/SourceServer.

Speech Folder

Happy Birthday To You
: This example script application sings the happy birthday song using the various voices installed in your system.

Speak A Text File
: This example script application speaks the contents of a file, line by line.

Libraries

The Libraries folder contains the following AppleScript libraries.

Gestalt Selector Lib
: This library defines the Gestalt selectors which are documented in *Inside Macintosh, Volume VI.*

Error/Return Codes Lib
: This library defines a variety of Macintosh OS error and return codes.

AppleScript Error Codes Lib
: This library defines AppleScript and AppleEvent related error codes.

Script Tools 1.3

The Late Night Software Scripting Additions are based on the popular and award winning Script Tools 1.3 package. If you have been using Script Tools 1.3, you may notice that some of the scripting additions have changed and some new ones have been added. These new additions include AppleTalk Control, More Math, List Manipulation, and Resource IO.

You will also notice that the Choose Folder command has been removed because it conflicts with Apple's Choose Folder command. The handler is still in the Choose Files & Folders scripting addition to provide backward compatiblity with older scripts, but the dictionary entry has been removed.

AppleScript 1.1 Issues

With the release of AppleScript 1.1, Apple has introduced a number of new scripting additions which make some Late Night Software commands obsolete. However, you may continue to use the commands without fear of conflicting with Apple's new commands.

Now that the Scriptable Finder has been released by Apple, you may find the functionality provided by some commands is also available through the Finder. Here too, you may continue to use the scripting additions without conflicting with the new Finder.

AppleTalk Control

The AppleTalk Control addition allows your scripts to perform AppleTalk network management operations. This section describes each of the AppleScript commands provided in the AppleTalk Control addition.

Get Zone

The Get Zone command allows you to determine the name of the AppleTalk zone to which your Macintosh belongs. If your AppleTalk network has only one zone, the command returns the string "*".

Syntax

```
get zone
```

Parameters

None.

Result

None.

Example

```
set myZone to get zone
if myZone = "*" then
   display dialog "No zones" buttons { "OK" }
else
   display dialog "Zone: " & myZone buttons { "OK" }
end if
```

Outcome:

```
tell current application
    get zone
—> "ARA"
    display dialog "Zone: ARA" buttons {"OK"}
—> {button returned:"OK"}
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox GetMyZone routine. These errors are explained in detail in the *Inside Mac* series.

List Zones

The List Zones command allows you to find the names of all the zones on your local network.

Syntax

```
list zones
```

Parameters

None.

Result

None.

Example

```
set localZoneNames to list zones
display dialog "Zones: " & count of localZoneNames
```

Outcome:

```
tell current application
    list zones
—> {"Engineering2", "Marketing", "Marketing2", "RandD",
"Admin", "Engineering"}
    display dialog "Zones: 6"
—> {button returned:"OK"}
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox GetLocalZones routine. These errors are explained in detail in the *Inside Mac* series.

List Network Names

The List Network Names command allows you to search for services available on an AppleTalk network.

Each Macintosh registers a series of names representing the various network services available on the machine. Other systems query these registered names to find the network addresses.

Syntax

```
list network names
  [ object objectName ]
  [ type typeName ]
  [ zone zoneName ]
```

Parameters

objectName This optional parameter specifies the objects to be found. If the parameter is not supplied, all objects are found.

Wildcard characters (=) may be used to replace portions of the object name. For example, "=Server" matches any object name ending in "Server".

typeName This optional parameter specifies the types of objects to be found. If the parameter is not supplied, all object types are found.

Wildcard characters (=) may be used to replace portions of the type name. For example, "=Server" matches any object types ending in "Server".

zoneName This optional parameter specifies the zone to search. If the parameter is omitted, the current zone is searched. No wildcard characters are allowed.

Result

The result of the list network names command is a list of Network Name classes. Network Name classes contain the following fields:

atObject The name of an AppleTalk network object

atType The type name of an AppleTalk network object

atZone The zone name of an AppleTalk network object

Example

```
—
—  Find all the AppleShare servers
—
list network names type “AFPServer”
```

Outcome:

```
tell current application
    list network names type “AFPServer”
—> {{class:network name, atObject:”pb1", atType:”AFPServer”,
atZone:”*”}, {class:network name, atObject:”pf’s PowerBook
170", atType:”AFPServer”, atZone:”*”}}
end tell
```

Get Network State

The Get Network State command allows your script to determine if the network is active.

Syntax

```
get network state
```

Parameters

```
None.
```

Result

```
None.
```

Example

```
get network state
```

Outcome:

```
tell current application
    get network state
—> network active
end tell
```

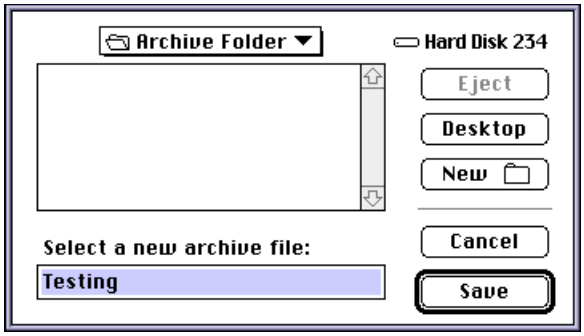
Choose Files and Folders Addition

The Choose Files & Folders addition allows you to prompt the user to select files and folders in your scripts. This addition builds on the file prompting capabilities provided by Apple in the AppleScript software. This section describes each of the AppleScript commands provided in the Choose Files & Folders addition.

Choose New File

The Choose New File command presents the standard Macintosh new file selection dialog box *(Figure C-2)*.

Figure C-2
The Choose New File dialog box



Syntax

```
choose new file
   [ with prompt promptString ]
   [ default name nameString ]
```

Parameters

promptString

This parameter is a string which is displayed in the dialog box. If this parameter is omitted, the string "Save As:" is displayed.

nameString

This parameter is a string which is offered as the default name for the new file. If this parameter is omitted, no default name is presented.

Result

The result of the Choose New File is a record containing three values:

filename returned
: This value is a string representing the name of the new file.

folder returned
: This value is an alias to the folder where the new file is to be placed.

replacing
: This Boolean value indicates whether or not the new file replaces an existing file (TRUE = Yes, FALSE = No).

Example

```
— Ask the user for a new file
set newFile to choose new file ¬
    with prompt "Select a new archive file:" ¬
    default name "Testing"
```

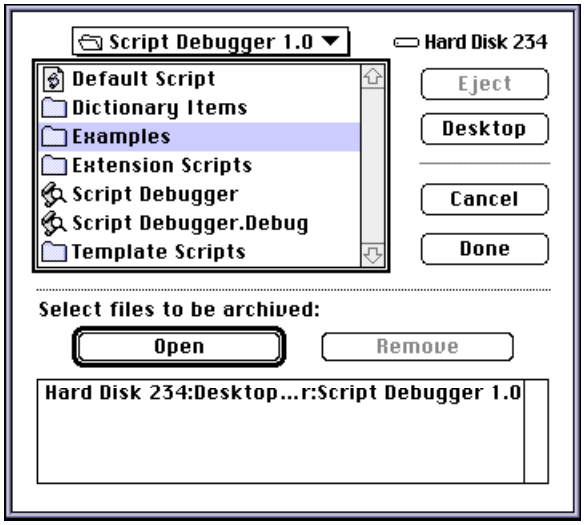
Outcome:

```
—> {resource name:"Testing", replacing:false, folder
returned:alias "Hard Disk 234:Archive Folder:"}
```

Choose Several Files

The Choose Several Files command presents a modified standard file selection dialog box which allows the user to choose several files at one time *(Figure C-3)*.

Figure C-3
Choose Several Files dialog box



Syntax

```
choose several files
  [ with prompt promptString ]
  [ of type typeList ]
  [ starting with fileList ]
```

Parameters

promptString

This parameter is a string which is displayed in the dialog box. If you omit the `with prompt` parameter, no prompt is displayed.

typeList This parameter is a list of strings specifying the file types of the files to be displayed in the dialog box. Each string is a four-character code for the file type, such as `"TEXT"`, `"APPL"`, `"PICT"` or `"PNTG"`. If you omit the `of type` parameter, all files are displayed. You may specify up to four file types.

fileList This parameter is a list of aliases referring to files which are to be displayed as already selected. If you omit the `starting with` parameter, the selected files list is left empty.

Result

The result is a list of aliases referring to the files selected by the user.

Example

```
choose several files ¬
    with prompt "Select files to be archived:" ¬
    of type {"APPL", "asDF" } ¬
    starting with¬
   { alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:"
}
```

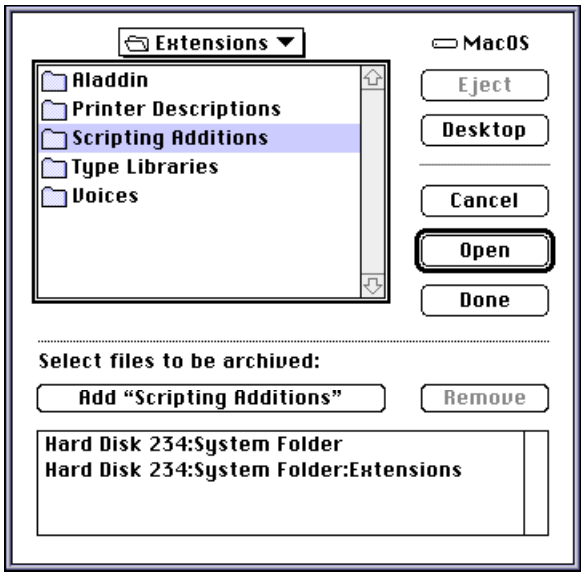
Outcome:

```
—> {alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:"}
```

Choose Several Folders

The Choose Several Folders command presents a modified standard file selection dialog box allowing the user to choose several folders at one time *(Figure C-4)*.

Figure C-4
Choose Several Folders dialog box



Syntax

```
choose several folders
   [ with prompt promptString ]
   [ starting with folderList ]
```

Parameters

promptString

This parameter is a string which is displayed in the dialog box. If you omit the `with prompt` parameter, no prompt is displayed.

folderList This parameter is a list of aliases referring to folders which are to be displayed as already selected. If you omit the `starting with` parameter, the selected folders list is left empty.

Result

The result is a list of aliases referring to the folders selected by the user.

Example

```
choose several folder ¬
    with prompt "Select files to be archived:" ¬
    starting with ¬
        { alias "Hard Disk 234:System Folder:" ¬
          alias "Hard Disk 234:System Folder:Extensions:"}
```

Outcome:

```
—> {alias "Hard Disk 234:System Folder:", alias "Hard Disk
234:System Folder:Extensions:"}
```

Get Default Folder

The Get Default Folder command returns the current folder used by the Choose File and Choose Folder commands in this package and those provided by Apple as part of AppleScript.

Syntax

```
get current folder
```

Result

This command returns an alias to the current default folder.

Example

```
set saveFolder to get default folder
set default folder path to extensions
choose file
set default folder saveFolder
```

Outcome:

```
tell current application
    get default folder
—> alias "MacOS:System Folder:Extensions:"
    path to preferences folder
—> alias "MacOS:System Folder:Extensions:"
    set default folder alias "MacOS:System
Folder:Preferences:"
    choose file
—> alias "MacOS:System Folder:Extensions:AppleScript™"
    set default folder alias "MacOS:System Folder:Extensions:"
end tell
```

NOTE: *If you have System 7.5 installed, the General Controls control panel has a setting that allows you to set the folder to which the Open and Save As commands default. The Choose commands in the example above do not override the effect of the control panel.*

Set Default Folder

The Set Default Folder command changes the current folder used by the Choose File and Choose Folder commands in this package and those provided by Apple as part of AppleScript.

Syntax

```
set default folder folderPath
```

Result

This command returns no result.

Parameters

folderPath This parameter is an alias to the folder which is to become the default folder. If you provide an alias to a file, the folder containing the file becomes the default folder.

Example

```
set default folder path to extensions
choose file
```

Outcome:

```
tell current application
    path to extensions folder
—> alias "MacOS:System Folder:Extensions:"
    set default folder alias "MacOS:System Folder:Extensions:"
    choose file
—> alias "MacOS:System Folder:Extensions:AppleScript™"
end tell
```

NOTE: *Under System 7.5, the Set Default Folder command does not have any effect.*

File IO Addition

The File IO addition allows your scripts to perform file operations directly within your scripts. Using the commands of the File IO addition, your scripts can move, rename, and delete files. Your scripts can also read and write text data. This section describes each of the AppleScript commands provided by the File IO addition.

CloseFile

The closeFile command closes a file previously opened with the openFile command.

Syntax

```
closeFile fileRefNum
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the Open File command.

Result

None.

Example

```
set filePath to choose file ¬
   with prompt "Select a file to open:" ¬
   of type "TEXT"
set refNum to openFile filePath for reading
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" for
reading
—> 8744
   closeFile 8744
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox FSClose routine.

CreateFile

The createFile command creates a new file of type "TEXT".

Syntax

```
createFile fileName
   [ in folder ]
   [ owner signature ]
```

Parameters

fileName This parameter is the new file's name.

folder This parameter is an alias to the folder where the new file is to be placed. If this parameter is omitted, the file is created in the current default folder.

signature This parameter is a list of aliases referring to folders which are to be displayed as already selected. If you omit the `owner` parameter, the new file is given the signature `'????'`.

Result

None.

Example

```
set newFile to choose new file ¬
   with prompt "Pick a new file name:"

createFile (filename returned of newFile) ¬
   in (folder returned of newFile) ¬
   owner "ttxt" — TeachText
```

Outcome:

```
tell current application
   choose new file with prompt "Pick a new file name:"
—> {resource name:"a text file", replacing:false, folder returned:alias "Hard Disk
234:Desktop Folder:Script Debugger 1.0:"}
   createFile "a text file" in alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:" owner "ttxt"
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox HCreate routine.

CreateFolder

The createFolder command creates a new folder.

Syntax

```
createFolder folderName
   [ in folder ]
```

Parameters

folderName

This parameter is the new folder's name.

folder This parameter is an alias to a folder where the new folder is to be placed. If this parameter is omitted, the file is created in the current default folder.

Result

None.

Example

```
set newFolder to choose new file ¬
   with prompt "Pick a new folder name:"

createFolder (filename returned of newFolder) ¬
   in (folder returned of newFolder)
```

Outcome:

```
tell current application
   choose new file with prompt "Pick a new folder name:"
—> {resource name:"Empty Folder", replacing:false, folder returned:alias "Hard
Disk 234:Desktop Folder:Script Debugger 1.0:"}
   createFolder "Empty Folder" in alias "Hard Disk 234:Desktop Folder:Script
Debugger 1.0:"
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox DirCreate routine.

DeleteFile

The deleteFile command deletes a file without placing it in the Trash.

Syntax

```
deleteFile theFile
```

Parameters

theFile

This parameter is an alias referring to the file being deleted.

Result

The result is a list of aliases referring to the folders selected by the user.

Example

```
choose file¬
    with prompt "Select a file to be deleted:" ¬
set trashMe to the result
deleteFile trashMe
```

Outcome:

```
tell current application
   choose several files with prompt "Select files to be deleted:"
—> {alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:a text file"}
   deleteFile {alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:a text
file"}
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox HDelete routine.

ExchangeFile

The exchangeFile command swaps the data stored in two files.

Syntax

```
exchangeFile firstFile with secondFile
```

Parameters

firstFile This parameter is an alias which identifies the first of the two files.

secondFile This parameter is an alias which identifies the second of the two files.

NOTE: *The two files being exchanged must be on the same disk volume.*

Result

None.

Example

```
exchangeFile (choose file) with (choose file)
```

Outcome:

```
tell current application
   choose file
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Backup"
   choose file
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   exchangeFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Backup"
with alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox PBExchangeFiles routine.

