OneClick Scripting Guide

WestCode Software, Inc. · 15050 Avenue of Science, Suite 112 · San Diego, CA 92128
619–487–9200 · fax 619–487–9255 · tech support 619–487–9233
www.westcodesoft.com · e-mail westcode@westcodesoft.com

The OneClick Product Team

Alan Bird
Rob Renstrom
John Oberrick
Jeff Jungblut
Mark Brooks

Manual and Layout

Jeff Jungblut

Cover and Package Design

Steve Sharp, Sharp Advertising & Design

Copyright

Trademarks

Second Printing, February 1997
Printed in the United States of America.

Contents

Chapter One

Introduction

This manual shows you how to write and edit scripts using OneClick's scripting language, EasyScript. Before you start writing your own scripts, you should be familiar with how to use basic OneClick features, such as buttons and palettes, and the button and palette options in the OneClick Editor window.

If you haven't already done so, read Chapter 3, "Getting Started with OneClick" and Chapter 4, "Using The OneClick Editor" in the *OneClick User's Guide*.

Why script?

It's not necessary to learn how to write scripts to make use of OneClick. The pre-designed buttons included with OneClick may already meet your needs. However, if you're an advanced user, a system integrator, or simply curious, we recommend that you read this manual. You'll learn how scripting can extend OneClick's capabilities for virtually any need.

About this manual

This manual is divided into six parts. We recommend that you read at least Chapter 2 to get started and Chapter 4 after you're comfortable using the Script Editor and reading scripts. You can read the other chapters in any order.

- Chapter 2, "Scripting Tutorial," introduces basic techniques for recording and writing your own button scripts. You'll learn how to write and edit scripts using OneClick's Script Editor, and you'll begin learning the EasyScript language.

- Chapter 3, "Using the Script Editor," describes all the options available in the Script Editor, including the script recorder, editor, compiler, and online help.
- Chapter 4, "Using EasyScript," provides a more thorough introduction to the EasyScript language. You'll learn how to use commands, functions, variables, objects, and other language elements to enhance your scripts.
- Chapter 5, "EasyScript Reference," contains detailed descriptions of all EasyScript keywords. A section for each keyword describes how and when to use the keyword, the keyword's syntax and parameters, and sample scripts that use the keyword.
- Appendix A, "EasyScript Summary," contains a brief summary of all the EasyScript keywords. Use this chapter as a quick reference when you want to find out what a keyword does.
- Appendix B, "AppleScript Information," shows how you can integrate AppleScript scripts with OneClick scripts and provides pointers to other sources of information for AppleScript users.

Technical information for Macintosh developers

The OneClick manuals do not include technical information regarding the development of OneClick extensions (external script keywords and plug-in button border styles). The use of this feature requires some knowledge of Macintosh programming. If you're a Macintosh developer and you're interested in adding new keywords to the EasyScript language, or you want to develop new button border styles, contact WestCode Software for more information about our OneClick Extension Developer's Toolkit.

Chapter Two

Scripting Tutorial

This chapter contains a few short tutorials that show you how to get up and running with the Script Editor and how to write some basic scripts.

These tutorials assume that you've read Chapter 3, "Getting Started with OneClick" in the *OneClick User's Guide*, which explains the basics of how to work with palettes, buttons, and the OneClick Editor window. This chapter builds upon the SimpleText palette you created in that chapter, so if you haven't yet created your SimpleText palette, you'll need to perform the exercises in *Making a custom palette* in Chapter 3 of the *OneClick User's Guide* first. Then return here when you're ready to uncover the scripts on the SimpleText palette and see how they work.

We recommend that you go through these exercises in the order presented. You'll learn how to do the following:

- View a button's script
- Make simple changes to a script
- Correct errors in a script
- Get help for script keywords
- Insert parameters for script keywords
- Copy a script from one button to another

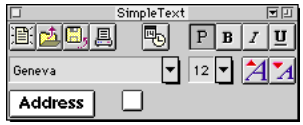
Along the way, you'll also learn a bit of the EasyScript language, such as how to use simple EasyScript commands and functions to type text, press command keys, and choose menu items.

Viewing a button's script

In the first part of this chapter, we'll take a look at the scripts you recorded earlier and make some changes to them.

1 If SimpleText is not already open, double-click the **SimpleText** icon to open it.

Your palette appears when SimpleText is open.



2 If there are no document windows open, click the  button to open a new untitled window.

3 Click the **Address** button once to type your address in the window.

The button you recorded to type your address probably works great, but what if you had made a typing error while recording? When you click the button to play back the recorded script, the button would faithfully reproduce the mistake.

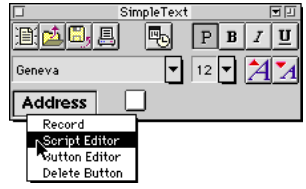
The script recorder accurately records all your actions—including any mistakes, such as typographical errors, misspellings, clicking the wrong button, or choosing the wrong menu item. To correct these kinds of errors, you can start over and record the button's script again, but that's probably not the most practical way to do it—especially if you were recording a long sequence of actions.

An easier way to correct errors made while recording is to edit the script and fix the mistakes directly. The Script Editor shows all your recorded actions in easy-to-understand *statements*, so you can quickly locate the error and fix it.

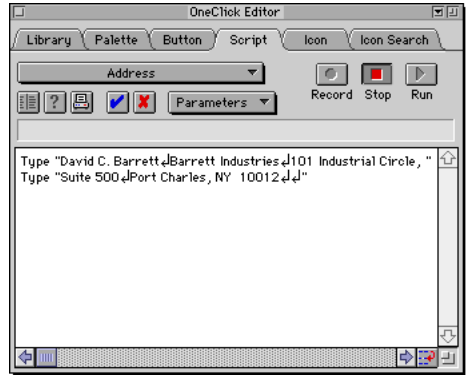
4 Do one of the following:

- If the OneClick Editor is closed, hold down the Command and Option keys, then click the **Address** button and choose **Script Editor** from the pop-up menu.

- If the OneClick Editor is open, hold down the Command and Option keys, then click the **Address** button. (No pop-up menu appears if the OneClick Editor is already open.)



The Script Editor appears. The script for the selected button (the Address button) appears in the text box at the bottom of the window.



5 If lines in the script run past the right edge of the window, drag the size box in the bottom-right corner to resize the window so you can see the rest of the script.

Checking out the script statements

Because you most likely typed your own address instead of David Barrett's address, you see some different information in the script window than the address shown above. Your script, however, should contain one or more lines beginning with the word Type, followed by the information you typed.

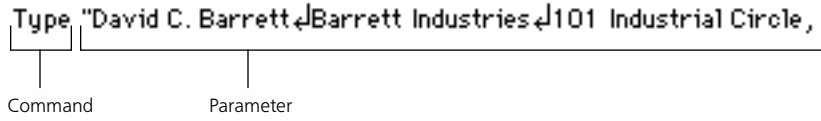
The ↵ symbols you see in the script are Return characters. When you press the Return key while recording, the script recorder represents the Return keystroke as the ↵ symbol in the script.

A script is like a mini-program: *statements* in the script tell OneClick what actions to perform when you click the button. In this script, a Type statement tells OneClick to type something as if you typed it yourself using the keyboard.

Your script may appear as one or more statements. If you type a lot of information while recording, OneClick divides the information into multiple Type statements. OneClick plays back the statements in the order in which they appear in the script.

Understanding commands and parameters

Each statement in the script contains two parts: a *command* and a *parameter*.



The *command* part of the statement, Type, tells OneClick what to do—that is, type text or other keystrokes. The Type command needs to know what text or keystrokes to type.

A *parameter* is a piece of information that you give to a command when the command needs more information to work with. The information between quotation marks (") is the text that OneClick types—the parameter for the Type command. A space separates the command name and the parameter so they don't run together. In programmer-speak, giving information (such as the quoted text) to a command is called "passing a parameter."

Not all commands require a parameter. The Beep command, for example, plays the Macintosh system beep. The command doesn't need additional information to play the beep sound.

Making simple changes to a script

Now that you've seen the anatomy of a script, you're ready to make some changes.

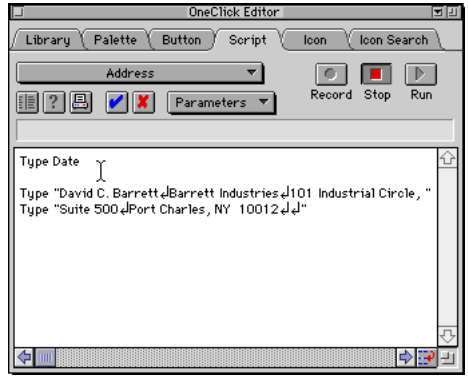
1. Click before the first **Type** statement in the script so that the cursor appears at the beginning of the line.
2. Press Return to insert a blank like.

3 Press the Up Arrow key to move the cursor back up to the first line.

4 Type the following statement in the script, then press Return to insert another blank line:

```
Type Date
```

The script window should now look something like this:



You can insert blank lines in a script to separate groups of statements, making the script easier to read. Blank lines aren't required.

5 Click the **Run** button in the upper-right corner of the Script Editor.

Clicking the Run button plays back the script. It's the same as clicking the Address button when the OneClick Editor window is closed.

Correcting errors in a script

The script now types the current date, followed by your address, in the SimpleText window. However, the date and the first line of the address all appear on the same line. You'll need to edit the script so that it presses the Return key a few times to put some blank lines between the date and the address.

1 Click in the Script Editor window and move the cursor to the blank line following the Type Date statement.

2 Type the command **Type**, followed by a space and a quotation mark (").

```
Type "
```

3 Hold down the Option key and press Return three times.

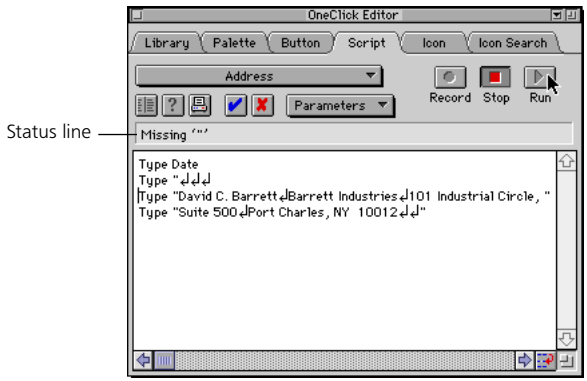
```
Type "↵↵↵
```

Three ↵ symbols appear in the script. (Remember, the ↵ symbol in a Type statement tells OneClick to press the Return key.)

Note Holding down the Option key lets you type other special characters in a script, not just Return. You could press Option-Delete and Option-Left Arrow to insert the symbols for the Delete and Left Arrow keys, for example.

4 Click the **Run** button.

The script won't run. Instead, the Script Editor beeps and displays an error message on the status line.



The message, Missing '"', means that there is a quotation mark (") missing somewhere in the script. The cursor blinks at the beginning of line below the new line you just typed, indicating that the quotation mark is probably missing on the previous line.

The parameter for the Type command, the three ↵ symbols, must be enclosed in quotation marks. The quotation marks tell OneClick where the parameter's text begins and ends.

OneClick won't run the script until all the errors in the script are corrected.

5 Move the cursor to the end of the line you typed in step 3, then type a quotation mark (").

6 Click the **Run** button.

The script now types the current date, followed by two blank lines and your address. The message "No errors" appears on the status line to indicate that no errors were found.

Note If you want to check a script for errors without running it, click the ☑ button in the Script Editor. The status line shows "No errors" if the script is OK, otherwise it displays an error message and highlights the error in the script.

There are additional error messages that can appear on the status line. Error messages are discussed at the end of Chapter 3, "Using the Script Editor."

OneClick always checks a changed script for errors when you select a different button, switch to another editor, or close the Script Editor window. You must correct any errors in the script before leaving the Script Editor, or else discard the changes you made to the script.

Understanding functions

The word **Date** you used in the previous exercise is called a *function*. A function gives a value, such as the current date, to a command (such as the Type command) or to another function. The Type command uses the Date function as its parameter. It takes the value given by the Date function and types it in the SimpleText window. The Date function itself doesn't do the actual typing, it just returns its value to the Type command.

The Date function you used earlier did not include a parameter. You can pass a number as a parameter to the Date function, however, that tells Date the format in which you want the date returned. For example, the statement "Date 3" returns the

date in the short format, 7/23/95; the statement "Date 12" returns the date in the long format: Sunday, July 23, 1995. Without a parameter, the Date function returns the date in the default short format.

Note The names of commands, functions, and all the other words that OneClick understands are called *keywords*. A keyword is simply a single word in the EasyScript language. Type and Date are both examples of keywords.

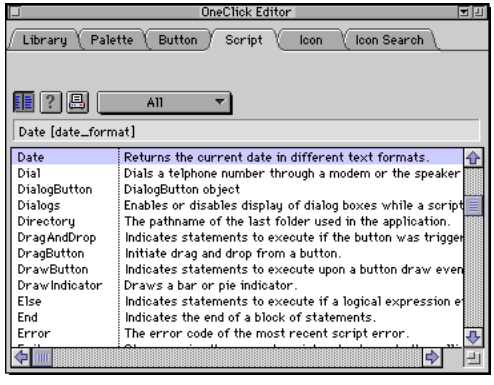
Getting help for script keywords

To find out how to tell the Date function to return the date in a different format, you need to know what numbers the Date function can use in its parameter. You can look up the information in Chapter 5, "EasyScript Reference" or you can use the online help available in the Script Editor.

1 Click the  button in the Script Editor.

A list of keywords appears in the window. This list shows all the keywords in the EasyScript language. A one-line description appears next to each keyword.

2 Type the letter "**D**" to scroll down to the keywords beginning with D.



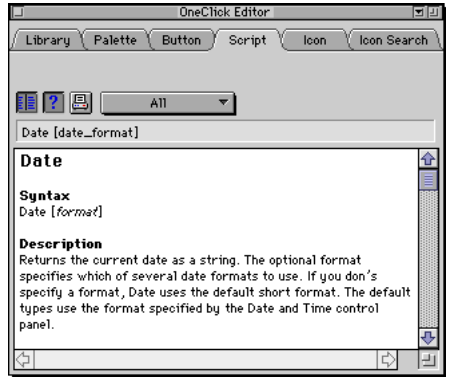
The status line shows the Date function, followed by the words "date_format" in square brackets. Date_format is the name of the Date function's parameter; the name tells you that the parameter you pass to the Date function indicates the

date format to use. The square brackets around the parameter name mean that the parameter is optional—you don't need to supply a parameter to use the Date function.

The keyword list gives brief descriptions of all the keywords, but it doesn't tell you what values you can use for the format parameter.

3 Click the ? button, or double-click the **Date** keyword in the list box.

Either method displays additional help for the Date function.



4 Scroll down past the Description paragraph in the help text to see a list of values for the format parameter.

As you can see in the Script Editor window on your screen, there are a variety of date formats you can use, and each format is represented by a number. Using a different number with the Date function causes the Date function to return the date in a different format. You can even add numbers together to create more formats, such as a short format with a dash separator (7-23-95).

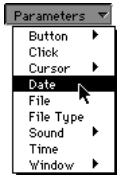
Looking up the different format numbers for the Date function may seem tedious if you want to use the function often. Fortunately, the Script Editor provides an easier way to figure out the parameter to use for the Date function (and other keywords). Also, the Date function is an exception; parameters for other commands and functions are typically much easier to remember and use.

Inserting parameters for script keywords

For the handful of commands and functions that have difficult-to-remember parameters, such as Date, the Script Editor provides an intuitive method for specifying the parameters.

1 Click the  button in the Script Editor to close the help window and return to the script.

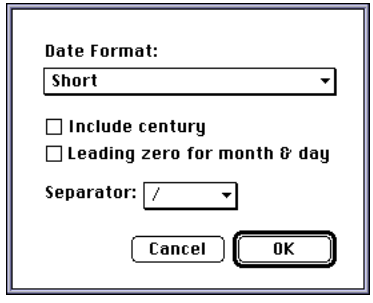
2 Move the cursor to the end of the **Type Date** statement in your script, then press the space bar to insert a space.

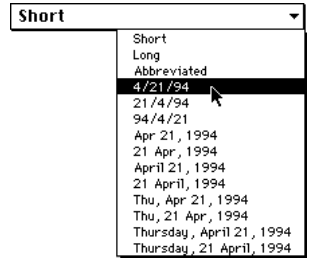


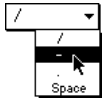
3 Choose **Date** from the Parameters pop-up menu in the Script Editor.

The Date Format dialog box appears.

4 Choose **4/21/94** from the Date Format pop-up menu.



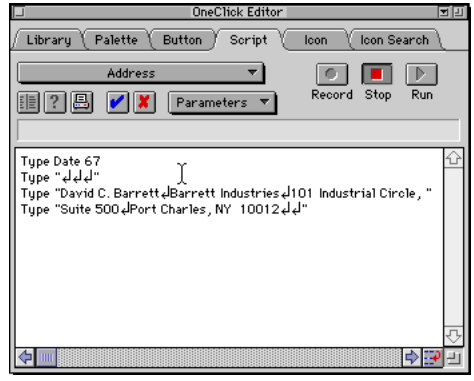




5 Choose the dash (–) from the Separator pop-up menu.

6 Click **OK**.

OneClick inserts the number 67 after the Date keyword in the script.



The number 67 is the format number that corresponds to the options you chose in the dialog box— the short M/D/YY format, using a dash (instead of a slash) for the separator character.

Note that the number 67 is *not* enclosed in quotation marks. Parameters that contain text (such as the parameters in the Type statements) must be enclosed in quotation marks, but numeric parameters are not. If the number 67 were inside quotation marks, then OneClick might try to interpret the value as the characters "6" and "7", not the number 67.

7 Click the **Run** button to run the script.

OneClick types the date in the format M-D-YY, followed by two blank lines and your address.

Other options in the Parameters pop-up menu

Other items in the Parameters pop-up menu let you insert parameters for different keywords. For example, if you want to play a sound using the Sound command, you can either type the sound's name yourself, or you can use the Sound submenu in the Parameters menu. The Sound submenu lets you pick a sound from all the sounds available in your System file. When you choose a sound, OneClick inserts the sound's name in the script, so you don't have to type the name yourself or remember its exact spelling.

The Time option in the Parameters menu is similar to the Date option: it displays a dialog box that lets you choose options for the parameter used by the Time function.

By the way, the Time function works just like the Date function—it returns the current time in a variety of formats, or a default format if you don't specify a parameter.

For information about the other options in the Parameters menu, see Chapter 3, "Using the Script Editor."

Copying a script from one button to another

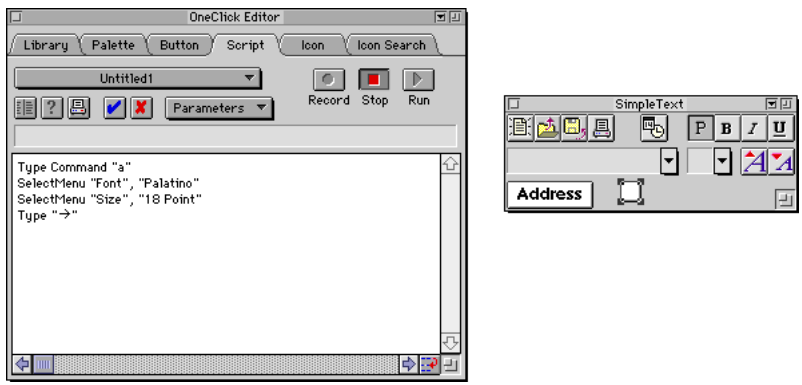
You can use copy and paste to edit statements in the Script Editor. In this exercise, you'll combine the two scripts you've created into a single script for the Address button. When you run the combined script, OneClick will do the following:

- Type the current date in the SimpleText window, followed by your address
- Select all the text in the window
- Change the font and size
- Move the cursor to the end of the document

First we'll take a look at the script that chooses commands from the Font and Size menus.

1. Open the OneClick Editor if it isn't already open.
2. Click the second button you created on the SimpleText palette (the button that selects all the text and changes the font and size).

   The Script Editor shows the script for the selected button.



Note If an error message appears when you click the button, it means that the script for the Address button (the one you were just editing) contains an error. Click **Edit** to return to the Script Editor and correct the error, then try again.

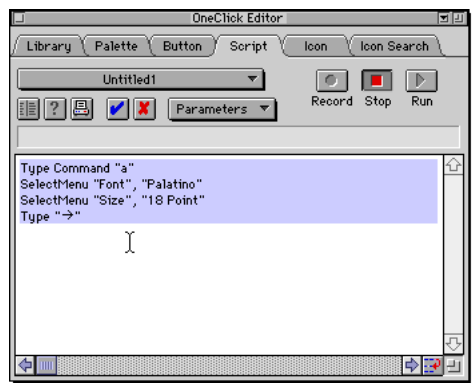
There are two new keywords in this script: the **Command** keyword in the first line, and the **SelectMenu** keyword in the second and third lines.

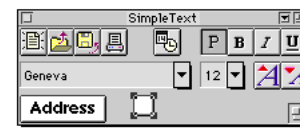
The Command keyword in the Type statement is called a *modifier*. The modifier changes the way the Type statement works. In this script, **Type Command "a"** means to hold down the Command key while typing the letter "a". When you run the script, OneClick presses Command-A to select all the text in the SimpleText window, instead of simply typing the letter "a" as text.

The SelectMenu command chooses an item from a pull-down menu in the menu bar. SelectMenu needs two parameters: the name of the menu in the menu bar and the name of the menu item to choose. In this script, the second statement chooses Palatino from the Font menu and 18 Point from the Size menu. The menu and menu item names, like other text parameters, are enclosed in quotation marks.

3 Use the mouse to select all the lines in the script.

Shortcut To select a whole line in the script, triple-click the line. To select the whole script, quadruple-click the script or press Command-A.





4 Press Command-C to copy the script to the Clipboard.

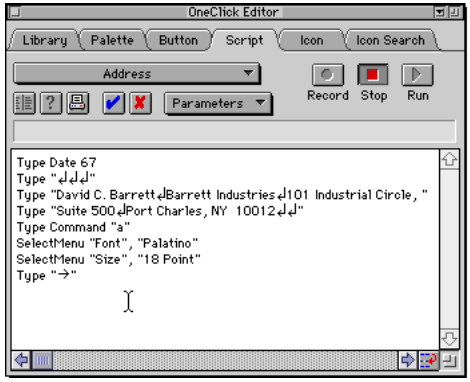
You must use the keyboard command, not the Copy command in the Edit menu. The command in the Edit menu copies text in the SimpleText window, not in the Script Editor window.

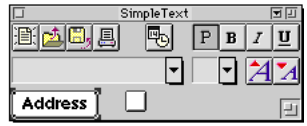
5 Click the Address button to display the button's script in the Script Editor.

6 In the Script Editor, move the cursor down to the blank line following the last Type statement.

7 Press Command-V to paste the script from the Clipboard into the script for the Address button.

Your script should now look something like this:





8 Choose **New** from the File menu to open another untitled SimpleText window.

9 Click the **Run** button in the Script Editor.

Your script now types the date, two blank lines and your address, then changes all the text to Palatino 18-point and moves the cursor to the end of the document.

Using the Check and Uncheck modifiers

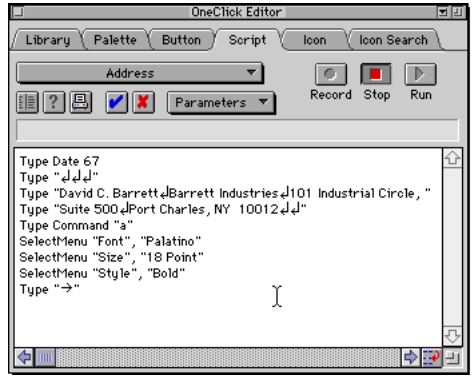
Now you'll add a new SelectMenu statement to the script. The statement will choose the Bold item from the Style menu.

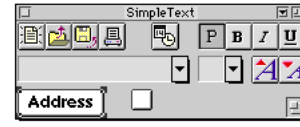
1 Move the cursor to the beginning of the last line in the script (the statement that types the Right Arrow key), then press Return to insert a blank line.

2 Type the following statement on the blank line before the last Type statement:

```
SelectMenu "Style", "Bold"
```

Your script should now look similar to this:





3 Close the OneClick Editor window.

If an error message appears, click **Edit** to return to the Script Editor. Make sure you typed the SelectMenu statement exactly as it appears above, then try again.

4 Click the Address button three or four times to run the script.

The text in the SimpleText window changes to Bold the first time you click the button. But every other time you click the button, the text alternates between Plain and Bold.