GetFileLength

The getFileLength command gets the length (in bytes) of the file.

Syntax

```
getFileLength fileRefNum
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the Open File command.

Result

The number of bytes stored in the file.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "TEXT"
set refNum to openFile filePath for reading
— position the marker at the end of the file so
— data can be appended to the file
getFileLength refNum
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" for
reading
—> 8838
   getFileLength 8838
—> 2837
   closeFile 8838
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox GetEOF routine.

GetFilePosition

The getFilePosition command obtains the current position of a file's marker. A file marker represents the address within a file where the next read or write will begin.

Syntax

```
getFilePosition fileRefNum
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the Open File command.

Result

The result is a number representing the address of the file's marker.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "TEXT"
set refNum to openFile filePath for reading
getFileLength refNum
positionFile refNum at (getFileLength refNum)
getFilePosition refNum
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" for
reading
—> 7804
   getFileLength 7804
—> 2837
   positionFile 7804 at 2837
—> 2837
   getFilePosition 7804
—> 2837
   closeFile 7804
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox GetFPos routine.

LengthenFile

You can use the lengthenFile command to shorten or extend the size of a file. Note that you must open the file for writing to lengthen or shorten a file.

Syntax

```
lengthenFile fileRefNum length fileLength
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the Open File command.

fileLength This parameter is the new length of the file.

Result

None.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "TEXT"
set refNum to openFile filePath for writing
— empty the contents of a file
lengthen file refNum length 0
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" for
writing
—> 8838
   lengthenFile 8838 length 0
   closeFile 8838
end tell
```

MoveFile

The moveFile command moves a file or a folder from one folder to another.

Syntax

```
moveFile fileOrFolder to destination
```

Parameters

fileOrFolder

This parameter is an alias which identifies the file or folder being moved.

destination

This parameter is an alias referring to the destination folder for *fileOrFolder*.

NOTE: *The file or folder being moved must be on the same disk volume as the destination.*

Result

None.

NOTE: *The file or folder being moved and the destination folder must be on the same volume.*

Example

```
moveFile ( choose file ) to ( choose folder )
```

Outcome:

```
tell current application
   choose file
—> alias “Hard Disk 234:Desktop Folder:Script Debugger 1.0:Backup”
   choose folder
—> alias “Hard Disk 234:Desktop Folder:Script Debugger 1.0:Empty Folder:”
   moveFile alias “Hard Disk 234:Desktop Folder:Script Debugger 1.0:Backup” to
alias “Hard Disk 234:Desktop Folder:Script Debugger 1.0:Empty Folder:”
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox PBCatMove routine.

OpenFile

The openFile command opens a text file for reading and/or writing. This command, when used with the readLine and writeLine commands, allows you to process text files within scripts without the aid of a scriptable text editor application.

Syntax

```
openFile file
  [ for reading|update|writing ]
```

Parameters

file This parameter is an alias to the file which is to be opened.

Result

The result is a file reference number. You must provide this number to all other commands that you issue when processing the file.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "TEXT"
set refNum to openFile filePath for reading
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Script Debugger Manual:SD error messagees"
   openFile alias "Hard Disk 234:Script Debugger Manual:SD error messagees" for
reading
—> 9026
   closeFile 9026
end tell
```

NOTE: *When the optional* `for` *is not specified, the file is opened for update.*

NOTE: *Be careful to ensure you close all the files you open. Due to the nature of AppleScript additions, the Open File command does not ensure the file is closed when a script aborts without first closing the file with the Close File command.*

Errors

This command can return any of the errors which are returned by the ToolBox HOpen routine.

PositionFile

The positionFile command changes the current position of a file's marker. A file marker represents the address within a file where the next read or write will begin.

Syntax

```
positionFile fileRefNum at filePosition
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the Open File command.

filePosition

This parameter is the new address for the file's marker.

Result

None.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "TEXT"
set refNum to openFile filePath for reading
— position the marker at the end of the file so
— data can be appended to the file
positionFile refNum at (getFileLength refNum)
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" for
reading
—> 8838
   getFileLength 8838
—> 2837
   positionFile 8838 at 2837
—> 2837
   closeFile 8838
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox SetFPos routine.

ReadLine

The readLine command reads a "line" of text from a file opened with the openFile command. A line in this case means all characters up to the next carriage return in the file. This is referred to as a paragraph in some applications since these lines may wrap around a number of times when displayed in a window.

Syntax

```
readLine fileRefNum
   [maximum length maxLength]
```

Parameters

fileRefNum This parameter is the reference number of a file. This value is returned by the openFile command.

maxLength This integer parameter specifies the maximum number of characters you wish to read. Normally, the readFile command reads a maximum of 1024 characters. The practical maximum for this value is limited only by the memory available.

Result

The command returns a string representing the data from the file.

Example

```
set myFile to choose file ¬
   with prompt "Select a text file:" ¬
   of type "TEXT"
set refNum to openFile myFile
set inputLine to readLine refNum
display dialog inputLine
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a text file:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
—> 9308
   readLine 9308
—> "tell application \"Finder\""
   display dialog "tell application \"Finder\""
—> {button returned:"OK"}
   closeFile 9308
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox PBRead routine.

RenameFile

The renameFile command changes a file's name.

Syntax

```
renameFile file to newName
```

Parameters

file This parameter is an alias which identifies the file whose name is being changed.

newName This parameter is a text string containing the file's new name.

Result

None.

Example

```
renameFile ( choose file ) to "Backup"
```

Outcome:

```
tell current application
   choose file
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   renameFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log" to
"Backup"
end tell
```

Errors

This command can return any of the errors which are produced by the ToolBox PBHRename routine.

WriteLine

The writeLine command writes a line to a text file.

Syntax

```
writeLine fileRefNum text data
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the openFile command.

data This parameter is the line of text to be written to the file.

Result

None.

Example

```
set myFile to ¬
   choose file with prompt "Select a text file:" of type
"TEXT"
set refNum to openFile myFile
writeLine refNum text "hello"
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a text file:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
—> 9308
   writeLine 9308 text "hello"
   closeFile 9308
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox FSWrite routine.

WriteString

The writeString command writes a string of text to a file. This command differs from writeLine in that it does not add a new-line character to the end of the text you write.

Syntax

```
writeString fileRefNum text data
```

Parameters

fileRefNum

This parameter is the reference number of a file. This value is returned by the openFile command.

data This parameter is the line of text to be written to the file.

Result2

None.

Example

```
set myFile to ¬
   choose file with prompt "Select a text file:" of type "TEXT"
set refNum to openFile myFile
writeString refNum text "hello"
closeFile refNum
```

Outcome:

```
tell current application
   choose file with prompt "Select a text file:" of type "TEXT"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
   openFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:Event Log"
—> 9308
   writeString 9308 text "hello"
   closeFile 9308
end tell
```

Errors

This command can return any of the errors which are returned by the ToolBox FSWrite routine.

Find Application

The Find Application addition allows you to determine the application which owns a file.

findApplication

The findApplication command allows you to find the application which owns a file.

Syntax

```
findApplication signature on volumeName
  [ index appIndex ]
```

Parameters

signature This parameter is a four-character string representing an application signature. The findApplication command locates the application with this signature.

volumeName
This parameter specifies the name of the disk volume to search.

appIndex This optional parameter specifies which version of the application to find. If you omit this parameter, the findApplication command returns the version of the application with the most recent creation date.

This parameter is only useful if you keep multiple copies of application files on your disk.

Result

The result of the findApplication command is an application file corresponding to the signature specified.

Example

```
find application "asDB" on "Hard Disk 245:"
```

Outcome:

```
tell current application
    find application "asDB" on "Hard Disk 245:"
—> file "Hard Disk 245:Script Debugger 1.0:Script Debugger"
end tell
```

Gestalt Addition

The Gestalt addition allows you to determine the services available on the Macintosh running your scripts. This section describes each of the Gestalt commands provided by the Get Gestalt addition.

Get Gestalt

The Get Gestalt command gets information about the operating environment.

Syntax

```
get gestalt selector ]
  [ bit bitNumber ]
  [ with/without report missing selectors ]
```

Parameters

selector This parameter is a string representing the type of operating environment information you want. This parameter must be a four-character code. The gestalt Selectors Lib file defines all of the Gestalt selectors documented in *Inside Macintosh*, volume VI, as well as selectors for Apple's Speech Manager.

bitNumber

This optional parameter defines which bit of the selectors value to test. If this parameter is specified, the command returns a Boolean value. If the parameter is omitted, the command returns the entire selector value.

Result

The result of this command is either the selector's integer value when the bit parameter is not specified or a Boolean value when the bit parameter is specified.

NOTE: *When the optional with* `report missing selectors` *is specified, the Get Gestalt command reports errors associated with unknown selectors. Otherwise, a value of 0 is returned.*

Example

```
— verify that the Speech Mgr is present

property gestaltSpeechAttr : "ttsc"
property gestaltSpeechMgrPresent : 0

if get gestalt gestaltSpeechAttr ¬
   bit gestaltSpeechMgrPresent then
    display dialog "Speech Mgr Present"
else
    display dialog "Speech Mgr Missing"
end if
```

Outcome:

```
tell current application
    get gestalt "ttsc" bit 0
—> true
    display dialog "Speech Mgr Present"
—> {button returned:"OK"}
end tell
```

List Manipulation

The List Manipulation scripting addition has commands that let you manipulate lists quickly. Its commands allow you to find the difference, intersection or union of two lists.

Difference of

The Difference of command compares two lists and produces a list which contains the differences of the two lists. In other words, it produces a list of items which the two lists do not have in common.

Syntax

```
difference of list1 and list2
```

Parameters

list1 A list of items.

list2 A second list that you compare to the first list.

Result

The result of this command is a list containing the items that are not common to both lists.

Example

```
set theDifference to difference ¬
   of {1, 5, 10, "this is a test", 4} ¬
   and {4, 2, "hi there", "this is a test", 1}
```

Outcome:

```
—> {"hi there", 2, 5, 10}
```

Intersection of

The Intersection of command compares two lists and produces a list which contains the intersection of the two lists. In other words, it produces a list of items which both lists have in common.

Syntax

```
intersection of list1 and list2
```

Parameters

list1 A list of items.

list2 A second list that you compare to the first list.

Result

The result of this command is a list containing the items common to both lists.

Example

```
set theIntersection to intersection¬
   of {1, 5, 10, "this is a test", 4} ¬
   and {4, 2, "hi there", "this is a test", 1}
```

Outcome:

```
—> {"this is a test", 1, 4}
```

Union of

The Union of command compares two lists and produces a list which contains all of the unique items of the two lists. This command is useful when you are trying to merge the contents of two lists quickly. It saves you the trouble of comparing each item in the two separate lists and then creating a third, unique list.

Syntax

```
union of list1 and list2
```

Parameters

list1 A list of items.

list2 A second list that you compare to the first list.

Result

The result of this command is a list containing all of the unique items in the two lists. It is a merged list.

Example

```
set theUnion to union¬
   of {1, 5, 10, "this is a test", 4}¬
   and {4, 2, "hi there", "this is a test", 1}
```

Outcome:

```
—> {"hi there", "this is a test", 1, 2, 4, 5, 10}
```

More Math Addition

The More Math and More Math 68881 additions allow your scripts to perform mathematical operations. This section describes each of the AppleScript commands provided by the More Math and More Math 68881addition. The Script Debugger installer installs either the More Math or More Math 68881. If your Macintosh has a floating point accelerator, More Math 68881 is installed.

abs

The abs command returns the absolute value of an integer.

Syntax:

```
abs intValue
```

acos

The acos command computes the arc cosine of a value.

Syntax:

```
acos value
```

asin

The asin command computes the arc sine of a value.

Syntax:

```
asin value
```

atan

The atan command computes the arc tangent of a value.

Syntax:

```
atan value
```

atan2

The atan2 command computes the arc tangent of the quotient of two values.

Syntax:

```
atan2 value1 over value2
```

cos

The cos command computes the cosine of a value.

Syntax:

```
cos value
```

cosh

The cosh command computes the hyperbolic cosine of a value.

Syntax:

```
cosh value
```

fabs — The fabs command calculates the absolute value of a floating point value.

Syntax:
`fabs` *value*

log — The log command computes the natural logarithm of a value.

Syntax:
`log` *value*

log10 — The log10 command computes the base-10 logarithm of a value.

Syntax:
`log10` *value*

power — The power command raises a value to a power.

Syntax:
`power` *value* `to` *powe*

sin — The sin command computes the sign of a value.

Syntax:
`sin` *value*

sinh — The sinh command computes the hyperbolic sign of a value.

Syntax:
`sinh` *value*

tan — The tan command computes the tangent of a value.

Syntax:
`tan` *value*

tanh — The tanh command computes the hyperbolic tangent of a value.

Syntax:
`tanh` *value*

Processes

The Processes scripting addition lets you get detailed information about the applications running on your Macintosh. It has commands to get the foreground application and currently executing application as well as any desk accessories and faceless background-only applications.

List Processes

The List Processes command obtains a list of the names of the applications running on your Macintosh. This includes normal Macintosh applications, desk accessories, and faceless-background-only applications.

Syntax

```
list processes
```

Parameters

None.

Result

The result is a list of strings representing the names of all the running applications.

Example

```
list processes
```

Outcome:

```
—> {“PowerTalk Manager”, “Finder”, “QuicKeys™ Toolbox”, “Script Debugger.Debug”,
“Microsoft Word”, “Canvas™ 3.5”, “Script Debugger”}
```

Get Process

The Get Process command obtains detailed information about a running application.

Syntax

```
get process processName
```

Parameters

processName

This parameter specifies the name of the process you want information about.

Result

The result of the Get Process command is a record containing the following values:

process name

This string is the name of the process.

process number

This value represents the serial number of the process. AppleScript translates this value into an application object automatically.

application type

This string is the application's four-character file type. Normally this value is "APPL".

signature

This string is the application's four-character signature.

partition size

This integer value represents the amount of memory the application occupies.

free memory

This integer value represents the amount of free memory within the application's partition.

launcher

This string is the name of the application which launched this application. If it's blank, then the application is no longer running.

launch date

This integer value represents the date and time when the application was launched.

active time

This integer value is the amount of CPU time used by the application since it was launched. The units for this value are ticks (1/60th of a second).

application file

This value is a reference to the application's file.

deskAccessory
multiLaunch
needSuspendResume
canBackground
activateOnForegroundSwitch
compatible32Bit
onlyBackground
getFrontClicks
getApplicationDiedEvents
highLevelEventAware
localAndRemoteEvents
stationeryAware
useTextEditServices

These Boolean values represent the application's mode flags.

Example

```
get process (first item of (list processes))
```

Outcome:

```
tell current application
   list processes current application
—> {“Finder”, “QuicKeys™ Toolbox”, “Script Debugger.Debug”, “Microsoft Word”,
“Canvas™ 3.5”, “Script Debugger”}
   get process “Finder”
—> {class:process info, resName:”Finder”, process number:application “Finder”,
application type:”FNDR”, signature:”MACS”, partition size:377856, free
memory:204448, launch date:date “Saturday, March 4, 1995 2:24:02 PM”, active
time:3203261, application file:file “MacOS:System Folder:Finder”,
deskDccessory:false, multiLaunch:false, needSuspendResume:true, canBackground:true,
activateOnForegroundSwitch:true, onlyBackground:false, getFrontClicks:true,
getApplicationDiedEvents:true, compatible32Bit:true, highLevelEventAware:true,
localAndRemoteEvents:true, stationeryAware:false, useTextEditServices:false,
launcher:””}
end tell
```

Get Foreground Process

The Get Foreground Process command gets the name of the foreground application. The foreground application is the application whose windows are presently active. Note that the foreground application is not necessarily the current application (see the Get Current Application command).