This happens because the Bold item in the Style menu is a checkmarked item. Each time the SelectMenu command chooses the menu item, it toggles the item on and off. This is a good example of why it's sometimes necessary to edit a script after you've recorded it—if you had simply recorded the menu choice, then ran the script a few times, the SelectMenu command would always toggle the Bold item on and off each time you ran the script. In cases like this, you'll need to do a little fine-tuning to make the script work the way you want.

To make the Bold option turn on and *stay* on, you'll add a modifier to the SelectMenu statement.

5 Hold down the Command and Option keys and click the Address button, then choose **Script Editor** from the pop-up menu.

The script for the Address button appears in the Script Editor.

6 Type the word **Check** in the last SelectMenu statement so that the statement reads as follows:

```
SelectMenu Check "Style", "Bold"
```

The Check keyword is a modifier that changes the way the SelectMenu command works. Using Check in a SelectMenu statement means that SelectMenu turns on the Bold item if it wasn't already on (checked). If the Bold item is already checked, then the SelectMenu statement does *not* choose the Bold item, because doing so would turn it off.

7 Click the **Run** button in the Script Editor a few more times.

Now when you run the script, the text in the SimpleText window changes to Bold and stays that way each time you run the script.

The SelectMenu command has another modifier, **Uncheck**, that you can use to turn off a checked menu item instead of turning it on (like Check) or toggling the menu item. You don't need to use Check and Uncheck with regular menu items, just checkmarked items when you want to force the checkmark on or off.

The SelectButton command

The **SelectButton** command is a command you'll see often in recorded scripts, but we haven't actually used this command in any of the scripts in this chapter.

SelectButton clicks a button or a checkbox in a dialog box or window. The command needs one parameter, the name of the button to click. Here are some examples:

```
SelectButton "OK"
SelectButton "Cancel"
SelectButton Uncheck "Smooth Graphics"
```

The first two statements click the OK and Cancel buttons in a dialog box. The third statement clicks a checkbox named Smooth Graphics (from the Page Setup Options dialog box).

As you see in the third statement, the SelectButton command can also use the Uncheck (and Check) modifiers. When the SelectButton command clicks a checkbox, it turns the checkbox on or off like a toggle. You can use the Check or Uncheck modifier to force the checkbox either on or off, just as you do with the SelectMenu command and checked menu items.

Displaying information in message boxes

The scripts you record and write aren't limited to choosing menu items, clicking buttons, pressing command keys, and typing text. Many other keywords let your scripts make decisions based on certain conditions, get input while the script runs, display information in message boxes, display pop-up menus, and more. Here's an example of one of the commands.

1. Close the OneClick Editor window if it's open.
2. Command-Option-click on an empty space on the SimpleText palette (not on a button), then choose Script Editor from the pop-up menu.

   When you click on the palette's background and choose Script Editor, OneClick creates a new button and opens the Script Editor for the new button. It doesn't start recording.
3. Type the following line in the script window:

   ```
   Message "Hello, World!"
   ```

   The **Message** keyword is a command that shows a message box on the screen. When you run the script, the box contains the message "Hello, World!" and an OK button.
4. Click the **Run** button in the Script Editor.

   A message box appears.

5 Click **OK** to close the message box.

Where to go from here

You've now learned the basic principles of scripting with OneClick. While working on your Address button, you learned how to do the following:

- Access the Script Editor and view a button's script
- Edit a script
- Get help for script commands
- Use the Parameters menu
- Run a script
- Correct mistakes

You have also learned a bit about the EasyScript language, such as how to use commands, functions, parameters, and modifiers.

There are many more features available in the EasyScript language than the few that you worked with in this chapter. Also, we haven't covered every single option available in the Script Editor. The following chapters cover the Script Editor and the EasyScript language in greater detail.

Chapter Three

Using the Script Editor

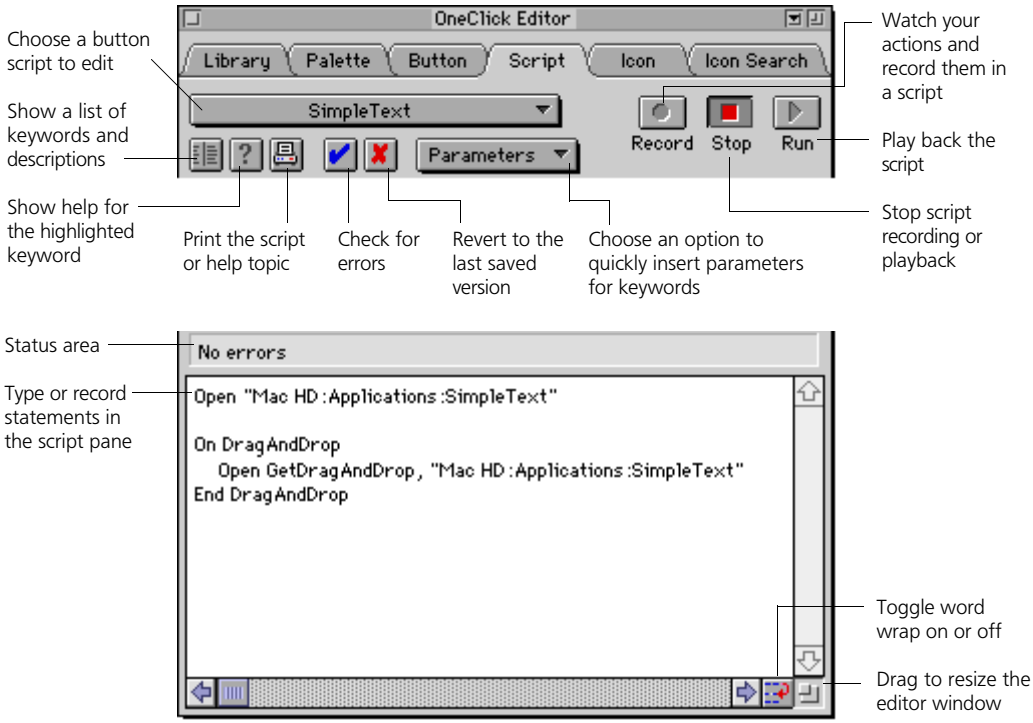
This chapter describes all the features of the Script Editor. The chapter covers the following topics:

- Accessing the Script Editor
- Recording a script
- Typing and editing in the script pane
- Checking a script for errors
- Running a script
- Printing scripts
- Getting help for script keywords
- Inserting parameters for script keywords
- Script compiler error messages

About the Script Editor

The Script Editor lets you record, write, and edit scripts for buttons. It's the one editor you'll probably use most often as you create your own custom buttons and palettes.

The Script Editor also provides online help for all the keywords in the EasyScript language.



You can use the Script Editor to view and make changes to the scripts for any of OneClick's pre-designed buttons. Using the Script Editor is a good way to find out how the pre-designed buttons work. Because the buttons all perform their tasks by running EasyScript scripts, you'll discover some valuable scripting techniques in the pre-designed buttons that you can copy and use in your own scripts.

Accessing the Script Editor

There are several ways to open the Script Editor and view a button's script.

▶ **To display a button's script**

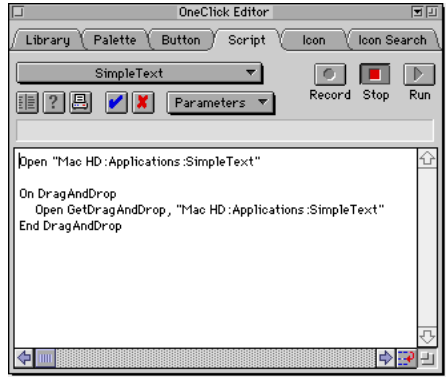
1 Choose **OneClick Editor** from the OneClick menu.

The OneClick Editor window appears.

2 On any OneClick palette, click the button whose script you want to view or edit.

3 Click the **Script** tab in the OneClick Editor window.

The selected button's script appears in the Script Editor.



Because writing a button script is usually an interactive process (record, edit, test, edit, test, and so on), OneClick provides several shortcuts you can use to access the Script Editor.

To	Do this
Create a new, blank button and edit its script	Command-Option-click a palette where you want the new button to appear, then choose Script Editor from the pop-up menu.
Edit a button's script when the OneClick Editor window is closed	Command-Option-click the button, then choose Script Editor from the pop-up menu.
Edit a button's script when the OneClick Editor window is open, but another editor is active	Command-Option-click the button to switch to the Script Editor.
Edit a different button's script while the Script Editor is active	Click the button or choose the button's name from the Button pop-up menu.

Recording a script

When you record a script, OneClick watches your mouse and keyboard actions and saves them as statements in the script. Recording is the best way to start writing a new script or to insert new statements in an existing script.

▶ **To record a script**

1. Place the insertion point in the script where you want the new statements to appear.
2. Click the **Record** button.

   The following indicators show that recording is in progress:

   - A microphone icon flashes in the menu bar.
   - The Record button lights up in the Script Editor.
   - The button you're recording flashes on the palette.
3. Perform the actions (clicking and typing) that you want the script to contain.

   Perform actions in an application as you normally would. A new script statement appears in the script pane each time you click or type.

   If you want to temporarily stop recording, choose **Pause Recording** from the OneClick menu. To continue recording where you left off, choose **Resume Recording** from the OneClick menu.
4. Click the **Stop** button or choose **Stop Recording** from the OneClick menu.

   Recording stops automatically if you close the OneClick Editor window while recording.

Note You cannot click buttons on OneClick palettes while recording is in progress.

The following table shows how OneClick records typical mouse and keyboard actions as script commands.

Your action	Script command
Typing text or commands	Type
Choosing an item from a menu in the menu bar	SelectMenu
Choosing an item from a pop-up menu	SelectPopUp
Clicking a button in a window or dialog box	SelectButton
Clicking in a scroll bar	Scroll
Clicking or dragging within a window	Click
Clicking or dragging outside a window	Click Global

Tips for recording a script

Usually, statements that use the SelectMenu, SelectPopUp, SelectButton, and Scroll commands perform more reliably than those that use Click commands. This is because the Click command performs just a simple click or drag on the screen at the specified coordinates; the command has no knowledge of what it is clicking at that location. Other commands are more intelligent: SelectButton, for example, clicks a named button and will work no matter where the button appears on the screen.

Follow these guidelines when recording a script to improve the script's reliability.

- Choose menu commands and type command keys where possible instead of clicking or dragging. For example, to switch to the Finder, you should choose "Finder" from the Application menu instead of clicking the desktop or a Finder window. This is because windows and items in them may not be in the exact same position each time you run the script. When recording, it's best to perform actions that you know will work the same way every time without depending on the position of items on the screen.
- When choosing a file in a directory dialog box, type the file's name to select it instead of clicking a name in the list. The file may not be in the same position in the list each time you run the script, so a Click statement may not choose the correct file.

- To select Finder icons, type the icon's name instead of clicking it. Icons may not always be in the same position.
- Take your time when recording a script to avoid making mistakes. The script recorder records any mistakes you make as well as your corrections, so it's best to go slow and be careful while recording. Of course, if you do make mistakes while recording a script, you can edit the script later to correct any errors.

Typing and editing in the script pane

The script pane works similar to other text editing programs for the Macintosh.

▶ **To type statements in the script pane**

1. Click in the script pane to place the cursor where you want your statements to appear.

   You need to click in the Script Editor to make it active before typing. Otherwise, keystrokes go to the active application or the selected palette (wherever you last clicked). The OneClick Editor window's title bar frame appears darkened when the window is active and receiving keystrokes.

2. Type a script statement.

3. Press Return to signal the end of the statement and move the cursor down to the next line.

Script statements do not automatically word-wrap when you type past the right edge of the script pane. Use the horizontal scroll bar to scroll sideways if your script statements go past the edge of the script pane. Or, resize the Script Editor by dragging the size box in the lower-right corner of the window.

You can enable automatic word wrap if you're working on a small screen and don't want to scroll back and forth to see all of a line.

▶ **To turn word wrap on or off**

- Click the  button next to the horizontal scroll bar.

Note Because EasyScript is a line-based language (meaning each statement occupies only one line), it's easier to see where one statement ends and the next statement begins if you leave word wrap turned off.

Script editing shortcuts

You can use the following shortcuts to select and edit text in the Script Editor.

To do this	Do this
Select a word	Double-click the word.
Select a line	Triple-click the line.
Select all text	Press Command-A or quadruple-click in the script.
Cut text to the clipboard	Press Command-X.
Copy text to the clipboard	Press Command-C.
Paste text from the clipboard	Press Command-V.
Undo the last typing or editing action	Press Command-Z.
Insert special characters in a script (such as Return, Delete, or arrows)	Hold down Option and type the character (Option-Return, Option-Delete, Option-Left Arrow, and so on).

Checking a script for errors

Whenever you click **Run** or close the Script Editor, OneClick first checks the script for errors. An error can occur because of a typographical mistake or a misspelling.

▶ **To check a script for errors**

■ Click the ✔ button in the Script Editor.

If any errors are present, a message describing the error appears in the status area and the location of the error appears highlighted in the script. See "Script compiler error messages" on page 40 for more information about each possible error.

You need to correct any errors in the script before you can save it and close the Script Editor.

Compiling a script

When you check a script for errors, OneClick *compiles* the script. Compiling means that OneClick translates the script from its human-readable text format into a more compact binary format. OneClick can understand and execute a compiled script much faster than a script in text format.

When OneClick compiles a script, each keyword is translated into a two-byte code; characters in literal strings and comments each take up one byte. A script statement must compile to less than 256 bytes or an error occurs. This method of compiling a script is often called *tokenizing* in other scripting or programming languages.

Automatic script formatting

You've probably noticed that when you save a script or check its syntax, OneClick reformats the script in the following ways:

- The case of any keywords you typed changes to the "proper" case (for example, "selectmenu" changes to "SelectMenu").
- The case of variable names changes to the case used in the Variable statement.
- Extra spaces between keywords, operators, and values are added or removed as necessary.
- Handlers and loops are indented with tab characters.

This reformatting occurs because OneClick *decompiles* the compiled script after the script compiles successfully. The compiled version, which does not retain any formatting, is translated back into a formatted text version that you can edit in the Script Editor. While you can control the script's content, OneClick helps improve the script's readability by controlling most of the formatting.

Saving changes to a script

Normally, you don't need to explicitly save a script after making changes to it; OneClick automatically saves the changes. You can, however, make OneClick save the changed script to its button at any time if you want.

▶ **To save a script to its button**

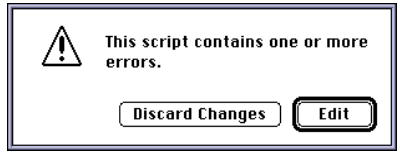
- Press Command-S.

Before saving a script, OneClick first attempts to compile the script. A message appears in the status area if the script contains errors, just as if you had clicked the ☑ button. After the script compiles without errors, OneClick saves the compiled script to the script's button.

OneClick automatically saves a changed script whenever you do any of the following:

- close the Script Editor
- switch to another button's script in the Script Editor
- switch to another editor in the OneClick Editor window
- quit the active application (if the script is for a button on an application-specific palette)
- run the script by clicking the **Run** button or pressing Command-R

If you try to close the Script Editor (or switch to another button's script) while the current script contains errors, a dialog box appears:



Click **Edit** to return to the Script Editor and fix the error, or click **Discard Changes** to throw away all changes you've made to the script since you last saved it.

Reverting to the last saved script

While editing a button's script, you can cancel any changes you've made revert to the last saved version of the script.

▶ **To revert to the last saved version of the script**

- Click the ✘ button in the Script Editor.

Running a script

You can play back the script to test it while the Script Editor remains open.

▶ **To run the current script in the Script Editor**

■ Click the **Run** button or press Command-R.

OneClick runs only the default handler in the script (usually MouseUp). To run other handlers, such as DragAndDrop or MouseDown, you must close the Script Editor and use the button as you normally would (click it or drag something to it) to trigger the appropriate handler.

▶ **To stop a running script**

■ Click the **Stop** button or press Command-period.

Printing scripts

You can print the current script, all scripts for buttons on the selected palette, or all scripts for visible (not hidden) palettes.

▶ **To print one or more scripts**

1 Click the button or press Command-P.

The bottom of the Print dialog box contains some additional options.

2 Choose one of the options on the left to specify which scripts you want to print.

To print scripts in a hidden palette, choose the palette from the OneClick menu to make it visible first. Then choose the third option.

3 To set paper size, orientation, and other printing options, click **Page Setup** and set the desired options, then click **OK**.

4 Click **Print**.

You can press Command-period to cancel printing.

Getting help for script keywords

The Script Editor provides two methods you can use to get online help for keywords: the Keyword List mode and the Detailed Help mode.

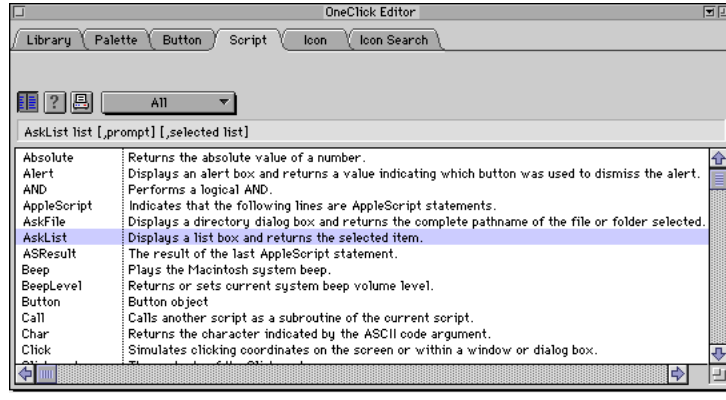
Using the Keyword List

The Keyword List mode is an online version of Appendix A, "EasyScript Summary."

▶ **To use the Keyword List**

1 Click the button or press Command-Tab in the Script Editor.

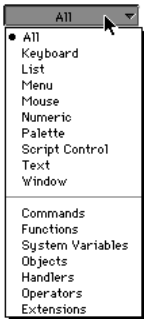
A list of all EasyScript keywords appears in place of the script pane.



2 Click a keyword in the list.

The selected keyword's name and parameters, if any, appear in the status area.

You can quickly scroll to the desired keyword by typing the first few letters of the keyword.



3 If desired, you can reduce the number of keywords displayed in the list by choosing a keyword category from the pop-up menu (shown at left).

Only keywords of the type you choose (such as Functions, Commands, or Menu-related keywords) appear in the list. For example, if you choose **Mouse** from the pop-up menu, the keyword list changes to show only the keywords that perform mouse-related activities.

4 To turn off the Keyword List mode, click the button again or press Command-Tab.

Using Detailed Help

The Detailed Help mode is an online version of Chapter 5, "EasyScript Reference."

▶ **To get detailed help for a keyword**

- If you're editing a script, double-click a keyword in the script to select it, then click the ? button or press Command-? to get help for the selected keyword.

  –Or–

- If you're viewing the Keyword List, select a keyword in the list, then do one of the following:
  - click the ? button,
  - double-click the keyword, or
  - press Return or Command-?.

Information for the selected keyword appears in the script pane. Help for each keyword includes the following:

- keyword syntax and parameters
- what the keyword does
- why and when you would use it
- sample scripts that use the keyword

You can use Command-C to copy sample script statements from the keyword help and paste the copied statements in your own script.

Printing keyword help

If your manual isn't close at hand, you can print selected topics from the detailed help.

▶ **To print help for one or more keywords**

1. While in Detailed Help mode, click the button or press Command-P.

   The bottom of the Print dialog box contains some additional options.

2. Choose one of the options on the left to specify which help topics you want to print.

   The **Current Help** option prints the help topic that's displayed in the Script Editor.

   To print help for keywords in a certain category (such as Menu- or Mouse-related keywords), choose the category from the pop-up menu, then choose the second option.

3. To set paper size, orientation, and other printing options, click **Page Setup** and set the desired options, then click **OK**.

4. Click **Print**.

You can press Command-period to cancel printing.

Note Printing help for all keywords may take a while and use up a lot of paper.

Inserting parameters for script keywords

The Parameters pop-up menu lets you insert parameters for various keywords into a script. Parameters that could otherwise be lengthy to type or tedious to figure out can be inserted in the script with just a few clicks.

▶ **To insert a parameter using the Parameters pop-up menu**

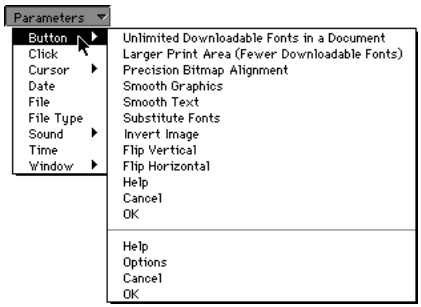
1. Place the insertion point where you want the parameter to appear in the script. (Usually you'll want the parameter to appear following its keyword and a space.)
2. Choose an option from the **Parameters** menu.
3. If the option you choose displays a dialog box, choose options in the dialog box and click **OK**.

The new parameter appears in the script at the insertion point. If the parameter is a string, then OneClick also inserts quote marks at either end of the string.

The following sections describe each option in the Parameters menu.

Button

The Button submenu contains the names of all named buttons in all on-screen dialog boxes and windows. Use the Button submenu to quickly insert the name of a button for use with the SelectButton or DialogButton keywords.



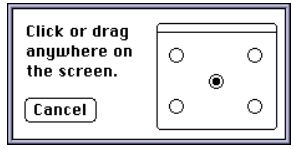
Button submenu when the Page Setup Options dialog box is open in an application.

Buttons below the divider line are in the Page Setup dialog box (behind Page Setup Options).

For more information, see "SelectButton" on page 115 and "DialogButton" on page 167.

Click

Use the Click option to insert screen or window coordinates for use with the Click command, or any other keyword that requires screen coordinates as a parameter. When you choose **Click**, a dialog box appears:



You can click the buttons in the miniature screen to reposition the dialog box if it's in the way of where you want to click.

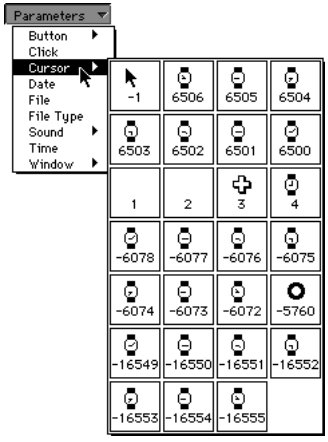
When you click and release the mouse button, OneClick inserts the mouse click's coordinates in the script. If you click within a window, the coordinates are local to the window; if you click outside of a window (such as on the desktop), the keyword Global is also inserted, indicating the coordinates are global to the entire screen.

If you click and drag the mouse, OneClick inserts two pairs of coordinates (the starting point and the ending point of the drag).

See "Click" on page 101 for more information.

Cursor

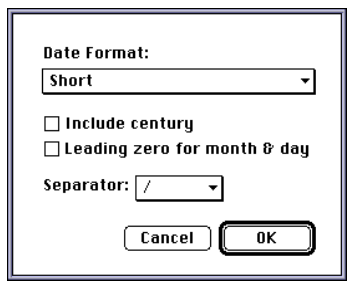
Use the Cursor submenu to insert the ID number of a cursor (for use with the Cursor system variable). The Cursor submenu shows all the cursors available in the System file and the active application, with the ID number of each cursor. Choosing a cursor from the submenu inserts its ID number in the script.

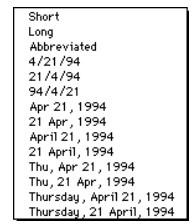


See “Cursor” on page 147 for more information.

Date

Use the Date option to insert a date format number for use with the Date function. When you choose Date, the Date Format dialog box appears:





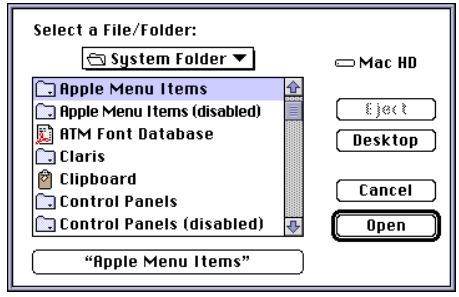
Date Format pop-up menu

Choose options in the dialog box to specify the format you want the Date function to return, then click **OK**. OneClick inserts in the script the format number that corresponds to the options you chose in the dialog box.

See “Date” on page 127 for more information.

File

The File option displays a dialog box that lets you choose a file or folder, then inserts in the script the full path to chosen the file or folder. Use File to insert a path for any keyword that requires a path parameter.



The button at the bottom of the dialog box shows the currently selected file or folder.