Syntax

```
get foreground process
```

Parameters

None.

Result

The result of this command is a string representing the name of the foreground application.

Example

```
get foreground process
```

Outcome:

```
—> "Script Debugger"
```

Get Current Process

The Get Current Process command gets the name of the currently executing application. This command is useful for finding the name of the process executing a script. The value returned by the Get Current Process command is different from the value returned by the Get Foreground Process command when the current process is in the background.

Syntax

```
get current process
```

Parameters

None.

Result

The result of this command is a string representing the name of the currently executing application.

Example

```
get current process
```

Outcome:

```
–> "Script Debugger"
```

Regular Expressions Addition

The Regular Expressions addition allows you to perform simple and complex textual pattern matching within your scripts. This section describes each of the AppleScript commands provided in the Regular Expressions addition.

Compile Regular Expression

The Compile Regular Expression command compiles a pattern string. Compiled Regular Expressions are used by the Match Regular Expression and Substitute Regular Expression commands.

Syntax

```
compile regular expression patternString
```

Parameters

patternString

This parameter is a string which is displayed in the dialog box. If you omit the `with prompt` parameter, no prompt is displayed.

For a description of the syntax of pattern strings, see the documentation for the UNIX grep command and the section *More about Regular Expressions* below. Information about Regular Expressions is also available in the THINK C *User's Guide*.

Result

The result is a compiled version of the patternString. This compiled pattern is used with the Match Regular Expression and Substitute Regular Expression commands.

Example

```
set pattern to compile ¬
  regular expression "(cat|dog)(fish|fight)"
```

Outcome:

```
tell current application
    compile regular expression "(cat|dog)(fish|fight)"
–> "<nonprintable characters>"
end tell
```

Errors

Compile Regular Expression returns errors when there is a problem with the expression being compiled. The error code returned is -50 (paramErr), and the error string contains the actual error. If you want to capture the actual error text, you could do the following:

```
try
    set theExpr to "(.*):(*)"
    set thePattern to compile regular expression theExpr
on error errMsg
    error "Can't compile \"" & theExpr & "\" - " & errMsg
end try
```

Match Regular Expression

The Match Regular Expression command matches a string to a Regular Expression and returns the portions of the string which match the regular expression.

Syntax

```
match regular expression compiledExpression
  to candidateString
```

Parameters

compiledExpression

This parameter is a compiled regular expression. This value is returned by the Compile Regular Expression command.

candidateString

This parameter is the string that is to be matched to the regular expression.

Result

The result of the Match Regular Expression command is a record containing the following values:

matched This Boolean value indicates if there was a match.

match string

This string value represents the largest match found.

match 1 This string value represents the portion of the string matching the first () expression.

match 2 This string value represents the portion of the string matching the second () expression.

match 3 This string value represents the portion of the string matching the third () expression.

match 4 This string value represents the portion of the string matching the fourth () expression.

match 5 This string value represents the portion of the string matching the fifth () expression.

match 6 This string value represents the portion of the string matching the sixth () expression.

match 7 This string value represents the portion of the string matching the seventh () expression.

match 8 This string value represents the portion of the string matching the eighth () expression.

match 9 This string value represents the portion of the string matching the ninth () expression.

Example

```
set pattern to ¬
   compile regular expression "This (.*) test"
set result to match regular expression pattern ¬
   to "This is a test"
{ result }
```

This script compiles a regular expression and then matches it to a string. It produces a match string and a match:

```
tell current application
   compile regular expression "This (.*) test"
—> "<nonprintable characters>"
   match regular expression "<nonprintable characters>"
—> {class:match reply, matched:true, match string:"This is a test", match 1:"is
a"}
end tell
```

Substitute Regular Expression

The Substitute Regular Expression command extracts the elements from a candidate string which match the patterns of a Regular Expression, and then substitutes the extracted elements into a template string.

Syntax

```
substitute regular expression compiledExpression
  of candidateString
  with templateString
```

Parameters

compiledExpression

This parameter is a compiled Regular Expression pattern. Regular Expressions are compiled using the Compile Regular Expression command.

candidateString

This parameter is a string representing the text which is to be compared to the Regular Expression, and then modified.

templateString

This parameter is a string representing a template for the substitutions which are to be performed. See the following section titled *Replacements for Regular Expressions,* for a description of the format of this string.

Result

The result is the substituted string.

Example

```
set pattern to ¬
   compile regular expression "This (.*) test"
substitute regular expression pattern ¬
   of "This is a test" with "–\\1–"
```

Outcome:

```
–is a–
```

Replacements for Regular Expressions

Within a template string, the following conventions apply:

- A backslash quotes the following character. The special characters within a template string are '`&`' and '`\`'; these are the only characters that need to be quoted. The construct "`\&`" produces a single '`&`' and the construct "`\\`" produces a single backslash.
- An ampersand (&) indicates the entire matched regular expression. For example, the replacement "`&&`" would consist of two copies of the matched expression.
- The sequence "`\`*n*", where n is a single digit, indicates the text matching the *n*th parenthesized component of the regular expression.

More about Regular Expressions

This section is taken with permission from Paul W. Abrahams and Bruce R. Larson's*UNIX for the Impatient* (Addison-Wesley, 1992), pp. 50-53.

A Regular Expression defines a pattern of text to be matched. The definition of a regular expression here is a sub-set of the regular expressions found on UNIX and other systems.

In general, any character appearing in a regular expression matches that character in the text. For example, the regular expression "elvis" matches the string "elvis". However, certain characters are used to specify variable patterns and are therefore special. In addition, other characters are special under particular conditions.

If you want to use a special character in a pattern, you must quote it, in effect, by escaping it with a preceding backslash (\). For example, the regular expression "cheap at $9\.98" matches the string "cheap at 9.98". Here the '.' needs to be quoted but the '$' does not because it isn't at the end of the string and therefore isn't special.

The meanings of the special characters are as follows:

\ The backslash quotes the character after it, whether special or not.

. The period matches any single character.

* A single character followed by an asterisk matches zero or more occurrences of that character. Similarly, a pattern that matches a set of characters followed by an asterisk matches zero or more characters from that set. In particular, '*' matches an arbitrary, possibly empty, string. The longest possible matching sequence is always used, although the matching mechanism is clever enough to consider the whole string when testing for a match. For example, it can discover that "^a.*b.c$" matches "axybbcc", even though this match requires that the ".*" should consume the first 'b', but not the second one.

+ The plus character following a regular expression matches one or more occurrences of that expression.

? The question mark character following a regular expression matches zero or one occurrence of that regular expression, i.e. it matches an optional regular expression.

$ A dollar sign at the end of an outermost regular expression matches the end of the line. Anywhere else in a regular expression, it matches itself.

^ A hat at the beginning of an outermost regular expression matches the beginning of a line. Anywhere else in a regular expression, it matches itself.

`[set]` A set of characters in square brackets matches any single character from the set. For example, "`[moxie]`" matches any of the characters (e i m o x). This notation is extended as follows:

- Within the set, the only characters with special meanings are (`- ] ^`). All other characters, even '\', stand for themselves.
- The notation *c1-c2* indicates the set of ASCII characters ranging from *c1* to *c2*. For example, "`[a-zA-Z_]`" matches any lowercase or uppercase letter, or an underscore. A minus sign at the beginning or end of the set stands for itself, however, this "`[+-]`" matches a plus or a minus.
- A right bracket as the first character of the set represents itself and does not end the set. (Within the set, a left bracket is not special.) Thus "`[] []`" matches a left bracket or a right bracket.
- The sequence `[^set]` matches any character that is *not* in *set*. (in this case a '`-`' or '`]`' following the initial '`^`' stands for itself, as above.) This "`[^0-9]`" matches any character except a digit.

Note that a set of characters can be followed by an asterisk. Thus "`[0-9a-f]*`" matches a possibly empty sequence of characters, each of which is either a digit or a letter between `a` and `f`.

`|` Two regular expressions separated by '`|`' match an occurrence of either of them, that is, the '`|`' operator acts as an *or*.

`(` Parentheses are used for grouping. For example, the pattern "`(cat|dog)(fish|fight)`" matches any line containing either "`catfish`", "`catfight`", "`dogfish`" or "`dogfight`".

Grouped expressions are also used to identify sections of a string which are to be substituted with the Substitute Regular Expression command.

{ You can follow a single character, or a regular expression that denotes a single character, with one of the following forms:

`{m}` `{m,}` `{m,n}`

Here m and n are non-negative integers less than 256. Let *S* be the set containing either the single character or the characters that match a regular expression.

- The first form denotes exactly m occurrences of characters belonging to *S*.
- The second form denotes at least m occurrences of characters belonging to *S*.
- The third form denotes between m and n occurrences of characters belonging to *S*.

  For example, "`[0-9]{2,}`" matches a sequence consisting of two or more digits.

Regular Expression Error Messages

The Regular Expression scripting addition returns error messages if it has problems compiling your expression. The following section contains some of the error messages that you are likely to see, and provides you with brief explanations of the messages.

Error: regexp too big

You will see this error message if the regular expression is larger than 32k.

Error: out of space

You will see this error message if you run out of memory while your regular expression is compiling.

Error: too many ()

You will see this expression if you have more than nine pairs of parentheses in your regular expression.

Error: unmatched ()

You will see this error message if you do not have the same number of left and right parentheses in your regular expression.

*Error: *+ operand could be empty*

A + or * operator must follow an atom. For instance, to match any number of "a" characters, you would use "a*" in your expression. If the "a" is missing, the expression is incomplete and will generate an error.

*Error: nested *?+*

You will see this error if you use "?*" in an expression. The expression is invalid because "?" is an operator. If you want to find any number of "?" characters, you would have to enter "\?*". Note that in AppleScript, you have to escape the "\" character as "\\?*".

Error: invalid [] range

You will see this error if you use an invalid character range in your expression. For example, the expression "[0-a]" is not a valid range of characters.

Error: unmatched []

You will see this error message if you do not have the same number of left and right brackets in your regular expression.

Error: ?+ follows nothing*

You will see this error message if you have not included an expression to repeat. The ?+* operators are repeating operators, and they must have an expression to repeat. For instance, [0-9]* indicates any number of digits.

*Error: trailing *

You will see this error message if your regular expression contains trailing backslashes.

Error: internal disaster

You will see this error message if there is an internal failure of the software.

Resource IO Addition

The Resource IO AppleScript addition allows you to write scripts which manipulate the contents of resource files. This includes both reading and writing resource data. This section describes each of the commands in the Resource IO addition.

CAUTION: *The commands provided in the Resource IO addition can damage your files, if used improperly. Please ensure that you fully understand how resources operate on the Macintosh before you begin using these commands. In particular, you should be familiar with Apple's ResEdit application and the Resource Manager chapters of the "Inside Macintosh" series of publications from Apple & Addison Wesley.*

AddResource

The addResource command adds a new resource to a resource file.

Syntax

```
addResource fileRefNum type resType id resID data
  resData
  [ name resName ]
  [ with/without purgeable ]
  [ with/without protected ]
  [ with/without preload ]
  [ with/without locked ]
  [ with/without system heap ]
```

Parameters

fileRefNum

The addResource command adds the new resource to the resource file referred to by this reference number. This value is returned by the openResourceFile command.

resType This parameter specifies the four-character resource type of the new resource. For example, `"PICT"`, `"TEXT"`.

resID This parameter specifies the ID of the new resource.

resData This parameter specifies the data for the new resource.

resName This optional parameter specifies the ID of the new resource. If this parameter is omitted, the resource is added with a blank name.

purgeable, protected, preload, locked, system heap

These optional parameters allow you to specify values for their corresponding resource status bits.

Result

None.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
addResource resFile of type "TEST" id 128 ¬
   name ¬
   "Testing" data "Testing"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 245:pf:Script Debugger Manual:Current Drafts:resource file"
   openResourceFile alias "Hard Disk 245:pf:Script Debugger Manual:Current
Drafts:resource file"
—> 7146
   addResource 7146 of type "TEST" id 128 name "Testing" data "Testing"
   closeResourceFile 7146
end tell
```

Errors

The addResource command can return any of the error codes provided by the ResError() toolbox function.

ChangeResource

The changeResource command changes the data, ID, name or status flags of a resource in a resource file.

Syntax

```
changeResource fileRefNum type resType
        id resID | name resName
        [ data resData ]
        [ newName newResName ]
        [ newID newResID ]
        [ with/without purgeable ]
        [ with/without protected ]
        [ with/without preload ]
        [ with/without locked ]
        [ with/without system heap ]
```

Parameters

fileRefNum

The changeResource command modifies a resource in the resource file referred to by this reference number. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter specifies the four-character resource type of resource you want to change. For example, `"PICT"`, `"TEXT"`.

resID This parameter specifies the ID of the resource being modified. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the resource being modified. If you specify this parameter, you cannot use the resID parameter.

resData This optional parameter specifies your new data for the resource being modified. If you do not provide this parameter, the resources data is not changed.

newResName

This optional parameter specifies a new name for the resource. If this parameter is omitted, the resource's Name is not changed.

newResID This optional parameter specifies a new ID for the resource. If this parameter is omitted, the resource's ID is not changed.

purgeable, protected, preload, locked, system heap
These optional parameters allow you to specify new values for the resources status bits.

Result

None.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
changeResource resFile of type "TEST" id 128 ¬
   data "New Data"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 245:pf:Script Debugger Manual:Current Drafts:resource file"
   openResourceFile alias "Hard Disk 245:pf:Script Debugger Manual:Current
Drafts:resource file"
—> 7146
   changeResource 7146 of type "TEST" id 128 data "New Data"
   closeResourceFile 7146
end tell
```

Errors

The changeResource command can return any of the error codes provided by the ResError() toolbox function.

ChangeStringResource

The changeStringResource command changes the data, ID, name or status flags of an `'STR '` resource in a resource file.

Syntax

```
changeStringResource fileRefNum
  id resID | name resName
  [ data resData ]
  [ newName newResName ]
  [ newID newResID ]
  [ with/without purgeable ]
  [ with/without protected ]
  [ with/without preload ]
  [ with/without locked ]
  [ with/without system heap ]
```

Parameters

fileRefNum

The changeStrResource command modifies a resource in the resource file referred to by this reference number. This value is returned by the openResourceFile and createResourceFile commands.

resID This parameter specifies the ID of the `'STR '` resource being modified. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the `'STR '` resource being modified. If you specify this parameter, you cannot use the resID parameter.

resData This optional parameter specifies your new string for the resource being modified. If you do not provide this parameter, the resources data is not changed.

newResName

This optional parameter specifies a new name for the resource. If this parameter is omitted, the resource's Name is not changed.

newResID This optional parameter specifies a new ID for the resource. If this parameter is omitted, the resource's ID is not changed.

purgeable, protected, preload, locked, system heap

These optional parameters allow you to specify new values for the resources status bits.