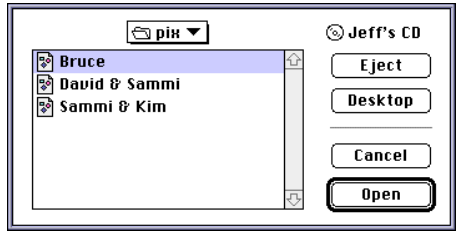
To choose a file, locate the file and then click **Select**, or click the button at the bottom of the dialog box. To choose a folder, locate the folder and then click the button at the bottom of the dialog box.

Folder paths always end in a colon (:), file paths do not. For the example dialog box above, the following path appears in the Script Editor:

```
"Mac HD:System Folder:Apple Menu Items:"
```

File Type

The File Type option inserts the four-character file type code (such as “TEXT” or “PICT”) of a file you choose. When you choose File Type, a directory dialog box appears.



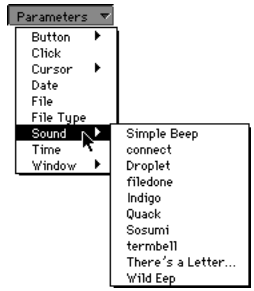
Choose the file whose file type code you want to insert, then click **Open**. OneClick inserts in the script the file type code of the chosen file.

The following keywords can use a file type parameter:

- AskFile (page 124)
- PopupFiles (page 137)
- File.Kind (page 170)

Sound

Use the Sound submenu to insert the name of a sound for use with the Sound command. The Sound submenu lists all the sounds available in the System file and the active application.

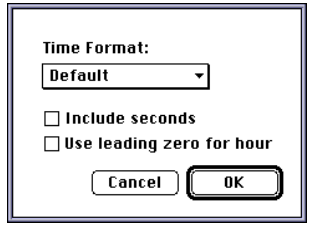


OneClick inserts in the script the name of the sound chosen from the submenu.

See "Sound" on page 118 for more information.

Time

Use the Time option to insert a time format number for use with the Time function. When you choose Time, the Time Format dialog box appears:



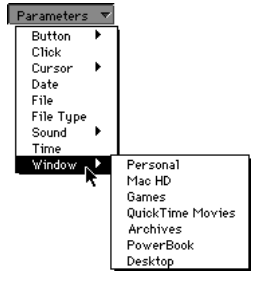
Time Format pop-up menu

Choose options in the dialog box to specify the format you want the Time function to return, then click **OK**. OneClick inserts in the script the format number that corresponds to the options you chose in the dialog box.

See “Time” on page 142 for more information.

Window

The Window submenu contains the names of all windows in the active application.



Use the Window submenu to quickly insert the name of a window for use with the Window object or another keyword that requires a window name as a parameter.

See “Window” on page 191 for more information.

Script compiler error messages

This section lists the possible error messages you may encounter when saving a script or checking its syntax and the solutions for each problem.

Unknown name

The script compiler doesn't recognize the name of a keyword or variable name as you've typed it.

- If a variable name is highlighted, make sure you declared the variable in a Variable statement before the variable is used in the script.
- If a script keyword is highlighted, make sure you spelled the keyword correctly.
- If part of a literal string is highlighted, make sure the string is enclosed in quotes (").
- If part of a comment is highlighted, make sure the comment follows a // (comment) keyword.

Not a command

The script attempted to use a function as if it were a command (the function is outside of an expression or assignment statement). Make sure you're assigning the function to a variable or evaluating it in an expression.

Invalid variable name

The name you specified in a Variable statement cannot be used, probably because it contains punctuation characters other than an underscore (_) or because it's the same name as a script keyword. See the rules for naming variables on page 50.

Missing '"'

A closing quote mark (") is missing at the end of a literal string. When this error occurs, the cursor usually appears on the line below the line that's missing the quote mark.

Missing '(' *or* Missing ')'

An opening or closing parenthesis is missing in an expression. The number of left and right parentheses must match. The cursor appears at the end of the line containing the error.

Valid END specifier required

The keyword in an End statement is missing, or the End statement contains something other than For, While, If, With, Repeat, or a handler name. Make sure you've specified the correct keyword in the End statement at the end of a loop, a handler, or another block of statements. For example, a While loop must end with an End While statement.

Line too long

The statement compiles to more than 255 bytes. Try breaking the statement up into two or more statements to make it shorter.

Insufficient memory

There isn't enough memory available in the system or the active application to compile the script. If the script is on an application-specific palette, try closing windows to make more memory available in the active application. If the script is on a global palette, try closing applications.

AppleScript Error

OneClick cannot connect to the AppleScript scripting system to compile an embedded AppleScript statement. This usually occurs when AppleScript is not installed, or when there is not enough memory to initialize AppleScript.

Other AppleScript errors

If AppleScript encounters an error while compiling an embedded AppleScript statement, AppleScript's error message appears in the status area where OneClick messages normally appear. Refer to an AppleScript reference manual for a description of AppleScript error messages.

Unknown version of script

The script appears to have been created with a version of OneClick that's newer than the current version of OneClick you have installed, and the script cannot be decompiled or run. Normally you shouldn't ever see this message.

Chapter Four

Using EasyScript

Overview

This chapter shows you how to write scripts for buttons and enhance recorded scripts. You'll learn basic scripting techniques using OneClick's EasyScript scripting language. Topics covered in this chapter include:

- Introduction to scripting
- Parts of the EasyScript language
- Common scripting techniques
- Testing and debugging a script

About scripting

OneClick's ability to record and play back a sequence of actions on the Macintosh is a powerful feature, because it can save you a lot of time and tedious repetition—letting you be more productive. OneClick's robust EasyScript language makes the sequencing ability even more powerful. Unlike scripts or macros that are simply recordings of keystrokes and mouse clicks, EasyScript scripts are recorded in a simple programming language.

At its core, EasyScript is a small language with a simple, easy-to-learn syntax. To this core, EasyScript adds dozens of commands and functions created specifically to access and manipulate the Macintosh user interface and operating environment. The addition of the built-in commands to EasyScript means that many actions that would take a number of instructions to execute in other scripting or macro software can usually be expressed with a single EasyScript command or function.

At the most basic level, you can record scripts to automate routine tasks, but that's only the beginning. When you click a button on a OneClick palette, you're simply running an EasyScript script. By learning EasyScript, you can edit and enhance your recorded scripts to increase their functionality.

Sample EasyScript scripts

While learning EasyScript, you'll find a good source of example scripts in the pre-made buttons that come with OneClick. Use the Script Editor to browse or print the scripts for different buttons. When you're not sure what a particular keyword does in a script, refer to this chapter and Chapter 5, "EasyScript Reference."

How scripting differs from programming

You don't need to know a programming language to use EasyScript. Although this chapter does not take a systematic approach to teaching a programming language, it does teach you what you need to know about EasyScript scripting.

Because of the power of the EasyScript language, the difference between it and a traditional programming language (such as BASIC, Pascal, or C++) may seem to blur. Although there are many features that EasyScript shares with traditional programming languages, there are a few distinct areas in which EasyScript is different.

Ability to create stand-alone applications

Programming languages are designed to let programmers develop stand-alone applications from the ground up. A typical application, such as a word processor, consists of thousands of lines of code that may take months or even years to develop.

EasyScript scripts usually perform just a single task or series of tasks *within* an application. Compared to a programming language, EasyScript scripts are very short and to the point; most of the scripts you'll write may contain no more than a few lines of EasyScript statements. Scripts that perform simple tasks, such as opening an application or document, may contain only one EasyScript statement.

Ability to control other applications

EasyScript commands and functions are uniquely designed to interact with an application's user interface elements, such as menus, windows, and buttons.

A programming language lets you create applications that display user interface elements, but the language itself doesn't provide the built-in ability to automatically interact with those elements.

Ability to create custom or structured data types

In a traditional programming language, the programmer can create new data types used to store information—often called records or structs, depending on the language.

EasyScript supports three data types needed to interact with the Macintosh user interface: number, string, and list, which is a special kind of string data. Lists are similar to arrays in other languages, but they can also be manipulated as string values.

Parts of the EasyScript language

In this section you'll become familiar with aspects of the EasyScript language, such as:

- statements and keywords
- values
- commands
- comments
- functions
- variables
- expressions and operators
- objects
- handlers

If you're already familiar with another scripting or programming language, skim this section to gain an understanding of the differences between EasyScript and other languages. If you're new to scripting, pay careful attention to each of the following sections.

Statements and keywords

A *statement* is one line of instructions in a script. A script is made up of one or more statements, with one statement per line in the script. You write statements using commands, functions, objects, handlers, and other elements in the EasyScript language. The names of all the different commands, functions, objects, and handlers—all words in the EasyScript language—are collectively called *keywords*.

The following are five statements in an example script. Keywords are highlighted in boldface.

```
Message "Hello, world!"
Open (FindFolder "amnu") & "Chooser"
Variable X, MyCount
X = Date 1
MyCount = MyCount + 10
```

The first two statements contain commands and their parameters. The third statement declares two variables named X and MyCount. The fourth statement assigns the result of the Date function to the variable X. The last statement adds 10 to the value of the MyCount variable.

Values

A value is a series of characters (called a string value) or a number (called a numeric value). String values can consist of any character, including numbers, symbols, and punctuation marks. String values must be enclosed in quotation marks when you type them in a script.

Numeric values are limited to numbers and a minus sign, if needed. Floating-point numbers (numbers with a decimal fraction) are not supported. A numeric value can range from –2,147,483,648 to 2,147,483,647.

Following are some examples of values. String values are enclosed in quotation marks.

```
29930
–62
"Projects"
"3:00 Meeting"
```

List values

A list value is a special type of string. A list is a series of individual strings separated by a special character, called the *list delimiter*. The preset delimiter is the Return character (↵). Examples of lists include the following:

```
"Apple↵Banana↵Navel Orange↵Strawberry↵Peach↵Pear↵Concord Grape"
"9↵26↵66↵7↵13↵63"
```

The first list contains seven string items: Apple, Banana, Navel Orange, and so on. The second list contains six strings. EasyScript treats numeric characters in a list as strings, not numbers.

Many EasyScript commands and functions use lists to perform their tasks. For example, the PopupMenu function (described later) displays a list as a pop-up menu; each individual string in the list appears as an item in the menu.

▶ **To insert the Return character (↵) in the Script Editor**

■ Press Option-Return.

You don't need to insert the ↵ character before the first item or after the last item in the list, just between each item.

For more information on how to use lists in your scripts, see "Manipulating lists" on page 75.

Commands

Commands are words that perform the work of a script. The other language elements (values, variables, functions, objects, and so on) just let you put the commands together in more useful ways.

Common commands you might use in scripts include the following:

Command	Description
Type	Types text or simulates pressing Command keys
SelectMenu	Chooses a command from a menu in the menu bar
SelectPopUp	Chooses a command from a pop-up menu

Command	Description
SelectButton	Clicks a button in a dialog box or window
Open	Opens an application, folder, document, or other Finder item
Wait	Waits for a certain condition to become true, such as waiting for a specific window to appear
Variable	Declares variables for use in a script

Command examples

Here's an example script that uses some of the above commands:

```
Open "Hard Disk:Applications:SimpleText"
SelectMenu "File", "New"
SelectMenu "Style", "Bold"
Type "Status Report — Alan Bird"
SelectMenu "Style", "Plain Text"
Type Return, "Week Ending ", Date
Type Return, Return, Return
```

The script opens SimpleText, opens a new document, types the status report heading in bold text, then types the week-ending date in plain text, followed by three carriage returns.

Parameters

Many commands and functions require one or more *parameters*. A parameter is a value you include as part of a statement so the command knows what value to work with. For example, the Type command requires at least one parameter that specifies what text or keystrokes to type. The SelectMenu command accepts two parameters: the first is the name of the menu, and the second is the name of the menu item to choose.

You can use either spaces or commas to separate multiple parameters. The following statements are equivalent:

```
SelectMenu "File", "New"
SelectMenu "File" "New"
```

Functions

Functions are commands that return a value. While commands usually perform some kind of action, such as choosing a menu item, functions usually just report a value, such as the current date or time.

Functions can be assigned to variables, used in If statements, or anywhere else a value of the specified type is expected. Common functions you might use include the following:

Function	Description
ListCount	Returns the number of items in a list
SubString	Returns a portion of a string
Date	Returns the current date as a string value

Some functions, such as AskFile and AskList, perform some action (such as displaying a dialog box) before returning a value. Other functions simply return a value.