Result

None.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
changeStringResource resFile id 128 ¬
   data "New Data"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:junk file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:junk file"
—> 5736
   changeStringResource 5736 id 128 data "New Data"
   closeResourceFile 5736
end tell
```

Errors

The changeStringResource command can return any of the error codes provided by the ResError() toolbox function.

CloseResourceFile

The closeResourceFile command closes a resource file which was previously opened by the openResourceFile command.

Syntax

`closeResourceFile` *`fileRefNum`*

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

Result

None.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:junk file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:junk file"
—> 6676
   closeResourceFile 6676
end tell
```

Errors

The closeResourceFile command can return any of the error codes provided by the ResError() toolbox function.

Count1Resources

Use the count1Resources command to count the number of resources of a certain type in a resource file.

Syntax

```
count1Resources fileRefNum type resType
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter is the type code of the resources you want to count. Specify the type code as a four-character string, for instance: "PICT" or "TEXT".

Result

An integer value representing the number of resources of the type specified in the resource file.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
set numRsrcs to count1Resources resFile of type "PICT"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file"
—> 6488
   count1Resources 6488 of type "PICT"
—> 0
   closeResourceFile 6488
end tell
```

Errors

The count1Resources command can return any of the error codes provided by the ResError() toolbox function.

Count1ResourceTypes

Use the count1ResourceTypes command to count the number of different types of resources stored in a resource file.

Syntax

```
count1ResourceTypes fileRefNum
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

Result

An integer value representing the number of different types of resources stored in the resource file.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
set numRsrcTypes to count1ResourceTypes resFile
closeResourceFile resFile
```

Outcome:

```
tell current application
    choose file of type "rsrc"
—> alias "junk:resource file"
    openResourceFile alias "junk:resource file"
—> 8838
    count1ResourceTypes 8838
—> 6
    closeResourceFile 8838
end tell
```

Errors

The count1ResourceTypes command can return any of the error codes provided by the ResError() toolbox function.

CountResources

Use the countResources command to count the number of resources of a certain type in a chain of resource files.

The Macintosh Resource Manager maintains a chain of open resource files. The chain is extended each time a new resource file is opened. For more detailed information about resource file chains, consult the Resource Manager chapter of *Inside Macintosh*.

Syntax

```
countResources fileRefNum type resType
```

Parameters

fileRefNum
: This parameter specifies the reference number of the first resource file to search. This value is returned by the openResourceFile and createResourceFile commands.

resType
: This parameter is the type code of the resources you want to count. Specify the type code as a four-character string, for instance: "PICT" or "TEXT".

Result

An integer value representing the number of resources of the type specified in the open resource files.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
set numRsrcs to countResources resFile of type "PICT"
closeResourceFile resFile
```

Outcome:

```
tell current application
    choose file of type "rsrc"
—> alias "junk:resource file"
    openResourceFile alias "junk:resource file"
—> 8838
    countResources 8838 of type "PICT"
—> 42
    closeResourceFile 8838
end tell
```

Errors

The countResources command can return any of the error codes provided by the ResError() toolbox function.

CountResourceTypes

Use the countResourceTypes command to count the number of different types of resources stored in a chain of resource files.

The Macintosh Resource Manager maintains a chain of open resource files. The chain is extended each time a new resource file is opened. For more detailed information about resource file chains, consult the Resource Manager chapter of *Inside Macintosh*.

Syntax

```
countResourceTypes fileRefNum
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands..

Result

An integer value representing the number of different types of resources stored in the resource file.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile
set numRsrcTypes to count1ResourceTypes resFile
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:resource file"
—> 8838
   count1ResourceTypes 8838
—> 6
   closeResourceFile 8838
end tell
```

Errors

The count ResourceTypes command can return any of the error codes provided by the ResError() toolbox function.

CreateResourceFile

The createResourceFile command creates a new resource file.

Syntax

```
createResFile fileName
        [ in folder ]
        [ owner signature ]
        [ filetype type ]
```

Parameters

fileName This parameter is the new file's name.

folder This parameter is an alias to the folder where the new file is to be placed. If this parameter is omitted, the file is created in the current default folder.

signature This parameter defines the application signature code of the new file. Specify this value as a four-character string such as "MSWD" or "RSED".

If you omit this parameter, the new file is given the signature code "????". The Finder uses this information to link the file to the application which can open it.

type This parameter defines the file type code of the new file. Specify this value as a four-character string such as "TEXT" or "PICT".

If you omit the owner parameter, the new file is given the file type code "????".

Result

None.

Example

```
set newFile to choose new file ¬
   with prompt "Pick a new file name:"

createResourceFile (filename returned of newFile) ¬
   in (folder returned of newFile) ¬
   owner "RSED" — ResEdit
```

Outcome:

```
tell current application
   choose new file with prompt "Pick a new file name:"
—> {resource name:"junk file", replacing:false, folder returned:alias "Hard Disk
234:Desktop Folder:"}
   createResourceFile "junk file" in alias "Hard Disk 234:Desktop Folder:"
signature "RSED"
end tell
```

Errors

This command can return any of the errors which are returned by the toolbox ResError routine.

Get1IndexedResource

The get1IndexedResource command allows you to read a resource of a given type by index rather than ID or name. The get1IndexedResource command indexes through all the resources of the type specified in a single resource file. If you want to index through the resources in a chain of resource files, see the getIndexedResource command.

This command is useful if you do not know the names or IDs of the resources you want to access. Also, when combined with the countResources command, you can use the get1IndexedResource command to read all of the resources of a given type.

Syntax

```
get1IndexedResource fileRefNum
  type resType index resIndex
  [ as dataType ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter specifies the type of resource you want to read. Specify resource types as four-character strings such as "PICT" or "TEXT".

resIndex This parameter specifies the index of the resource you want to read. The value of this parameter must be in the range of 1 to the value returned by Count1Resources for *resType*.

dataType This parameter allows you to control the data type applied to the resource data. For instance, you can use this parameter to have your resource data treated as a string, picture, number or any other type of information.

Result

The result of this command is a Resource class. See the Resource Class sub-section below for more information.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set numRsrcs to count1Resources resFile of type "PICT"
set thePICTs to {}
repeat with i from 1 to numRsrcs
   set thePICTs to thePICTs & ¬
      {get1IndexedResource resFile of type "PICT" index i}
end repeat
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:resource file" for reading
—> 9214
   count1Resources 9214 of type "PICT"
—> 1
   get1IndexedResource 9214 of type "PICT" index 1
—> {class:resource, resType:"PICT", resID:128, resName:"", resData:«data
****0ADC000000000022002CF0C00F2C0000000000988034016100BA0183000000000000004E0000...
ECFA00FEFFED000BF20001F00FBF10001F9FFFD008F00000F9FE30002CD0002CD0002CD000000FF»,
resPurgeable:false, resPreload:false, resProtected:false, resLocked:false,
resSystemHeap:false}
   closeResourceFile 9214
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

Get1IndexedResourceType

The get1IndexedResourceType command allows you to obtain the type codes of each of the different resource types stored in a resource file.

Get1IndexedResourceType allows you to access the type codes available in a single resource file. If you want to access type codes in a chain of resource files, see the getIndexedResourceType command.

Syntax

```
get1IndexedResourceType fileRefNum index typeIndex
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

typeIndex This parameter specifies the index of the resource type you want to read. The value of this parameter must be in the range of 1 through the value of countResourceTypes for the resource file.

Result

The result of this command is a four-character string representing a resource type code.

Example

The following example builds a list containing the type codes of all the resources stored in a resource file.

```
set typesList to {}
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
repeat with i from 1 to count1ResourceTypes resFile
   set typesList to typesList & ¬
      {get1IndexedResourceType resFile index i}
end repeat
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for reading
—> 6394
   count1ResourceTypes 6394
—> 6
   get1IndexedResourceType 6394 index 1
—> "ics4"
   get1IndexedResourceType 6394 index 2
—> "ics8"
   get1IndexedResourceType 6394 index 3
—> "ics#"
   get1IndexedResourceType 6394 index 4
—> "icl4"
   get1IndexedResourceType 6394 index 5
—> "icl8"
   get1IndexedResourceType 6394 index 6
—> "ICN#"
   closeResourceFile 6394
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

Get1Resource

The get1Resource command allows you to read a resource. Get1Resource looks only in the resource file specified. See the getResource command if you want to search a chain of resource files for a resource.

Syntax

```
get1Resource fileRefNum type resType
  [ id resID ]
  [ name resName ]
  [ as resType ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter specifies the type of resource you want to read. Specify resource types as four-character strings such as "PICT" or "TEXT".

resID This parameter specifies the ID of the resource you want to read. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the resource you want to read. If you specify this parameter, you cannot use the resID parameter.

resType This parameter allows you to control the data type applied to the resource data. For instance, you can use this parameter to have your resource data treated as a string, picture, number or any other type of information.

Result

The result of this command is a Resource class. See the Resource Class sub-section below for more information.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set strData to get1Resource resFile of type "PICT" id 128
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "RSRC"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger Manual:AppleScript
icons.rsrc"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
Manual:AppleScript icons.rsrc" for reading
—> 7804
   get1Resource 7804 of type "PICT" id 128
—> {class:resource, resType:"PICT", resID:128, resName:"", resData:«data
****0ADC000000000022002C02FF0C0FF002C00000000001000A00002C0098803401610 0BA300ED00000000000004E0000...
ECFA00FEFFED000BF20001F9FFFFEA10000001F9FFFD00FDFFE700000F9FDFFE30002C02CD0002CD000000FF»,
resPurgeable:false, resPreload:false, resProtected:false, resLocked:false,
resSystemHeap:false}
   closeResourceFile 7804
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetIndexedResource

The getIndexedResource command allows you to read the resources of a given type by index rather than ID or name. The getIndexedResource command indexes through all the resources of the type specified in the entire resource file chain. If you want to index through the resources in a single file, see the Get1IndexedResource command.

This command is useful if you do not know the names or IDs of the resources you want to access. Also, when combined with the countResources command, you can use the getIndexedResource command to read all of the resources of a given type.

Syntax

```
getIndexedResource fileRefNum
  type resType index resIndex
  [ as dataType ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter specifies the type of resource you want to read. Specify resource types as four-character strings such as "PICT" or "TEXT".

resIndex This parameter specifies the index of the resource you want to read. The value of this parameter must be in the range of 1 to the value returned by CountResources for *resType*.

resType This parameter allows you to control the data type applied to the resource data. For instance, you can use this parameter to have your resource data treated as a string, picture, number or any other type of information.

Result

The result of this command is a Resource class. See the Resource Class sub-section below for more information.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set numRsrcs to countResources resFile of type "PICT"
set thePICTs to {}
repeat with i from 1 to numRsrcs
   set thePICTs to thePICTs & ¬
      {getIndexedResource resFile of type "PICT" index i}
end repeat
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "RSRC"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger Manual:AppleScript
icons.rsrc"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
Manual:AppleScript icons.rsrc" for reading
—> 7522
   countResources 7522 of type "PICT"
—> 40
   getIndexedResource 7522 of type "PICT" index 1
—> {class:resource, resType:"PICT", resID:128, resName:"", resData:«data
****0ADC000000000201102FF0000000220000000000000022002C0098300E000004E000000400000000080001000800000
....
   getIndexedResource 7522 of type "PICT" index 40
—> {class:resource, resType:"PICT", resID:-5694, resName:"", resData:«data
****018E000000000100010001102FF0C00F000000480000001000100000001000A00010990002F9000000FF»,
resPurgeable:true, resPreload:false, resProtected:false, resLocked:false,
resSystemHeap:false}
   closeResourceFile 7522
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetIndexedResourceType

The getIndexedResourceType command allows you to obtain the type codes of each of the different resource types stored in a chain of resource files. If you want to access type codes in a single resource file, see the get1IndexedResourceType command.

Syntax

```
getIndexedResourceType fileRefNum index typeIndex
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile.

typeIndex This parameter specifies the index of the resource type you want to read. The value of this parameter must be in the range of 1 through the value of countResourceTypes for the resource file.

Result

The result of this command is a four-character string representing a resource type code.

Example

The following example builds a list containing the type codes of all the resources available to the script.

```
set typesList to {}
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
repeat with i from 1 to countResourceTypes resFile
   set typesList to typesList & ¬
      {getIndexedResourceType resFile index i}
end repeat
closeResourceFile resFile
```

Outcome:

```
{"cfrg", "gmra", "hman", "pflp", "fwst", "sfnt", "NFNT", "KCOD", "IDLE", "DKND",
"LOCN", "PREF", "FLSH", "THSF", "sysz", "nrct", "mach", "OPTS", "DICS", "ACTV",
"BARY", "SARY", "PHON", "STXT", "crsr", "emg1", "cdev", "PROC", "DCOD", "dtn ",
"audt", "movp", "QPrf", "APrf", "alis", "atlk", "iopc", "ltlk", "AINI", "drvr",
"lmgr", "picb", "fmap", "ROvr", "FOND", "wedg", "pixs", "pmap", "rtt#", "FMTR",
"ppcc", "ADBS", "CDEF", "DRVR", "DSAT", "FKEY", "FRSV", "ICON", "INTL", "KCAP",
"KCHR", "KMAP", "KSWP", "LDEF", "MACS", "MBDF", "PACK", "PAT ", "PAT#", "PTCH",
"RECT", "ROv#", "boot", "bst#", "card", "cctb", "clut", "cmtb", "ctb ", "dbex",
"dcmp", "dimg", "enet", "fld#", "flst", "gbly", "indl", "inpm", "itl0", "itl1",
"itl2", "itl4", "itlb", "itlc", "itlm", "kcs#", "kcs4", "lodr", "lpch", "lstr",
"mcky", "mitq", "mntr", "ppat", "ppci", "ppt#", "proc", "pslt", "ptbl", "ptch",
"resf", "rgb ", "snd ", "snth", "vm  ", "wctb", "scod", "lmem", "TWIT", "MDRW",
"MPNT", "MACA", "mst#", "mstr", "INIT", "DLOG", "DITL", "xMAP", "xRSC", "WDEF",
"hfdr", "hmnu", "hdlg", "scsz", "Xete", "RDls", "Pane", "pltt", "MDEF", "dctb",
"DBsc", "TEXT", "BNDL", "asDB", "StdP", "cicn", "FREF", "acur", "FONT", "CURS",
"PICT", "ScPn", "Estr", "MBAR", "SICN", "STR#", "STR ", "WIND", "MENU", "ALRT",
"CNTL", "SIZE", "CODE", "DREL", "ZERO", "DATA", "TEST", "ckid", "vers", "c++ ",
"dcmd", "mxbc", "mxbi", "mxwt", "mxbm", "TMPL", "ics4", "ics8", "ics#", "icl4",
"icl8", "ICN#"}
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetIndexedStringResource

The getIndexedStringResource command allows you to read the text data stored in an "STR#" resource.

Each "STR#" resource contains a series of strings. GetIndexedStringResource allows you to access any of the strings stored in an "STR#" resource .

Syntax

```
getIndexedStringResource fileRefNum
  [ id resID ]
  [ name resName ]
  index strIndex
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resID This parameter specifies the ID of the "STR#" resource you want to read. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the "STR#" resource you want to read. If you specify this parameter, you cannot use the resID parameter.

strIndex This parameter specifies the index of the string within the "STR#" resource you want to read.

Result

The result of this command is a string.

Example

```
set filePath to ¬
  choose file with prompt ¬
    "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set strData to getIndexedStringResource id 128 index 1
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for reading
—> 5830
   getIndexedStringResource id 128 index 1
—> "quitting"
   closeResourceFile 5830
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetResource

The getResource command allows you to read a resource. GetResource searches through a chain of resource files to locate your resource. See the get1Resource command if you want to search through a single resource file.

Syntax

```
getResource fileRefNum type resType
  [ id resID ]
  [ name resName ]
  [ as resType ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile, and createResourceFile commands.

resType This parameter specifies the type of resource you want to read. Specify resource types as four-character strings such as "PICT" or "TEXT".

resID This parameter specifies the ID of the resource you want to read. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the resource you want to read. If you specify this parameter, you cannot use the resID parameter.

resType This parameter allows you to control the data type applied to the resource data. For instance, you can use this parameter to have your resource data treated as a string, picture, number or any other type of information.

Result

The result of this command is a Resource class. See the Resource Class sub-section below for more information.

Example

This script returns the hexadecimal version of the PICT.

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set strData to getResource of type "PICT" id 128
closeResourceFile resFile
```

Outcome:

```
Ttell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for reading
—> 6582
   getResource of type "PICT" id 128
—> {class:resource, resType:"PICT", resID:128, resName:"", resData:«data
****0D2A0152000D027101DB2FF0C00FFF015200001DB000271000000000001E001FFFF0001000A0152000D027101DB00080031000E027101D7
...
0038001BFFFF00000000003101520ID101D700C0F60011F40001FE900061FFFE0000FFFFF5000CD000101FFE00010FF8F50002C70000FF»,
resPurgeable:false, resPreload:false, resProtected:false, resLocked:false,
resSystemHeap:false}
   closeResourceFile 6582
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetStringResource

The getStringResource command allows you to read the text data stored in an "STR " resource.

Syntax

```
getStringResource fileRefNum
   [ id resID ]
   [ name resName ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile, createResourceFile, and getCurResFile commands.

resID This parameter specifies the ID of the "STR" resource you want to read. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the "STR" resource you want to read. If you specify this parameter, you cannot use the resID parameter.

Result

The result of this command is a string.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
set strData to getStringResource id 128
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for reading
—> 7146
   getStringResource id 128
—> "Script Application

"
   closeResourceFile 7146
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

OpenResourceFile

The openResourceFile command opens a resource file (or the resource fork of any other type of file) for reading, updating, or writing. This command, when used with the other commands provided by the Resource IO addition, allows you to manipulate resources within scripts.

Syntax

```
openResourceFile file
        [ for reading|update|writing ]
```

Parameters

file This parameter is an alias to the file which is to be opened.

Result

The result is a resource file reference number. You must provide this number to all other commands you issue when processing the resource file.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath for reading
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource
file" for reading
—> 6582
   closeResourceFile 6582
end tell
```

NOTES: *When the optional* `for` *parameter is not specified, the file is opened for update.*

Be careful to ensure you close all the files you open. Due to the nature of AppleScript additions, the openResourceFile command does not ensure the file is closed when a script aborts without first closing the file with the closeResourceFile command.

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

RemoveResource

The removeResource command removes a resource from a resource file.

Syntax

```
removeResource fileRefNum type resType
  [ id resID ]
  [ name resName ]
```

Parameters

fileRefNum

This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter specifies the type of resource you want to remove. Specify resource types as four-character strings such as "PICT" or "TEXT".

resID This parameter specifies the ID of the resource you want to remove. If you specify this parameter, you cannot use the resName parameter.

resName This parameter specifies the Name of the resource you want to remove. If you specify this parameter, you cannot use the resID parameter.

Result

None.

Example

```
set filePath to ¬
   choose file with prompt ¬
      "Select a file to open:" of type "rsrc"
set resFile to openResourceFile filePath
removeResource resFile of type "PICT" id 128
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file with prompt "Select a file to open:" of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file"
—> 5548
   removeResource 5548 of type "PICT" id 128
   closeResourceFile 5548
end tell
```

Errors

This command can return any of the errors which are returned by the ResError() toolbox routine.

GetUnique1ResourceID

Use the getUnique1ResourceID command to find a unique ID for a given type of resource. The getUnique1ResourceID command ensures that the value it returns is unique within a single resource file.

Syntax

```
getUnique1ResourceID fileRefNum type resType
```

Parameters

fileRefNum This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter is the type code of the resources you want to count. Specify the type code as a four-character string, such as: "PICT" or "TEXT".

Result

The new unique resource ID.

Errors

The getUnique1ResourceID command can return any of the error codes provided by the ResError() toolbox function.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile for writing
set newResID to 0
repeat while newResID < 128
   set newResID to getUnique1ResourceID resFile of type "TEST"
end repeat
addResource resFile ¬
   of type ¬
   "TEST" id newResID ¬
   data "Testing"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for writing
—> 6958
   getUnique1ResourceID 6958 of type "TEST"
—> 19718
   addResource 6958 of type "TEST" id 19718 data "Testing"
   closeResourceFile 6958
end tell
```

GetUniqueResourceID

Use the getUniqueResourceID command to find a unique ID for a given type of resource. The getUniqueResourceID command ensures that the value it returns is unique within an entire chain of resource files.

Syntax

```
getUniqueResourceID fileRefNum type resType
```

Parameters

fileRefNum This parameter specifies the reference number of an open resource file. This value is returned by the openResourceFile and createResourceFile commands.

resType This parameter is the type code of the resources you want to count. Specify the type code as a four-character string, such as: "PICT" or "TEXT".

Result

The new unique resource ID.

Errors

The uniqueResourceID command can return any of the error codes provided by the ResError() toolbox function.

Example

```
set theFile to choose file of type "rsrc"
set resFile to openResourceFile theFile for writing
set newResID to 0
repeat while newResID < 128
   set newResID to getUniqueResourceID resFile of type "TEST"
end repeat
addResource resFile ¬
   of type ¬
   "TEST" id newResID ¬
   data "Testing"
closeResourceFile resFile
```

Outcome:

```
tell current application
   choose file of type "rsrc"
—> alias "Hard Disk 234:Desktop Folder:Script Debugger 1.0:resource file"
   openResourceFile alias "Hard Disk 234:Desktop Folder:Script Debugger
1.0:resource file" for writing
—> 6958
   getUniqueResourceID 6958 of type "TEST"
—> 19718
   addResource 6958 of type "TEST" id 19718 data "Testing"
   closeResourceFile 6958
end tell
```

Resource Class

The various GetResource commands of the Resource IO addition return resource information in the form of a Resource class.

The resource class has the following properties:

resID This property corresponds to the resource's ID.

resName This property corresponds to the resource's Name.

resData This property contains a resource's data. The type of this property is governed by the AS parameter of the GetResource commands. If the AS parameter is not specified, the GetResource commands use the data type "anything".

resPurgeable
This Boolean property represents the purgeable bit of the resource's status bits.

resPreload
This Boolean property represents the preload bit of the resource's status bits.

resProtected
This Boolean property represents the protected bit of the resource's status bits.

resLocked
This Boolean property represents the locked bit of the resource's status bits.

resSystemHeap
This Boolean property represents the system heap bit of the resource's status bits.

Screens Addition

List Screens

The List Screens command obtains detailed information about each of the Macintosh's display screens.

Syntax

```
list screens
```

Result

The result of the List Screens command is a list of records. Each record describes a different display screen. The records contain the following values:

main screen
: This Boolean value indicates whether or not the screen is the main screen. The main screen is the screen containing the menu bar.

bit depth
: This value represents the number of bits in the display screen.

bounds
: This value is the screen's bounding rectangle.

Example

```
list screens
```

Outcome:

```
—> {{class:screen information, main screen:true, bit depth:8, bounds:{0, 0, 640,
870}}, {class:screen information, main screen:false, bit depth:8, bounds:{-1024,
0, 0, 768}}}
```

Shutdown Addition

The Shutdown addition allows your scripts to shutdown the Macintosh.

Shutdown

The Shutdown command shuts down and optionally restarts your Macintosh.

Syntax

```
shutdown
   [ with restart ]
```

Result

This command returns no result.

Example

```
set result to display dialog ¬
   "Are you sure you want to shutdown?" ¬
   buttons {"Shutdown", "Restart", "Cancel"} ¬  default button "Cancel"
if button returned of result = "Shutdown" then ¬  shutdown
if button returned of result = "Restart" then ¬ shutdown with restart
```

Outcome:

```
tell current application
   display dialog "Are you sure you want to shutdown?" buttons {"Shutdown",
"Restart", "Cancel"} default button "Cancel"
—> User cancelled.
end tell
```

Speech Addition

The Speech addition allows your scripts to use the services of Apple's Speech Manager within your scripts. This section describes each of the AppleScript commands provided in the Speech addition.

To use the Speech addition, you will need version 1.1.1 or later of Apple's Speech Manager software.

Speak

The Speak command uses the Apple Macintosh Speech Manager to speak text strings. Note that because of its dependency on the Speech Manager, this command only operates on Macintoshes which have the Speech Manager installed.

If you enter the Command-. while the speak command is speaking a long text string, the command is aborted and a userCanceledErr is returned.

Syntax

```
speak message
  [ voice voice ]
  [ rate rate ]
  [ pitch pitch ]
```

Parameters

message This parameter is the text you want to have spoken.

voice This optional parameter allows you to specify the name of the voice you want used when the message is spoken.

rate This optional parameter specifies the rate at which your message is spoken. Express the rate as a number representing words per minute.

pitch This optional parameter specifies the pitch at which your message is spoken.

Result

None.

Example

```
speak "The wind blows mainly in the plains"
```

List Voices

The List Voices command obtains a list of the names of the voices available. Note that because of its dependency on the Speech Manager, this command only operates on Macintoshes which have the Speech Manager installed.

Syntax

```
list voices
```

Parameters

None.

Result

The result is a list of strings representing the names of all the Speech Manager voices.

Example

```
list voices
```

Outcome:

```
—> {“Zarvox”, “Whisper”, “Trinoids”, “Ralph”, “Princess”, “Kathy”, “Junior”, “Good News”, “Bad News”, “Fred”}
```

Get Voice

The Get Voice command returns detailed information about a particular Speech Manager voice. Note that because of its dependency on the Speech Manager, this command only operates on Macintoshes which have the Speech Manager installed.

Syntax

```
get voice voice
```

Parameters

voice This parameter specifies the name of the voice you want information about.

Result

The result of the Get Voice command is a record containing the following values:

voice version

This integer value represents the voice's version number.

voice name

This string is the voice's name.

comment

This string further describes the voice.

gender This integer value defines the gender of the voice—1 = neuter, 2 = male and 3 = female.

age This integer value represents the approximate age of the voice.

voice script

This integer corresponds with the voice's script code.

language

This integer value is the voice's language code.

Example

```
get voice (first item of (list voices))
```

Outcome:

```
tell current application
   list voices
—> {“Zarvox”, “Whisper”, “Trinoids”, “Ralph”, “Princess”, “Kathy”, “Junior”, “Good
News”, “Bad News”, “Fred”}
   get voice “Zarvox”
—> {class:voice information, voice version:259, voice name:”Zarvox”, comment:”That
looks like a peaceful planet.”, gender:0, age:1, voice script:0, language:0,
region:0}
end tell
```

APPENDIX D

Scheduler

Scheduler is a utility that lets you schedule the launch of applications and the opening of documents in response to a wide variety of events. For example, you can create an event in Scheduler that can open a spreadsheet regularly each week. It is fully scriptable and recordable so you can write scripts using AppleScript or any other OSA language to automate common activities. Scheduler's recordability allows you to create scripts by recording your actions; this makes it easy to learn how to script Scheduler.

Because of its flexibility, Scheduler has applications in Network Management, Telecommunications, Business Systems and many other areas where actions within your application need to be performed on a regular basis. Since it is scriptable, you can program your Macintosh to perform tasks involving a number of applications.

Scheduler Components

When you ran the Script Debugger Installer, it installed the Scheduler components if you chose the Easy Install option or if you selected Scheduler in the Custom Install. If you look in your Control Panels folder, you will find the Scheduler Setup control panel file. You will also find the Scheduler extension file in your Extensions folder.

NOTE: *The Scheduler extension is a fat binary so it will work with both 68K and Power Macintoshes.*

To use Scheduler effectively, your Macintosh computer must be running system software version 7.0 or later with at least two megabytes of memory. Four megabytes is the recommended amount of RAM for Scheduler.

The following sections in this appendix show you how to create an event in Scheduler. You can refer to Chapter 3 to see how Scheduler's recordability works. You can find out more about the events and classes that Scheduler supports by referring to its dictionary.

Scheduling An Application or Document Event

Scheduler makes it very easy to schedule the launch of applications and documents. This example shows you how to use Scheduler to open the Read Me First file on the Script Debugger Install disk whenever the disk is inserted into your Macintosh's disk drive.

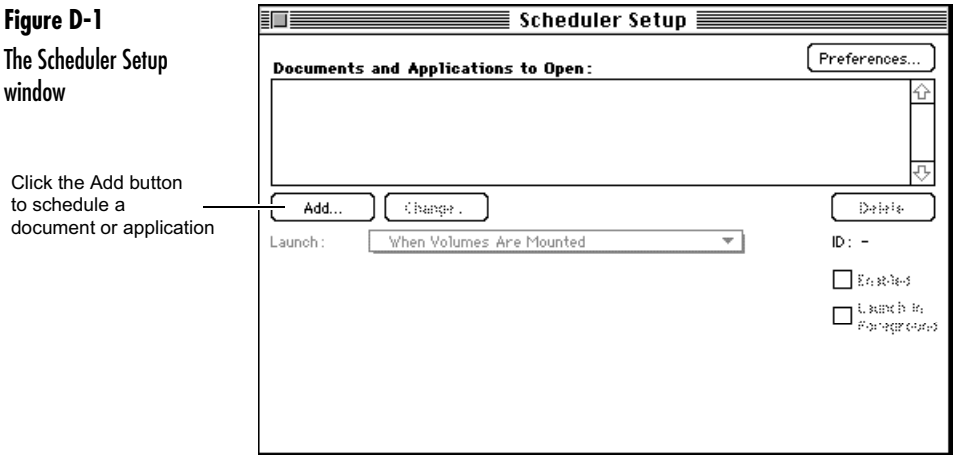
1. **Double-click the Scheduler Setup control panel.**

The Scheduler Setup file is located in the Control Panels folder within your System Folder.


Scheduler Setup

2. Click Add (Figure D-1).

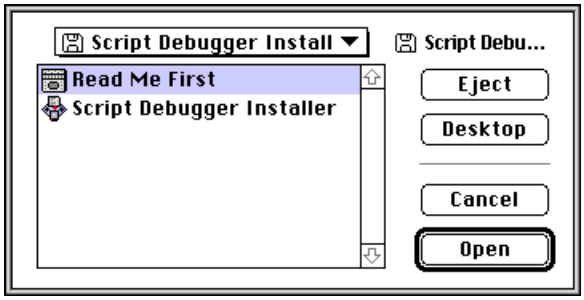
Figure D-1
The Scheduler Setup window



Click the Add button to schedule a document or application

When you click Add, Scheduler Setup presents a file selection dialog box *(Figure D-2)*.

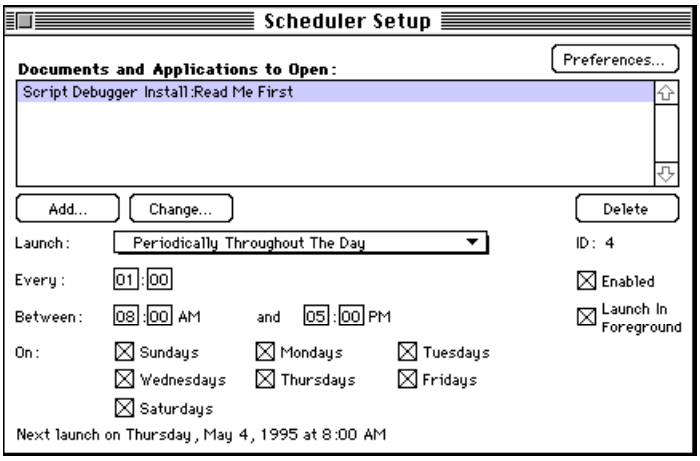
Figure D-2
Scheduler file selection dialog box



Use this dialog box to locate the application or document you want to schedule. For this example, select the Read Me First file and click the Open button.