Function examples

Here's an example script that uses the AskFile, Return, and AskButton functions:

```
Type AskFile "TEXT"
Type Return
Type AskButton "You chose a file.", "Yes, I know", "I goofed"
```

Comments

A comment is a note to yourself that you type in a script. OneClick ignores any comments in your scripts. It's a good idea to include comments in the scripts you write so that if you write something complicated, you can quickly figure out what the script does later on.

You use two slashes (//) to mark the beginning of a comment. A comment can appear on a line by itself or after a statement; a comment always extends to the end of the line. Here are some examples of comments in a script:

```
// This is my Hello World script
Sound "Quack"       // quack like a duck
Message "Hello, World!"       // displays a greeting in a dialog box
```

You can also use the comment marker to "comment-out" statements you don't want OneClick to execute while you're testing a script. Just put the comment indicator at the beginning of the statement you want OneClick to ignore:

```
// This is my Hello World script
// Sound "Quack"    // quack like a duck
Message "Hello, World!"        // displays a greeting in a dialog box
```

The above script works like the previous version, except it doesn't play the Quack sound. When you want to re-enable a statement you commented out, just remove the comment marker.

Variables

Variables are containers which store a string or number value that can change as a script runs. In script statements, you can use variables instead of literal values (text or numbers typed directly in the script).

Before you can use a variable, you must first use the Variable command to declare the variable's name. Declaring a variable name lets OneClick recognize the word as a variable when it appears in your script.

Variables must be named according to these rules:

- The variable name must start with a letter (A–Z or a–z).
- The rest of the name can contain letters, numbers, or underscores (_).
- The name can be up to 255 characters long.
- The name can't be the same as an EasyScript keyword or any other type of variable.

Following are some examples of correct and incorrect variable names:

Variable name	Valid?
theText	Yes
My_Number_Variable	Yes
X	Yes
Message	No (Message is an EasyScript keyword)

Variable name	Valid?
num-lines	No (contains punctuation other than an underscore)
4files	No (starts with a number)

When you save a script or check its syntax in the Script Editor, OneClick checks to make sure the variable names you declared are all valid. If you use an invalid variable name, the Script Editor displays the message "Invalid variable name" and highlights the name so you can change it.

Variable names are not case-sensitive. The variable names "thetext", "TheText", and "THETEXT" all refer to the same variable. When you save a script or check its syntax, OneClick changes the case of variable names to match the case used in the Variable statement.

The variables you declare assume their type (string or number) the first time they are assigned a value, so you don't need to explicitly declare them as string or number variables like you might do in some programming languages.

Assigning variables

Use the equal (=) operator to assign values to variables:

```
MyNumberVar = 47024
MyStringVar = "Monday is my favorite day of the week"
FruitListVar = "Apples↵Oranges↵Bananas↵Pears"
WindowListVar = Window.List
```

As mentioned earlier, variables assume their type when they are initially assigned. However, you can change the type of a variable by assigning it a value of a different type. For example, consider the following:

```
MyStringVar = "Forty Two"
MyStringVar = 42
```

In the second statement, variable MyStringVar becomes a numeric variable containing the value 42 instead of a string variable.

The MakeText and MakeNumber functions allow you to interpret a string variable as a numeric value and vice versa. For example:

```
MyVar1 = 42
MyVar2 = MakeText MyVar1
```

MyVar1 contains the numeric value 42 and MyVar2 contains the string value "42".

Note A variable has no value until you assign a value to it. When a variable has no value, it is considered equal to both the empty string ("") and zero (0).

Local and global variables

When you declare a variable, you can access that variable only from within the script in which the variable is defined. This kind of variable is called a *local* variable because it can only be accessed locally within a single script; other scripts cannot access the same variable. Local variables have no value when they are declared and lose their value when the script ends.

Global variables, unlike local variables, can be shared between scripts in different buttons. Because they are meant to be shared between different scripts, global variables do not lose their value when a script ends. When a global variable is declared and assigned a value in one script, the variable's value is not re-initialized when it's declared in another script.

Global variables do lose their values when the application for which the script was written quits. For example, if you assign values to global variables in scripts written for a SimpleText palette, those variables lose their values when you quit SimpleText. The variables are re-initialized the next time you open SimpleText and run the script.

To access a variable from any script (either on the current palette or from another palette within the same application), use the Global keyword in the Variable statement:

```
Variable Global FavoriteTeam
```

The above statement declares one global variable, FavoriteTeam. You can now access FavoriteTeam from any other script that also declares FavoriteTeam as a global variable. Each script that accesses a global variable must declare it. (Make sure to use the Global keyword and to spell the variable name the same in each script.) Here are two scripts (for a pair of buttons) that share a global variable:

```
// Script #1: This script shows a list box and gets a response
Variable Global FavoriteTeam
FavoriteTeam = AskList "Padres␍Dodgers␍Giants", "Pick your favorite team:"
```

```
// Script #2: This script shows the result of the AskList function in script #1
Variable Global FavoriteTeam
Message "My favorite team is the " & FavoriteTeam
```

You can share global variables between scripts on the same palette or between scripts on different palettes. The only limitation is that you cannot share global variables between palettes of different applications. For example, if you have a global variable named PictureName in both an Adobe Photoshop palette and a Microsoft Word palette, EasyScript treats the variable as two different global variables. This is because only one application's palettes are available at a time—when Photoshop is active, only Photoshop's palettes are active; the Microsoft Word palettes (including its buttons, scripts, and therefore variables) are unavailable.

Global variables on global palettes work the same way. A script on a global palette can access global variables only on other global palettes, not on application-specific palettes. Likewise, scripts on application-specific palettes cannot access global variables on global palettes.

Tip for naming global variables

When working with global variables, it's a good idea to come up with unique variable names to avoid potential conflicts with global variables in other scripts. For example, consider a script that relies on the following global value:

```
Variable Global Num
Num = 16
```

If another script also has Num declared as global variable, and each script assigns a different value to Num, then the scripts may not work correctly if Num contains a value that one of the scripts didn't expect.

A better strategy is to use local variables, when possible, and change the global variable names to more unique (but still readable) names. For example, you might add an abbreviation of the button's name to the global variable name, so the variable name is distinct from any global variables declared in other scripts:

```
Variable Global QH_Num
QH_Num = 16
```

Static variables

When you need to store data in a variable that doesn't go away when the script ends or when the application quits, use a static variable. Static variables always remember their values, even when you shut down or restart your computer. (Static variables are stored on disk in the button's palette file.)

To declare a static variable, use the keyword Static in the Variable statement.

```
Variable Static PhoneList
Variable Static Addresses, JobLeads
```

Static variables are always local to the script in which they are declared. You cannot declare a variable to be both static and global.

System variables

A system variable is a built-in variable whose value is changed and maintained by OneClick. System variables behave like functions, except they don't require parameters and don't do any special processing like some functions do. Following are some examples of system variables:

System Variable	Description
Clipboard	Returns or sets the contents of the Clipboard.
CommandKey	True when the Command key is pressed, otherwise False.
SoundLevel	Returns the current speaker volume level (0—7) or sets the volume to a new level.

Some system variables, such as Clipboard and SoundLevel, allow you to change their value. Other system variables are maintained by OneClick and cannot be changed in a script.

Here is a sample script that uses the SoundLevel system variable:

```
Variable CurrentSound
CurrentSound = SoundLevel
SoundLevel = 7
Sound "Quack"
Message "The sound level is " & SoundLevel
SoundLevel = CurrentSound
Sound "Quack"
Message "The sound level is " & SoundLevel
```

The script stores the value of SoundLevel (the current sound volume) in the variable CurrentSound. The script then sets the volume to 7, plays a sound, and displays a message indicating the current sound level. After you click OK in the message box, the script restores the previous sound volume, plays the sound again and displays another message box.

Expressions and operators

Values, variables, and functions can be combined into expressions using *operators*. The expressions, in turn, can be used anywhere a value is expected.

Arithmetic operators

These operators perform arithmetic on two expressions. In the following examples, assume that x = 32 and y = 45.

Operator	Description	Example	Result
–	negation	–x	–32
+	addition	x + y	77
–	subtraction	x – y	–13
*	multiplication	x * y	1440
/	integer division	x / y	0

Note Because EasyScript does not support floating-point (decimal) numbers, the division operator returns the result without the decimal fraction.

Relational operators

Relational operators compare the values of two expressions. If the comparison is true, the resulting expression has the value 1 (True). Otherwise the resulting expression has the value 0 (False). In the following examples, assume that x = 32 and y = 45.

Operator	Description	Example	Result
=	equal	x = y	False (0)
<>	not equal	x <> y	True (non-zero)
>	greater than	x > y	False (0)
>=	greater than or equal	x >= y	False (0)
<	less than	x < y	True (non-zero)
<=	less than or equal	x <= y	True (non-zero)

You can also use relational operators to compare string values. EasyScript uses the ASCII sort order for <, >, <=, and >= comparisons. In the following examples, assume that x = "One" and y = "Click":

Operator	Description	Example	Result
=	equal	x = y	False (0)
<>	not equal	x <> y	True (non-zero)
>	greater than	x > y	True (non-zero)
>=	greater than or equal	x >= y	True (non-zero)
<	less than	x < y	False (0)
<=	less than or equal	x <= y	False (0)

Logical operators

Logical operators perform logical (Boolean) operations on their operands. The result of a logical expression is either True (1) or False (0). In the following examples, assume x = 32 and y = 45.

Operator	Description	Example	Result
NOT	logical negate	NOT (x < y)	False (0)
AND	logical and	(x < y) AND (x > y)	False (0)
OR	logical or	(x < y) OR (x > y)	True (non-zero)

String operator

The string concatenation operator (&) joins two string values. Use it to glue two strings together and store the result in a string variable. In the following examples, assume that Var1= “One” and Var2 = “Click”.

Operator	Description	Example	Result
&	string concatenation	"Today is " & Date	"Today is 2/20/97"
		Var1 & Var2	"OneClick"

Parentheses

Parentheses change the order of evaluation:

Operator	Description	Example	Result
( )	parentheses	(32 + 45) * 5	385
		32 + (45 * 5)	257

When you save a script or check its syntax, OneClick checks for mismatched parentheses in expressions. (There should always be an equal number of left and right parentheses.) If a parenthesis is missing, OneClick displays a Missing ‘(’ or Missing ‘)’ message and moves the cursor to the line where the parenthesis is missing.

Operator precedence

EasyScript uses the following order of precedence to determine the order in which parts of an expression are evaluated. Operators of equal precedence (such as + and –) are evaluated from left to right.

```
unary +, –, NOT
*, /
+, _
<, >, <=, >=
=, <>
AND
OR
&
```

Control statements (branching and looping)

EasyScript provides several types of *control statements* you can use to create intelligent scripts. Control statements act on the value of an expression and execute different statements depending on the expression's value. The control statements in EasyScript are similar to those found in traditional programming languages:

- If, Else, Else If, End If
- For, Next For, Exit For, End For
- Repeat, Next Repeat, Exit Repeat, End Repeat
- While, Next While, Exit While, End While

Each set of statements has its own purpose: for conditional execution and decision making; looping (repeating statements); and conditional looping.

Conditionally executing statements

Use an If…End If statement to execute one or more statements only when a certain condition is true. All statements between If and End If are executed only if the condition in the If statement is true. If the condition is false, the statements between If and End If are skipped. The syntax of the If statement is as follows:

```
If expression
    statements
End If
```

Here's an example script that uses If and End If to compare the values of two variables, X and Y:

```
Variable X, Y
X = 12
Y = 43
If X < Y
    Message "X is less than Y"
End If
Sound "Quack"
```

In the above script, a message box appears only when the value of X is less than Y. If X is greater than or equal to Y, the Message statement is skipped. Execution always continues with the Sound statement following the End If statement.

An expression is true if it is a number that is not equal to zero, or a string that is not equal to the null string (""). For example, the Sound statement in the following script will execute if X = 35 or X = "Hello", but will not execute if X = 0 or X = "" (the null string).

```
If X
    Sound "Quack"
End If
```

The Else statement lets you specify alternate statements to execute if the condition in the If statement is false. The syntax of an If, Else, End If statement is as follows:

```
If expression
    statements
Else
    statements
End If
```

Here's an example (similar to the previous script) that uses an Else statement.

```
Variable X, Y
X = 12
Y = 43
If X < Y
    Message "X is less than Y"
Else
    Sound "Indigo"
    Message "X is NOT less than Y"
End If
Sound "Quack"
```

In the above script, a message appears if X is less than Y, just as it did in the previous script. But if X is not less than Y (the condition is false), then the Indigo sound plays and a different message appears. As before, the Quack sound plays following the End If statement, regardless of the condition in the If statement.