Once you have selected a file, the Scheduler Setup window will look like the one in Figure D-3.

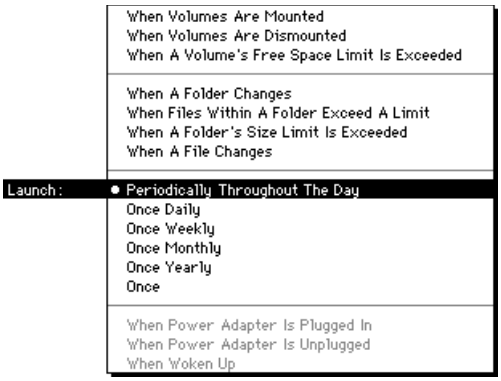
Figure D-3
The Read Me First file selected



3. Select a scheduling type.

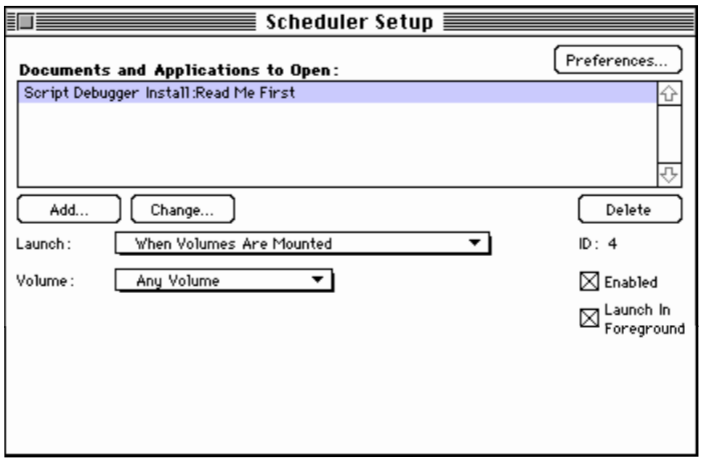
By default, Scheduler Setup schedules your document to execute "Periodically Throughout The Day". In this example, choose "When Volumes Are Mounted" from the Launch pop-up menu *(Figure D-4)*.

Figure D-4
Scheduler Setup's Launch menu



Once you have selected a scheduling option, the bottom portion of the Scheduler Setup window changes to display scheduling parameters *(Figure D-5)*.

Figure D-5
Scheduler with the scheduling parameters set

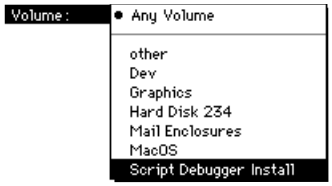


4. Set scheduling parameters.

Once you have selected the type of scheduling you want, you can provide additional information governing when your document or application is opened. In this example, choose the Script Debugger Installer disk which contains Read Me First.

Choose Script Debugger Install from the Volume pop-up menu *(Figure D-6)*.

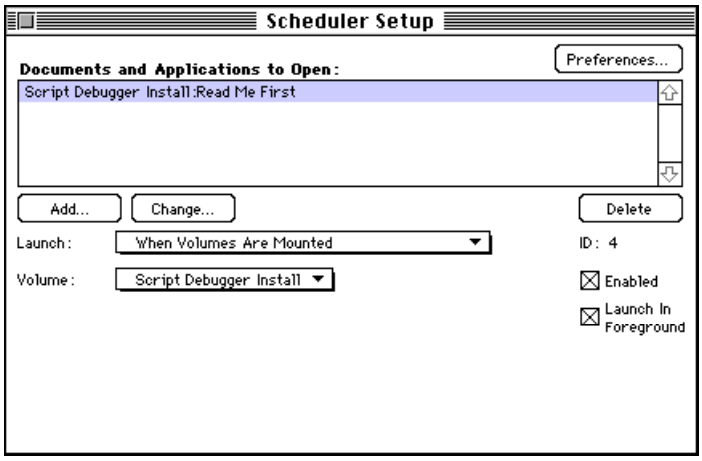
Figure D-6
The Volume pop-up menu



5. Close Scheduler Setup.

You have finished scheduling the Read Me First document. The Scheduler Setup window should look something like Figure D-7.

Figure D-7
The completed Scheduler window



All scheduling changes made with Scheduler Setup take effect immediately after the Scheduler Setup window is closed.

6. Dismount the Script Debugger Install disk by dragging it to the Trash.

7. Re-insert the Scheduler Install disk into your Macintosh.

When you insert the Script Debugger Install diskette, Scheduler opens the Read Me First file.

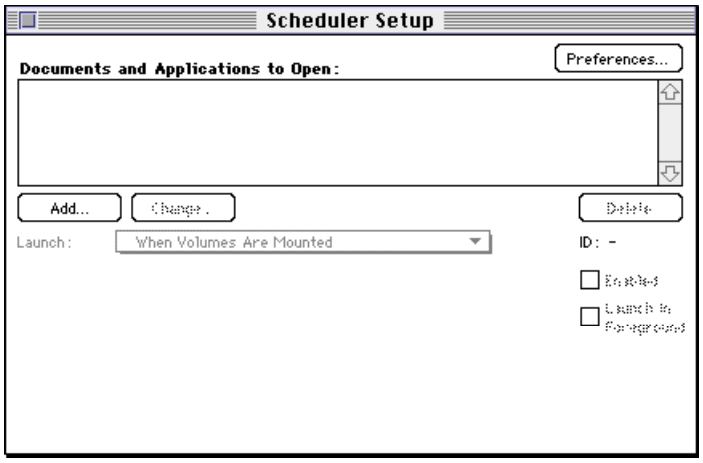
Scheduler Features

Scheduler is a powerful and flexible tool for scheduling the opening of documents and applications. This section describes all of the advanced Scheduler features.

Adding, Setting and Deleting Scheduled Events

The Scheduler Setup control panel is your interface to Scheduler. Scheduler Setup allows you to add, change and delete scheduled documents and applications *(Figure D-8)*.

Figure D-8
The Scheduler Setup window



Add Click the Add button to add a new document or application to the list of scheduled documents and applications.

When you click the Add button, a file selection dialog box appears so that you can choose the application or document to be launched.

Change Click the Change button to change the application or document being scheduled.

The Change button is active only when a scheduled document or application is selected.

Delete Click the Delete button to remove a document or application from the list of scheduled documents and applications.

The Delete button is active only when a document or application is selected.

Launch Use the Launch pop-up menu to choose the type of scheduling for your document or application.

The Launch pop-up menu is active only when a document or application is selected *(Figure D-9)*.

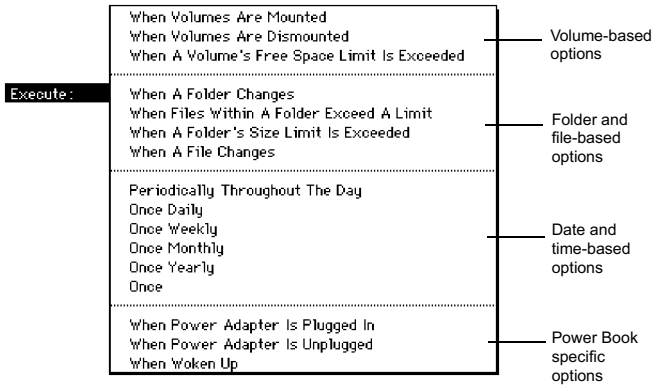


Figure D-9
Scheduler Setup's Launch menu

Date and Time-Based Scheduling

Scheduler provides a series of options for launching applications and documents at a specific time in the future. Using these options, you can configure your Macintosh to perform repetitive operations periodically throughout the day, week, month or year. You can also schedule applications and documents to launch at a specific date and time in the future.

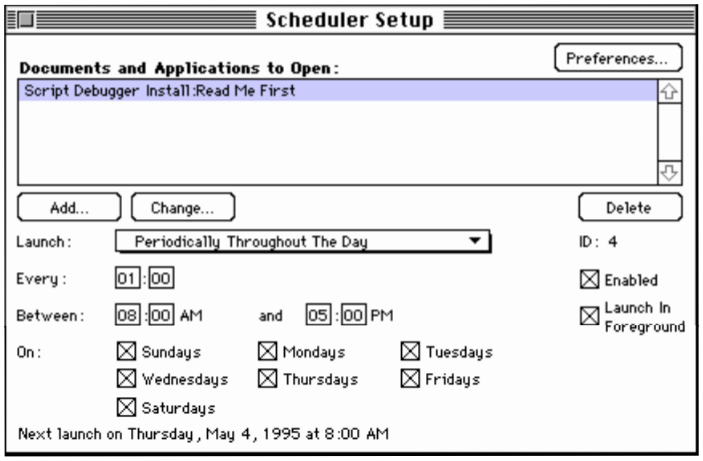
The following sub-sections describe each of Scheduler's time-based options in detail.

Scheduling Events Periodically Throughout the Day

This scheduling option allows you to have your application or document opened at regular intervals throughout the day.

Select this option by choosing "Periodically Throughout The Day" from the Launch pop-up menu *(Figure D-10)*.

Figure D-10
The "Periodically Throughout The Day" settings

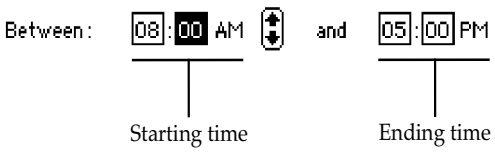


The "Periodically Throughout The Day" scheduling option provides the following parameters:

Every: This parameter defines the interval, in hours and minutes, between launches. To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

Between: This parameter specifies the period of the day when scheduling occurs.

As with the Every parameter, to change either the start or ending time, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears to adjust the value.



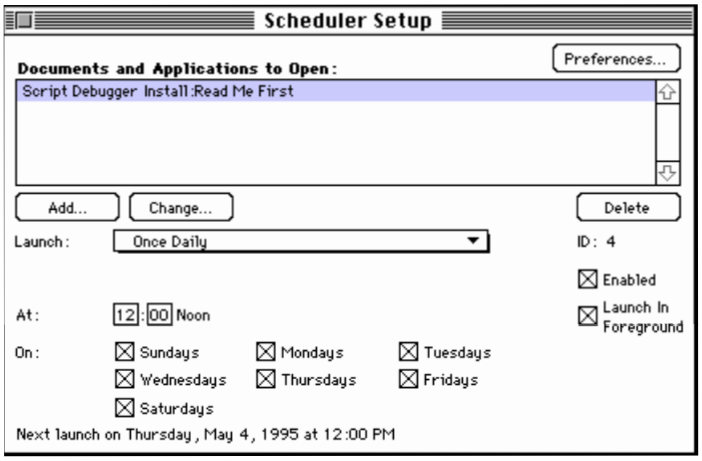
On: This parameter specifies the days of the week when you want scheduling to occur. To prevent scheduling on a particular day, simply uncheck that day's checkbox.

Scheduling Daily Events

This scheduling option causes your documents and applications to open once daily.

Select this option by choosing "Once Daily" from the Launch pop-up menu *(Figure D-11)*.

Figure D-11
The "Once Daily" settings



The “Once Daily” scheduling option provides the following parameters:

At: This parameter defines the time of the day, in hours and minutes, when the document or application is to be opened.

To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

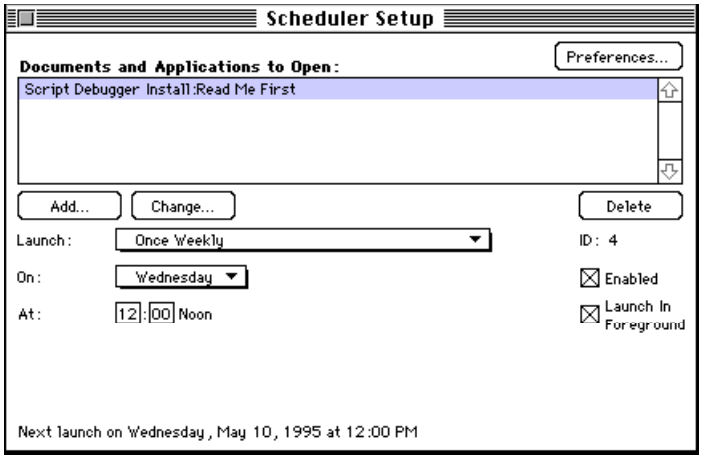
On: This parameter specifies the days of the week when you want your application or document to open.

Scheduling Weekly Events

This scheduling option allows you to have your application or document opened once each week.

Select this option by choosing “Once Weekly” from the Launch pop-up menu *(Figure D-12)*.

Figure D-12
The “Once Weekly” settings

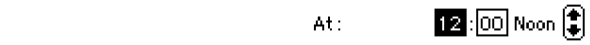


The "Once Weekly" scheduling option provides the following parameters:

On: This parameter specifies the day of the week when you want your application or document to open.

At: This parameter defines the time of the day, in hours and minutes, when the document or application is to be opened.

To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

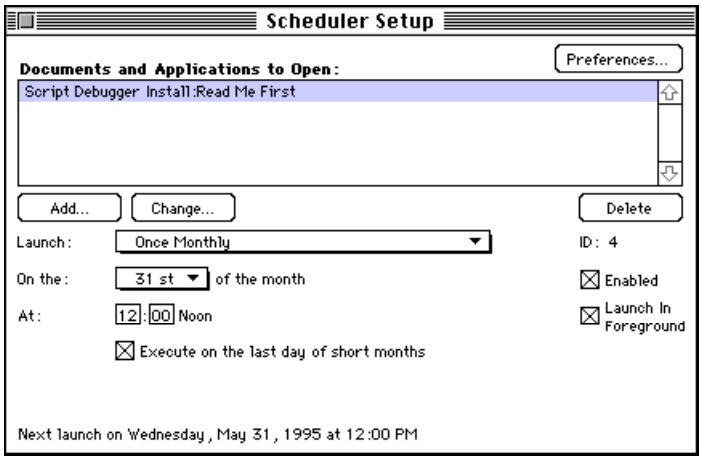


Scheduling Monthly Events

This scheduling option allows you to have your application or document opened once each month.

Select this option by choosing "Once Monthly" from the Launch pop-up menu *(Figure D-13)*.

Figure D-13
The "Once Monthly" settings



The “Once Monthly” scheduling option provides the following parameters:

On the: This parameter specifies the day of the month when you want your application or document to open.

At: This parameter defines the time of the day, in hours and minutes, when the document or application is to be opened.

To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

Execute on the last day of short months:

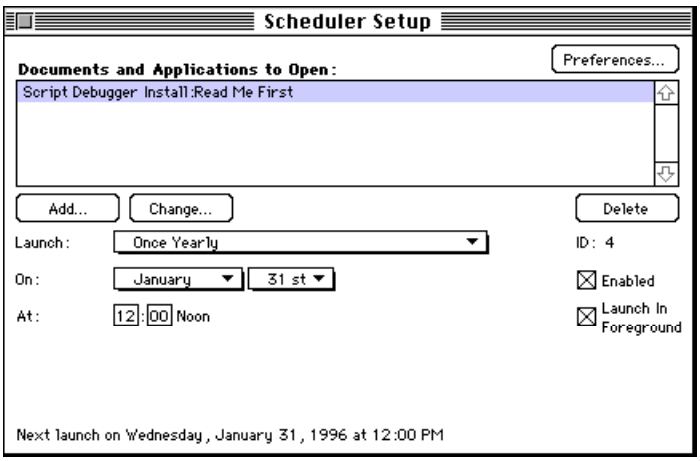
This parameter describes what happens in months with fewer days than the day you selected in the On The parameter. When this parameter is checked, your document or application is operated in the last day of the month. When not checked, your document or application is not opened in short months.

Scheduling Yearly Events

This scheduling option allows you to have your application or document opened once each year.

Select this option by choosing "Once Yearly" from the Launch pop-up menu *(Figure D-14)*.

Figure D-14
The "Once Yearly" settings



The "Once Yearly" scheduling option provides the following parameters:

On: This parameter lets you specify the month and day of the year when your application or document is to be opened.

At: This parameter defines the time of the day, in hours and minutes, when the document or application is to be opened.

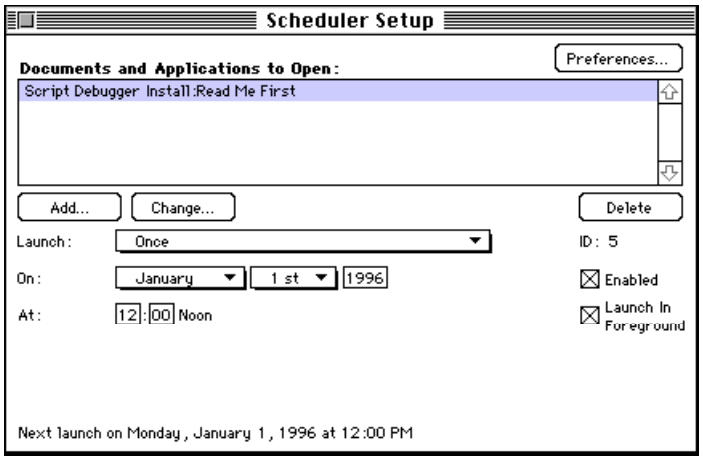
To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

Scheduling Events That Occur at a Specific Date and Time

This scheduling option allows you to have your application or document opened on a specific day and time in the future.

Select this option by choosing "Once" from the Launch pop-up menu *(Figure D-15)*.

Figure D-15
The "Once" settings



The "Once" scheduling option provides the following parameters:

On: This parameter lets you specify the month, day and year when your application or document is to be opened.

At: This parameter defines the time of the day, in hours and minutes, when the document or application is to be opened.

To change this value, click on the hours or minutes field with the mouse. Then use the up/down arrow that appears, to adjust the value.

File and Folder-Based Scheduling

Scheduler provides a series of options for launching applications and documents whenever a file or folder is changed. You can use these options to configure your Macintosh to perform repetitive tasks when files are placed in a folder or whenever a file is edited.

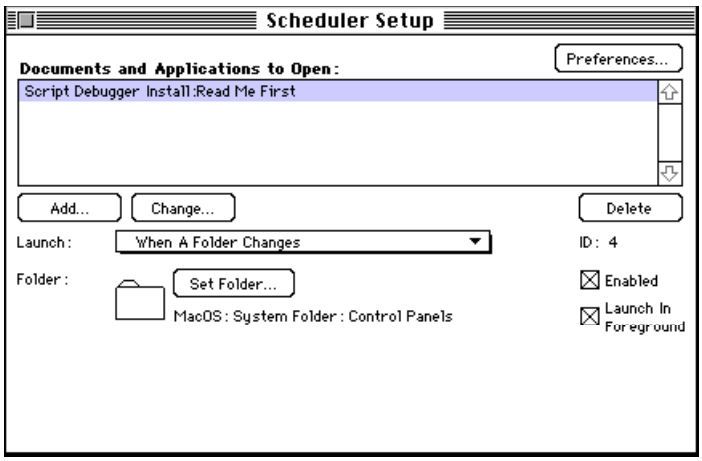
The following sub-sections describe each of Scheduler's file and folder options in detail.

Scheduling Events When Folders Are Changed

This scheduling option allows you to have your application or document opened whenever files are added to or removed from a folder. Scheduler recognizes changed folders by periodically checking their modification dates.

Select this option by choosing "When A Folder Changes" from the Launch pop-up menu *(Figure D-16)*.

Figure D-16
The "When A Folder Changes" settings



The "When A Folder Changes" scheduling option provides the following parameter:

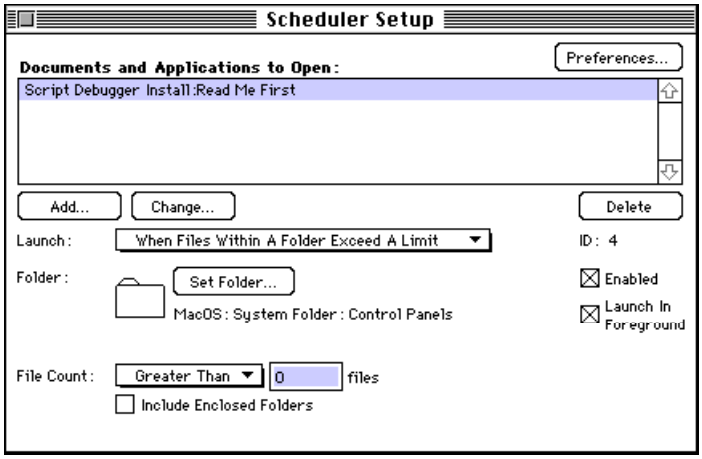
Folder: The folder parameter allows you to select the folder which triggers your application or document. Click the Set Folder button to set or change the folder.

Scheduling Events When the Number of Files In a Folder Reaches a Limit

This scheduling option allows you to have your application or document opened whenever the number of files stored in a folder rises above or falls below a limit.

Select this option by choosing "When Files Within A Folder Exceed A Limit" from the Launch pop-up menu *(Figure D-17)*.

Figure D-17
The "When Files Within A Folder Exceed A Limit" setting



The "When Files Within A Folder Exceed A Limit" scheduling option provides the following parameter:

Folder: The folder parameter allows you to select the folder which triggers your application or document. Click the Set Folder button to set or change the folder.

File Count:

This parameter allows you to define a limit for the number of files stored in the folder. The pop-up menu lets you choose the type of limit on free space, either Greater Than or Less Than.

Once you have selected the type of limit, you can enter the number of files into the text box. Use the Include Enclosed Folders checkbox to indicate whether or not you want Scheduler to include the files stored within enclosed folders when it calculates the number of files stored in the folder.

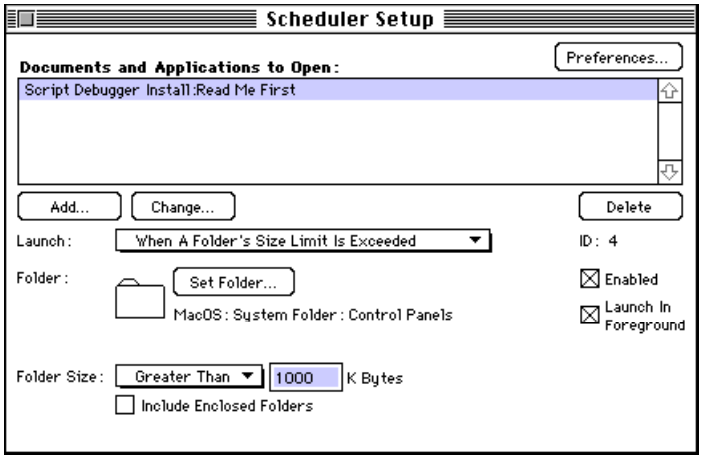
CAUTION: *Use the Include Enclosed Folders feature with care. When this feature is checked, Scheduler must scan through all the enclosed folders to calculate the number of files. If you have a deep folder structure, Scheduler requires extra time to calculate the number of files. The extra time taken by Scheduler to perform this function may slow down other activities on your Macintosh.*

Scheduling Events When the Size of a Folder Reaches a Limit

This scheduling option allows you to have your application or document opened whenever the size of a folder rises above or falls below a limit.

Select this option by choosing "When A Folder's Size Limit Is Exceeded" from the Launch pop-up menu *(Figure D-18)*.

Figure D-18
The "When A Folder's Size Limit Is Exceeded" settings



The "When A Folder's Size Limit Is Exceeded" scheduling option provides the following parameter:

Folder: The folder parameter allows you to select the folder which triggers your application or document. Click the Set Folder button to set or change the folder.

Folder Size:

This parameter allows you to define a limit for the size of the folder. The pop-up menu lets you choose the type of limit on free space, either Greater Than or Less Than.

Once you have selected the type of limit, you can enter the size limit into the text box.

Use the Include Enclosed Folders checkbox to indicate whether or not you want Scheduler to include the files stored in enclosed folders when it calculates the size of the folder.

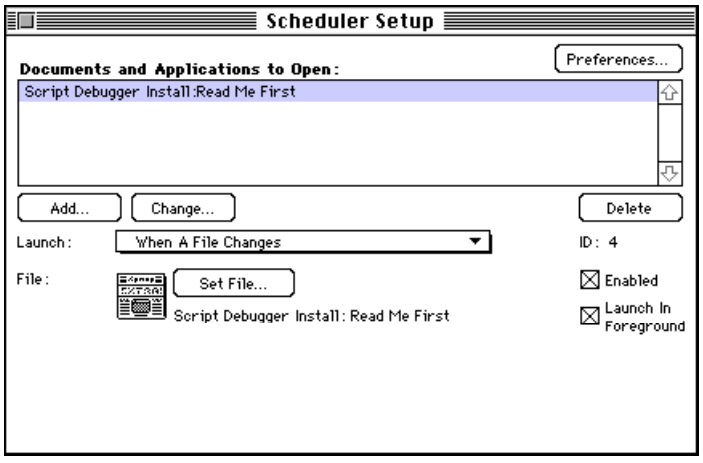
CAUTION: *Use the Include Enclosed Folders feature with care. When this feature is checked, Scheduler must check every file in all the enclosed folders to calculate the total size of the folder. If you have a deep folder structure containing many files, Scheduler requires extra time to calculate the total folder size. The extra time taken by Scheduler to perform this function may slow down other activities on your Macintosh.*

Scheduling Events When Files Are Changed

This scheduling option allows you to have your application or document opened whenever a particular file is changed. Scheduler recognizes changed files by periodically checking their modification dates.

Select this option by choosing "When A File Changes" from the Launch pop-up menu *(Figure D-19)*.

Figure D-19
The "When A File Changes" settings



The "When A Folder Changes" scheduling option provides the following parameter:

File: The file parameter allows you to select the file which triggers your application or document. Click the Set File button to set or change the file.

Volume-Based Scheduling

Scheduler provides a series of options for launching applications and documents whenever volumes are mounted and dismounted, or when the free space available on a particular volume reaches a limit.

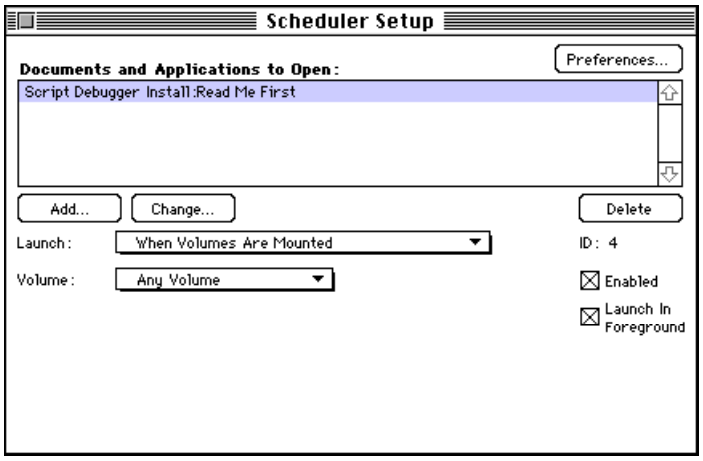
The following sub-sections describe each of Scheduler's volume-based options in detail.

Scheduling Events When Volumes Are Mounted or Dismounted

This scheduling option allows you to have your application or document opened whenever a volume is mounted or dismounted.

Select this option by choosing "When Volumes Are Mounted" or "When Volumes Are Dismounted" from the Launch pop-up menu *(Figure D-20)*.

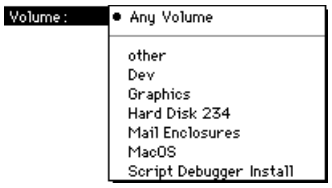
Figure D-20
The "When Volumes Are Mounted" settings



The “When Volumes Are Mounted” and “When Volumes Are Dismounted” scheduling options provide the following parameter:

Volume: This parameter lets you specify the name of the volume which triggers the opening of your document or application *(Figure D-21)*.

Figure D-21
The Volume pop-up menu



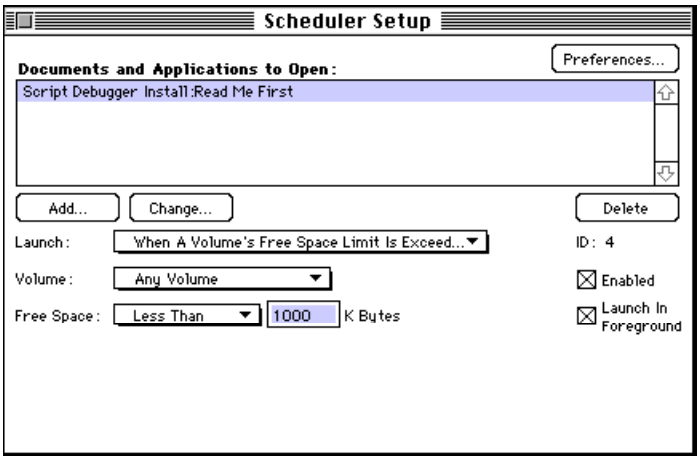
The Volume pop-up menu lists all mounted volumes and contains an option titled *Any Volume*. Selecting *Any Volume* causes your document or application to be launched whenever any volume is mounted or dismounted, depending on the option you have chosen in the Launch pop-up menu.

Scheduling Events When Free Space on a Volume Reaches a Limit

This scheduling option allows you to have your application or document opened whenever the free space available on a volume reaches a limit. You can use this option to launch applications or documents when the space available on a volume drops below a certain value or when the space available rises above a limit.

Select this option by choosing "When A Volume's Free Space Limit Is Exceeded" from the Launch pop-up menu *(Figure D-22)*.

Figure D-22
The "When A Volume's Free Space Limit Is Exceeded" settings



The "When A Volume's Free Space Limit Is Exceeded" scheduling option provides the following parameter:

Volume: This parameter lets you specify the name of the volume which triggers the opening of your document or application *(Figure D-21)*.

The Volume pop-up menu lists all mounted volumes and contains an option titled *Any Volume*. If you select *Any Volume*, your document or application is launched whenever any volume is mounted or dismounted, depending on the option you have chosen in the Launch pop-up menu.

Free Space:

This parameter allows you to define the free space limit for the volume. The pop-up menu lets you choose the type of limit on free space, either Greater Than or Less Than.

Once you have selected the type of limit, you can enter the amount of free space into the text box.

Scheduling Features for PowerBook Users

Scheduler provides a series of scheduling options designed specifically for PowerBook users. These scheduling options allow PowerBook users to perform operations in response to events which can only occur on battery-powered Macintoshes.