You can create a series of If statements using one or more Else If statements. Each Else If statement contains a different expression to evaluate. Here's the syntax of an If statement that includes one Else If statement:

```
If expression
      statements
Else If expression
      statements
Else
      statements
End If
```

As with a regular If…End If statement, the Else statement is optional.

The sample script below uses an If statement with two Else If statements. To run the script, you choose a name from a pop-up menu button; the pop-up menu contains three names (Lucy, Viki, and Erica). The If and Else If statements type a certain mailing address depending on the value of MyChoice (the name chosen from the pop-up menu). After one of the mailing addresses is typed, the script finishes by typing a salutation.

```
// Get a choice from a pop-up menu
Variable MyChoice
MyChoice = PopupMenu "Lucy↵Viki↵Erica"

// Type a different address depending on the value of MyChoice
If MyChoice = "Lucy"
      Type "Lucy Coe↵Deception, Inc.↵Port Charles, NY↵↵"
Else If MyChoice = "Viki"
      Type "Viki Carpenter↵The Banner↵Llanview, PA↵↵"
Else If MyChoice = "Erica"
      Type "Erica Kane↵Enchantment, Inc.↵Pine Valley, PA↵↵"
End If

// Type the salutation
Type "Dear ", MyChoice, ", ↵↵"
```

If statements can be nested to create even more complex conditions. For more information, see "If, Else, Else If, End If" on page 109.

Repeating a sequence of statements a number of times

Use a Repeat…End Repeat loop to repeat one or more statements a certain number of times. All statements between Repeat and End Repeat are repeated the number of times specified by the Repeat parameter. The following script opens, resizes, and moves new document windows in SimpleText:

```
// Assign the starting values for the window position
Variable X, Y, HowMany
X = 10
Y = 45
HowMany = AskText "How many new windows?"

// Open and cascade some new windows
Repeat HowMany
    SelectMenu "File", "New"
    Window.Location = X, Y
    Window.Size = 300, 250
    X = X + 20
    Y = Y + 20
End Repeat
Message "All done."
```

This script uses the AskText function to display a dialog box and request the number of new windows to open. The result is stored in the HowMany variable, which is used as the parameter to the Repeat command. If HowMany is greater than zero, then the script executes the statements between Repeat and End Repeat the number of times specified by HowMany, then continues with the statement following End Repeat. If HowMany is zero or a negative number, the Repeat loop is skipped entirely and execution continues with the statement following End Repeat.

To improve readability, the statements between Repeat and End Repeat are indented automatically when you check or save the script.

Repeating a sequence of statements using a counter

A For…End For loop is similar to a Repeat…End Repeat loop, except you supply a variable that OneClick increments each time through the loop. You can use this counter variable in statements within the For loop, perhaps as an index into a list value. (See “Manipulating lists” on page 75.)

A For…End For loop in EasyScript is very much like a For…Next loop in many programming languages. The syntax for a For…End For loop is as follows:

```
For index-variable = start To end
    statements
End For
```

Index-variable is a variable you declare at some point before the beginning of the For loop. You don't need to initialize its value. *Start* is a numeric value (or expression) that indicates the starting value for *index-variable* in the beginning of the loop. Each time through the loop, OneClick adds 1 to *index-variable* and then compares the new value with *end*, a numeric value (or expression). When *index-variable* is greater than *end*, the loop terminates and execution continues with the statements following End For. Here's an example:

```
Variable X
For X = 1 to 5
    Message X
End For
```

The first time through the loop, X equals 1 (the value of *start*). OneClick increments X each time through the loop until X equals 5 (the value of *end*). The result of this script is a series of five message boxes, displaying the numbers 1 through 5.

Start can be any number; it doesn't need to be 1. *End* must be greater than *start*, however. (You cannot loop backwards, counting down from *end* to *start*.)

For more information, see "For, Next For, Exit For, End For" on page 108.

Repeating statements while a condition is true

Use a While…End While loop to repeat one or more statements while a certain condition is true. The following script loops through all of the open windows in an application, saving and closing each document until there are no more open windows.

```
While (Window.Front <> "")
    SelectMenu "File", "Save"
    CloseWindow
End While
Message "All done."
```

The script works by repeatedly checking the Window.Front property, which is equal to the empty string ("") if there are no open windows.

The While loop works as follows: The expression following the While command (Window.Front <> "") is tested. If it is true (Window.Front is not equal to the empty string), the statements between While and End While are executed. Then the expression is re-tested, and if true, the body of the loop is executed again. When the expression becomes false (Window.Front equals the empty string) the loop ends, and execution continues at the statement following End While.

In a While…End While loop, it's possible to use an expression in the While statement that never evaluates to false. This causes an endless loop—the statements between While and End While continue to repeat and the loop never ends. You can press Command-period to stop a script that's stuck in an endless loop.

To improve readability, the statements between While and End While are indented automatically when you check or save the script.

For more information, see "While, Next While, Exit While, End While" on page 121.

Pausing a script for a specified period of time

Use the Pause command to wait for a certain duration. Pause accepts one parameter, the number of 1/10ths of a second to wait:

```
// Wait 10 seconds between saving the file and quitting
SelectMenu "File", "Save"
Pause 100
SelectMenu "File", "Quit"
```

In the example script, Pause waits 10 seconds (100/10ths of a second) before quitting an application. While the script is paused, you can press Command-period to stop and cancel the script.

Pausing a script until a condition is true

When you want a script to stop running until some action occurs in the application (such as waiting for a certain window to appear), use the Wait command to check for a condition:

```
// Open our e-mail program, then open the "In Basket" window and wait for it to appear.
Open "Macintosh HD:Applications:E-mail"
Type Command "I"
Wait (Window.Front = "In Basket")
```

```
// The "In Basket" window appeared, so let's click the "Check Mail" button
Button "Check Mail"
```

You can use any logical expression in a Wait statement. The Wait command evaluates the expression repeatedly until the result is True (1), then the script resumes running.

Note that you have control of the application while the Wait statement is waiting for something to happen. By using this feature, you can create interactive scripts in which the script does something, stops and waits for you to do something, then continues doing something else when you're done. Here is a sample script that displays the Open dialog box, waits for you to open a document, then prints the document when the document window appears:

```
Variable oldWindowCount
oldWindowCount = Window.Count
SelectMenu "File", "Open*"
Wait ((Window.Count) = (oldWindowCount + 1))
SelectMenu "File", "Print*"
SelectButton "Print"
```

The script first declares a variable, oldWindowCount, and sets that variable equal to the number of windows currently open. (Window.Count returns the number of open windows in an application.) The SelectMenu command chooses Open from the File menu, causing a directory dialog box to appear.

The Wait command then sits and waits for another window to appear; it does this by checking to see if the number of open windows is one greater than the number previously stored in oldWindowCount. While the script is waiting, you can use the directory dialog box to locate and select a file to open. When the window for the opened document appears, the number of windows will then equal oldWindowCount + 1, allowing the script to continue with the SelectMenu statement following the Wait statement.

If the expression in the Wait statement always stays False (0) and never changes to True (1), the script will appear to hang—it just keeps evaluating the Wait expression endlessly. To cancel a hung script, press Command-period.

For more information, see "Wait" on page 120.

Stopping a script before it ends normally

Use the Exit command when you want to immediately stop the execution of a script and ignore all remaining script statements. The following is a script that will either stop short or continue executing depending on whether a certain window is active:

```
If (Window.Front <> "In Basket")
      Exit
End If
SelectMenu "Mail", "Sort Mail", "by Date"
Sound "Quack"
Message "You have mail."
```

The script first checks to see if the In Basket window is the active window in an e-mail program. If In Basket isn't the active window, the Exit statement causes the script to end (no other statements are executed). If In Basket is the active window, the script continues with the SelectMenu statement (following End If) and continues to the end of the script.

Objects

An *object* is a type of data with several *properties* that describe the object. Think about a physical, real-life object, such as a banana: properties that might describe a banana object include size, color, weight, ripeness, flavor, and so on.

You can set or retrieve the value of object properties using EasyScript statements. The syntax for doing so is as follows:

```
Object(specifier).Property = value    // assigns a value to an object's property
value = Object(specifier).Property    // assigns an object's property to a value
```

Using a pair of bananas as an example, you can access the properties of the bananas and assign the properties to variables. If bananas actually supported scripting, you might also assign values to their properties. Assume you have two bananas named Chiquita and Dole:

```
Variable Size1, Size2, Weight1, Weight2

Size1 = Banana("Chiquita").Size
Size2 = Banana("Dole").Size
```

```
If Size1 > Size2
      Message "Chiquita is larger than Dole"
Else
      Message "Dole is larger than Chiquita"
End If

Banana("Chiquita").Ripeness = "Fresh"
Banana("Dole").Ripeness = Banana("Chiquita").Ripeness
```

In the above example, Banana is the object type. “Chiquita” and “Dole” are the specifiers, the names of the Banana objects. The script compares the Size property of each banana, then sets the Ripeness property of each banana to “Fresh”.

Like real-life objects, OneClick objects have properties that describe their contents or appearance. For example, a Window object has Height and Width properties that tell you the dimensions of a window on the screen.

```
// set the width of the window named "Document1" to 540
Window("Document1").Width = 540

// set the height of the window named "Checkbook" to 300
Window("Checkbook").Height = 300
```

OneClick supports the following object types:

Object type	Description
Button	A button on a OneClick palette
DialogButton	A button or checkbox in a dialog box or window
File	A file on disk
Menu	A menu in the menu bar
Palette	A OneClick button palette
Process	A running application
Screen	A monitor connected to your Mac
Volume	A mounted disk, CD-ROM, or file server volume
Window	A window in the active application

Specifying an object

An object *specifier* identifies which object a statement refers to. If you omit the specifier, OneClick assumes you're specifying the active, or default, object. Which object is considered the default object depends on the type of object you're working with. In the case of a Window object, the default object is the active (frontmost) window.

```
// set the width of the active window to 540
Window.Width = 540

// set the height of the active window to 300
Window.Height = 300
```

The following table summarizes the default objects for each object type.

Object type	Default object if no specifier given
Button	The button containing the active script
DialogButton	(No default)
File	(No default)
Menu	(No default)
Palette	The palette containing the active script
Process	The active application
Screen	The main (menu bar) screen (if you have more than one monitor connected)
Volume	The startup disk
Window	The active (frontmost) window in the active application

Setting and retrieving object properties

As you saw earlier, you can get and set the values of properties, much like you do with variables. The key difference between a property and a variable is that properties are dynamic. When you get a property's value, the value returned is the property's value at the time the statement is executed. When you assign a value to a property, the object itself changes to match the property's new value.

```
// set the width of the active window to 540
Window.Width = 540
```

```
// set the height of the active window to 300
Window.Height = 300
```

When you run the above script, you'll see that the active window's size actually changes as each statement executes.

Manipulating many properties at once

When you get or set the values of several properties for the same object, you can use a With statement to specify the object just once, which simplifies the script and reduces the amount of typing required. The following script sets four different properties of a Button object.

```
With Button("E-mail")
    .Color = 43
    .Width = 60
    .Height = 22
    .Text = "Check E-mail"
End With
```

The above script is functionally the same as the following script, written the long way.

```
Button("E-mail").Color = 43
Button("E-mail").Width = 60
Button("E-mail").Height = 22
Button("E-mail").Text = "Check E-mail"
```

Telling an object to do something

A *message* tells an object to perform some kind of action. To continue our banana analogy, two possible messages for a Banana object might be Peel and Ripen.

```
Banana("Chiquita").Peel
Banana("Dole").Ripen
```

The first statement causes the Chiquita banana to peel itself. The second statement causes the Dole banana to ripen, possibly by incrementing the banana's Ripeness property.

The Process (running application) object lets you use the Quit message to tell an application to quit itself.

```
// quit the active application
Process.Quit

// quit SimpleText
Process("SimpleText").Quit
```

The Palette and Button objects each allow you to create and delete palettes or buttons on the fly, using the New and Delete messages. This is an advanced feature that lets you create dynamic palettes and buttons—for example, you can write a script that creates buttons for all the active applications. When you quit an open application or launch a new one, the script can create a new button or delete an old one as appropriate. (The Task Bar included with OneClick does just that.)

To create a new palette, use the New message. A newly-created palette is hidden, so you'll need to set its Visible property to 1 to make it appear.