NOTE: *These scheduling options are only available on PowerBook computers. Scheduler Setup does not allow access to these options on desktop computer systems.*

Scheduling Events When the Power Adapter is Plugged in or Unplugged

This scheduling option causes your document or application to be opened or launched when you plug the power adapter into the PowerBook or when you unplug the power adapter *(Figure D-23)*.

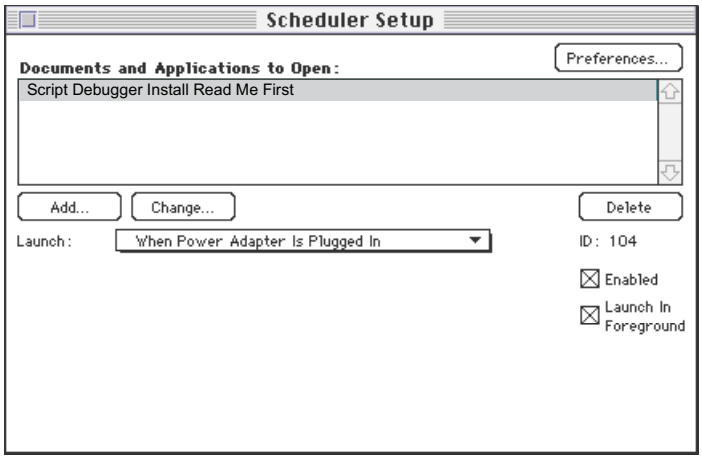


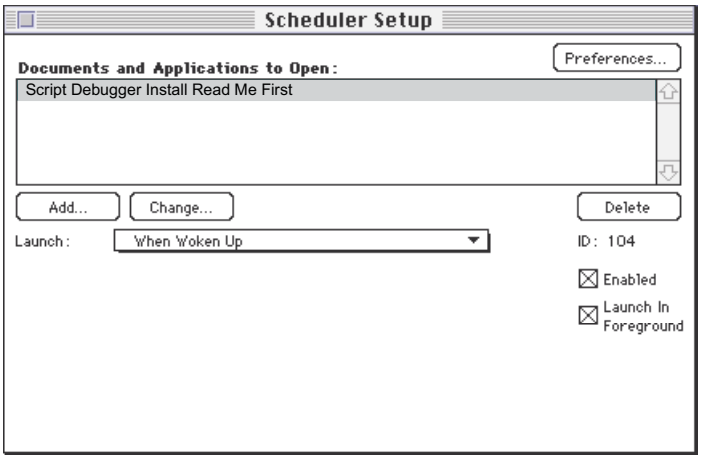
Figure D-23
The "When Power Adapter Is Plugged In" setting

This scheduling option is useful for configuring your PowerBook based on the level of power available. For instance, when the power adapter is unplugged, you may want to terminate power-hungry applications—applications which use the serial ports or modems.

Scheduling Events When the PowerBook Wakes Up

This scheduling option causes your document or application to be opened or launched when you wake up your PowerBook *(Figure D-24)*.

Figure D-24
The "When Woken Up" settings



This scheduling option is useful for restoring services which may have been interrupted while your PowerBook was sleeping.

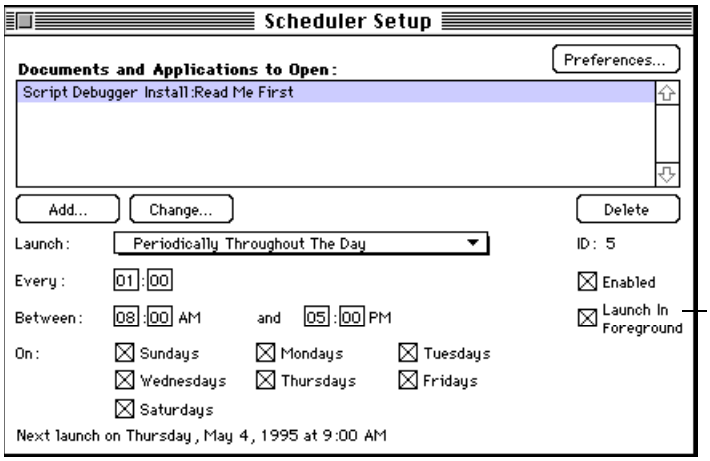
Launching Applications in the Background

Scheduler allows you to select how your applications are launched and how your documents are opened. Scheduler can open applications and documents in the background or in the foreground.

When documents or applications are launched in the foreground, they interrupt any work you might already be doing on your computer. When documents and applications are opened in the background, your work is not affected, except possibly for a brief slowdown of your computer.

By default, Scheduler launches applications and opens documents in the foreground. If you want to have your document or application opened in the background, uncheck the Launch in Foreground checkbox *(Figure D-25)*.

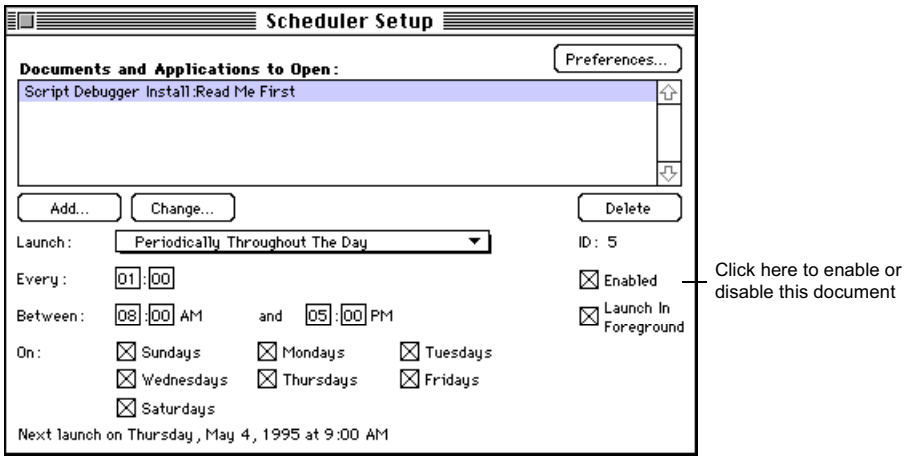
Figure D-25
The Launch in Foreground checkbox



Disabling Scheduled Events

There may be times when you want to temporarily prevent one of your applications or documents from being opened. You can do this easily using the Enabled checkbox in the Scheduler Setup window. Normally, this checkbox is checked, indicating that the document or application will be opened. When it is unchecked, Scheduler ignores the document or application *(Figure D-26)*.

Figure D-26
The Enabled checkbox

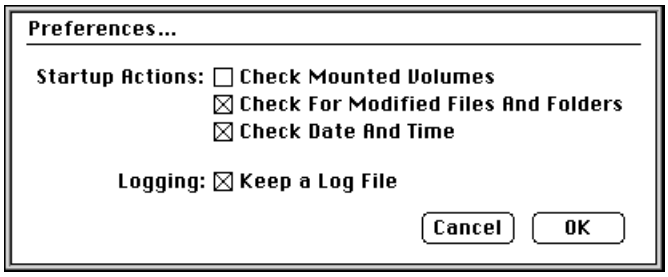


Modifying Scheduler Preferences

Scheduler provides a series of preferences settings which allow you to configure the way Scheduler operates.

To modify the Scheduler preferences, click the Preferences button in the Scheduler Setup window *(Figure D-27)*.

Figure D-27
The Scheduler Setup Preferences dialog



Check Mounted Volumes

This checkbox allows you to control how Scheduler responds to mounted volumes during system start-up.

When checked, this setting causes Scheduler to check mounted volumes. When not checked, Scheduler only begins to watch for mounted volumes after system start-up.

Check For Modified Files And Folders

This checkbox allows you to control how Scheduler responds to files and folders on file servers which have changed when your Macintosh is not running. When checked, this setting causes Scheduler to check for modified files and folders which may have changed. When not checked, Scheduler ignores changed files and folders and begins to watch for file and folder changes following system start-up.

Check Date And Time

This checkbox allows you to control how Scheduler responds when a scheduled application or document could not be launched because your Macintosh was not operating.

When checked, this setting causes Scheduler to launch documents (at start-up time) which would have been launched if your Macintosh had been operating.

Keep a Log File

This checkbox allows you to start or stop Scheduler logging.

When checked, this setting causes Scheduler to keep a record of all events in the Scheduler Log file located in the Preferences folder within your System Folder. When not checked, no logging is performed.

APPENDIX E

Script Debugger and Projector

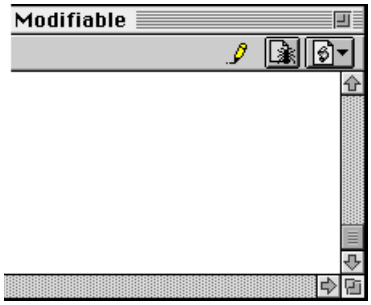
Projector is an integrated collection of MPW tools and scripts for managing source files. Projector regulates users' access to files since they must check files in and out of the Projector database. It also maintains revisions and comments to the source files. If you have a large number of scripts and you use Projector to manage them, Script Debugger can be used in conjunction with it. 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.

NOTE: *In this discussion, we do 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 Symantec C++, CodeWarrior, or MPW.*

Files Checked out for 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 E-1)*.

Figure E-1
The Pencil icon indicates the file is modifiable

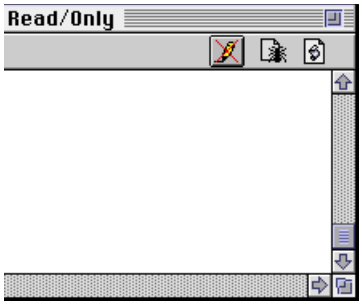


After you have made modifications to the script, you will check it back into Projector's database.

Files Checked as 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 E-2)*.

Figure E-2
A crossed Pencil icon indicates the file is read-only



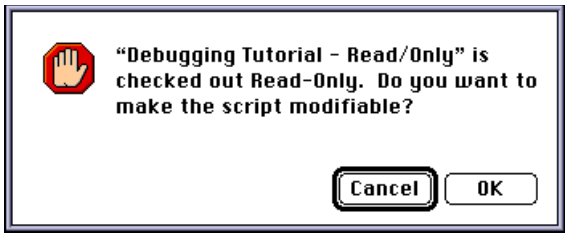
In this case, Script Debugger treats the file as though it were locked. You cannot make changes to the contents of the file. For instance, the Save menu is disabled to prevent you from accidentally making any changes to the file.

If you decide to make changes to the file, you can change its state to Modifiable Read-Only. A button and menu item are provided for these circumstances. You can click on the Pencil button in the script header or select Modify read-only from the File menu *(Figure E-3)* to make the Read-Only file modifiable.

Figure E-3
The Modify Read-Only menu item

Clicking on the button or selecting the Modify Read-Only command displays a dialog box to confirm that you want to change the state of the file *(Figure E-4)*.

Figure E-4
Confirming a change to modifiable read-only



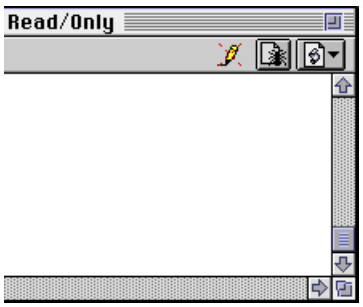
Modifiable read-only files are a feature of Projector which allow you to make changes to otherwise read-only files without checking them out of the Projector database. You do have to check the changes back into Projector at a later time.

Files Checked Out as Modifiable Read-Only

If the script file has been checked out as Modifiable Read-Only, the Pencil icon in the script header has a dimmed X through it *(Figure E-5)*.

Figure E-5

A Pencil icon with a dimmed cross indicates it is Modifiable Read-Only



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

APPENDIX F

More Information about AppleScript

Some of the best information about AppleScript can be found in AppleScript manuals, however, not everyone has them or knows how they can be obtained. This section of the manual provides pointers to Apple's manuals, other books about AppleScript, and other sources of information.

Apple's Scripting Guides

If you purchase the AppleScript Scripter's Kit (M1730LL/B, $139.00, available from APDA or Apple) or the Developer's Kit (R0175Z/C, $199.00, available from APDA), you receive several AppleScript books: *Getting Started*, *Scripting Language Guide*, *Scripting Additions,* and *Building Interfaces*. Some of these books are also available directly from Addison-Wesley (the *Getting Started* and *Building Interfaces* manuals can only be purchased with the two kits). These manuals can be purchased separately:

AppleScript Language Guide, Addison-Wesley, 1993 (ISBN 0-201-40735-3). $29.95

AppleScript Additions Guide, Addison-Wesley, 1993 (ISBN 0-201-40736-1). $18.95

Apple also has a book which documents the standard suites of Apple events. You can purchase a printed copy of the *Apple Event Registry* from APDA, and you can find electronic versions on the Developer's Kit CD and the *develop* 20 Bookmark CD.

Apple Event Registry: Standard Suites, Apple Computer, 1992. APDA, R0130LL/A, $85.00

Apple's Finder Scripting Kit

Since AppleScript was first released, Apple has released the Scriptable Finder. First made available in the Finder Scripting Kit (R0573Z/A, $25.00, available from APDA), it includes the Finder software and a manual. The Scriptable Finder was incorporated into System 7.5, but the manual was not shipped with the System Software. It is available separately: *AppleScript Finder Guide*, Addison-Wesley, 1994, (ISBN 0-201-40910-0). $19.95

Third-Party Books

In addition to Apple's AppleScript Guides, there are currently four third-party books:

Goodman, Danny. *The Complete AppleScript Handbook*. 2nd ed. Random House, 1995 (ISBN 0-679-75806-2). $35.00

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

Articles About AppleScript

The following is a list of articles about AppleScript published in the quarterly Apple Technical Journal, *develop* . Each issue comes with a bookmark CD which contains all of the *develop* magazines in electronic form along with other documentation, system software components, and utilities. Yearly subscriptions can be obtained for $30.00 from Apple Computer by calling 800/877-5548 (or 815/734-1116 outside the U.S.) or by sending e-mail to dev.subs@applelink.apple.com.

Anderson, Greg. "Scripting the Finder from Your Application," *develop* 20, December 1994, pp. 65-78.

Berdahl, Eric M. "Better Apple Event Coding Through Objects," *develop* 12, December 1992, pp. 58-83.

Clark, Richard. "Apple Event Objects and You," *develop* 10, May 1992, pp. 8-32.

Smith, Paul G. "Programming for Flexibility: The Open Scripting Architecture," *develop* 18, June 1994, pp. 26-40.

Smith, Paul G. "Implementing Inheritance in Scripts," *develop* 19, September 1994, pp. 89-99.

MacScripting Mailing List

If you have an electronic mail account, you may be interested in the MacScripting mailing list. A mailing list is a group of people interested in a particular topic whose e-mail addresses are kept in a list maintained by a mailing list manager running on a workstation or mainframe. The mailing list allows subscribers to ask questions and exchange information.

NOTE: *If your e-mail account is on a commercial service, you should be warned that some services charge for messages that pass through a gateway to the Internet. Be sure to check with your service provider to see if you are charged for individual mail messages to the Internet.*

The MacScripting mailing list is devoted to the discussion of AppleScript, Frontier, and other OSA scripting languages. You can subscribe to the mailing list 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 sent information about changing your mail options and unsubscribing.

Another informal source of information is the Frequently Asked Questions file generated by the mailing list. The FAQ contains the most commonly asked questions on the mailing list, and it exists to reduce the amount of mail traffic. You can obtain a copy of the MacScripting FAQ via anonymous ftp from gaea.kgs.ukans.edu if you have ftp access to the Internet. You may also find it in the AppleScript sections and forums on some of the commercial services.

Index

F

G

H

I

K

L

M

N

T

U

V

W