The optional Global modifier lets you create a global palette.

```
// create a new application palette named My Palette
Palette("My Palette").New
Palette("My Palette").Visible = 1

// create a new global palette named Global Controls
Palette("Global Controls").New Global
Palette("Global Controls").Visible = 1
```

You can also use the New message to create new, blank buttons on a palette. A newly-created button uses the default button properties from the Button Editor, except it's invisible (just like a new palette). This lets you set all the button's properties (color, size, location, and so on) before you make the button visible.

Here's a script that creates a row of five buttons and sets their properties. The row of buttons appears in the upper-left corner of the palette.

```
// create a row of five new, blank buttons
// the buttons are named "1" to "5"
```

```
Variable X
For X = 1 to 5
    Button(MakeText X).New
    With Button(MakeText X)
        .Color = 43
        .Width = 22
        .Height = 22
        .Top = 1
        .Left = (X – 1) * 23
        .Visible = 1
    End With
End For
```

To delete a button or palette, use the Delete message.

```
// permanently delete the palette named Switcher
Palette("Switcher").Delete

// permanently delete the button named Temp
Button("Temp").Delete
```

Supported object properties and messages

All objects support multiple properties, and many objects have properties with the same names. You can get and set the values of most properties; however, some properties are read-only, meaning you can't set their value. For example, Window and Palette objects each have a .Name property containing the name that appears in the window or palette's title bar. You can set the .Name property of a Palette object to change the name in a palette's title bar, but you can't do the same for a Window object—you cannot change the name of a window.

The .Size and .Location properties are write-only—you can set their values, but you can't retrieve them, because these properties contain a pair of values instead of a single value.

```
Window.Size = 540, 300
Window.Location = 50, 50
```

To get an object's .Size property, get the .Height and .Width properties instead; to get an object's .Location property, get the .Top and .Left properties.

Some objects have the same property, but the property's meaning is different depending on the object. File and Button objects each have a .Text property, for example; for the Button object, the .Text property contains the text that appears on

the button. But for a File object, the .Text property contains the text in the specified file.

The following table summarizes the properties and messages available for each object.

Properties & Messages / Objects	Button	DialogButton	File	Palette	Process	Menu	Screen	Volume	Window
.Border	●								
.Checked		❍				❍			
.Color	●			●			●		
.Count	❍	❍	❍	❍	❍	❍	❍	❍	❍
.Creator			●		❍				
.Delete	◆			◆					
.Depth							●		
.Drag				◆					
.Eject								◆	
.Enabled		❍				❍			
.Exists	❍	❍	❍	❍	❍	❍	❍	❍	❍
.Folder					❍				
.Free					❍			❍	
.Front					●				●
.Grow				◆					
.Height	●			●			❍		●
.Help	●								
.Icon	●								
.IconAlign	●								
.Kind			●		❍				❍
.Left	●			●			❍		●
.List	❍	❍	❍	❍	❍	❍		❍	❍
.Location	✪			✪					✪
.Locked			●						
.Mode	●								
.Name	●			●	❍	❍		❍	❍
.New	◆			◆					

Key: ● Read/write ❍ Read-only ✪ Write-only ◆ Message

Properties & Messages / Objects	Button	DialogButton	File	Palette	Process	Menu	Screen	Volume	Window
.NewFolder			◆						
.Quit					◆				
.Script	●								
.Selection					●				
.SendAE					◆				
.Size	✪			✪	❍			❍	✪
.Text	●		●						
.TextAlign	●								
.TextColor	●								
.TextFont	●								
.TextSize	●								
.TextStyle	●								
.TitleBar				●					
.Top	●			●			❍		●
.Update	◆			◆			◆		◆
.Unmount								◆	
.Visible	●			●	●				●
.Width	●			●			❍		●
.Zoom									●

Key: ● Read/write ❍ Read-only ✪ Write-only ◆ Message

For more detailed information about objects and their associated properties and messages, refer to the descriptions of each object in Chapter 5, "EasyScript Reference."

Handlers

A *handler* is a series of statements that run when a specific event occurs, such as when you click the button or when you drag a Finder icon to the button.

The syntax for writing a handler is as follows.

```
On HandlerName
    statements
End HandlerName
```

Statements inside a handler are run only when the event associated with the handler occurs. A script can contain more than one handler to respond to different kinds of events. The following script contains three common handlers: Startup, Scheduled, and DragAndDrop.

```
// these statements run only once when the application starts up
On Startup
      Sound "Wild Eep"
      Schedule 100
End Startup

// these statements run every 10 seconds after the Schedule 100 command runs
On Scheduled
      Sound "Quack"
End Scheduled

// these statements run only when a Finder icon is dragged to the button
On DragAndDrop
      Sound "Sosumi"
      Message GetDragAndDrop
End DragAndDrop

// this is the default handler—these statements run only when you click the button
Sound "Indigo"
Message "I've been clicked!"
```

The default handler for a script is the MouseUp handler. You don't need to explicitly write a MouseUp handler when writing a script; statements not in any handler are assumed to be in a MouseUp handler. The MouseUp handler runs whenever you click and release the mouse on a button.

The exception to this rule is when the script contains a PopupMenu, PopupPalette, or PopupFiles command. Because these commands require you to hold down the mouse button while you choose something from the popped-up menu or palette, the default handler becomes MouseDown instead of MouseUp. You don't need to explicitly write a MouseDown handler; if the script contains PopupMenu, PopupPalette, or PopupFiles, the script automatically runs when you click (but before you release) the mouse on a button.

The following table summarizes the handlers that OneClick supports.

Handler	Description
DragAndDrop	Executed when a Finder icon or text clipping is dragged and dropped on the button
DrawButton	Executed when OneClick draws or redraws the button
MouseDown	Executed when you click the mouse on a button, but before you release the mouse
MouseUp	Executed when you click and release the mouse on a button
Scheduled	Executed when a Scheduled event occurs (initiated with the Schedule command)
Startup	Executed when any of the following occur: ■ the application starts up (for global palettes, Startup handlers execute after the computer starts up) ■ a button assigns a script containing a Startup handler to another button ■ you edit a script containing a Startup handler and then close the OneClick Editor window ■ you import a palette that contains a Startup handler in one of its scripts ■ you copy a button that contains a Startup handler from a palette or the Button Library to another palette

See the descriptions of individual handlers in Chapter 5, “EasyScript Reference,” for more information about each handler.

Common scripting techniques

This section shows you how to perform certain tasks that you can use in a variety of different scripts. You’ll learn how to use different commands together in new ways, letting you create even more powerful and useful buttons for your palettes.

Many of the examples in this section are taken from the actual scripts included with OneClick. For more information about the particular commands described in this section, see Chapter 5, “EasyScript Reference” in this manual.

Finding the checked item in a menu

You can use the Menu object to determine which item in a menu is checked, if any. Here's an example script that sets the button's text label to the font name that's checked in the Font menu of a word processor.

```
On Startup
    Schedule 5
End Startup

On Scheduled
    Menu.Update
    Button.Text = Menu("Font").Checked
End Scheduled
```

The Menu object's .Checked property returns a list of checked items in the specified menu. Only one font is selected at a time, so the .Checked property returns the name of the checked font. If no menu items are checked, .Checked returns the empty string ("").

The script example is a scheduled script that runs once every half second. (See "Scheduling a script to run periodically" on page 90.) The statement that does all the work is the Button.Text statement: Button.Text changes the text label of the button to the value of the Menu.Checked property, which is the checked font name. The script runs every half second so that as you click different sections of text that use different fonts, the button's text label is continually updated with the selected font name.

Some applications don't update the checkmarks and enabled/disabled status of menu items until you pull down a menu, which would cause .Checked to give incorrect results. The Menu.Update statement forces the application to update its menus before .Checked looks for checked menu items. For applications that do update their menus normally, you don't need to use Menu.Update.

Manipulating lists

OneClick provides a lot of versatility through the use of the list data type, since you can treat a list value as both a single string and as a collection of strings. Several commands and functions let you create lists and access list elements as if the list were an indexed array. Also, some objects return a list of items as an object property: for example, Process.List returns a list of active applications, and Window.List returns a list of all open windows.

Accessing items in a list

The ListItems function lets you get individual items out of a list. ListItems returns the list item at the numeric position you specify. In the following example, a message box displays the second item in the list (Banana).

```
Variable fruitList, theFruit
fruitList = "Apple↵Banana↵Orange"
theFruit = ListItems fruitList, 2
Message theFruit
```

The ListCount function returns the number of items in a list. Using this information, you can write a For loop to loop through every item in a list. The following script loops through all the fruits in a list and displays each fruit in a message box.

```
Variable fruitList, fruitCount, theFruit, X
fruitList = "Apple↵Banana↵Orange↵Strawberry↵Peach↵Pear↵Grape"
fruitCount = ListCount fruitList
For X = 1 to fruitCount
      theFruit = ListItems fruitList, X
      Message theFruit
End For
```

Accessing items in file paths and other types of lists

A list is usually a series of substrings separated by ↵ (the Return character). By changing the ListDelimiter system variable, you can use list values to work with other types of lists, not just lists containing one-line strings. For example, the following script displays a directory dialog box from which you can select a file. It then displays three messages showing the full path of the chosen file, the file's name, and the name of the volume it's on.

```
Variable thePath theDisk theFileName
thePath = AskFile
ListDelimiter = ":"
theDisk = ListItems thePath, 1
theFileName = ListItems thePath, –1      // –1 gets the last item in the list
Message "The complete path is: " & thePath
Message "The volume name is: " & theDisk
Message "The file name is: " & theFileName
```

The script changes the ListDelimiter value, which is normally ↵ (Return), to the colon (:) character. Because paths use colons to separate folder and file names, you can treat a path as a list of items. The first item in the list is the volume or disk name and the

last item is the file name. Other items between the first and last items (if any) are folder names.

Another useful feature is the ability to access a word in a sentence. A sentence is just a list of words separated by space characters.

```
Variable theSentence
theSentence = "Bats are not rodents, Dr. Meridian."
ListDelimiter = " "     // space character
// Display a message box containing the word "Bats"
Message ListItems theSentence, 1
```

Creating multi-dimensional lists

Using the ListDelimiter system variable, it is possible to have lists of lists. This allows you to use lists as multi-dimensional arrays or lists of records.

For example, if you want a list of names, telephone numbers, and ages, separate each record in the list with a Return character and each field within a record with a slash (/) character. To get an individual record out of the list, use the Return delimiter. After you have the record, change the delimiter to a slash (/) to extract each field of the record. If you put the name as the first field in the record, you can sort the records by name (make sure the delimiter is Return before sorting).

```
// Define a list.
Variable myList, Record, Telephone
myList = "Oberrick, J./555-2708/22↵Renstrom, R./555-5721/25↵Bird, A./555-6020/29"

// Sort by name. ListDelimiter is Return by default.
myList = ListSort myList

// Get the telephone number of the 2nd record.
Record = ListItems myList, 2
ListDelimiter = "/"
Telephone = ListItems Record, 2
ListDelimiter = Return
Message Telephone
```

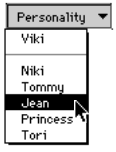
Following is a brief summary of the commands, functions, system variables, and object properties that support lists. For more information about each item, see the appropriate section in Chapter 5, "EasyScript Reference."

Keyword	Description
AskList	Displays a list in a list box and returns a list of the selected item(s).
GetDragAndDrop	Returns a list of paths when multiple Finder items are dropped on a button.
GetResources	Returns a list of resources of the specified resource type (sound, font, and so on).
ListCount	Returns the number of items in a list. Useful for accessing the last item in a list, or for processing items in a list by starting with the last item and ending with the first.
ListDelimiter	Sets or gets the character used to separate items in a list. The default list delimiter is ↵ (Return).
ListItems	Returns (as a list) one or more items from another list. Lets you access (by number) individual items in a list.
ListSort	Alphabetically sorts all items in a list.
ListSum	Adds together all numbers in a list and returns the numeric result.
Button.List	Returns a list of buttons on the specified palette, or on the palette containing the script if no palette is specified.
DialogButton.List	Returns a list of buttons, radio buttons, and checkboxes in the active window or dialog box.
File.List	Returns a list of files or folders in the specified folder, or files in the current folder if no folder is specified.
Menu.List	Returns a list of menu items in the specified menu, or a list of menus in the menu bar if no menu is specified.
Palette.List	Returns a list of global and application palettes available in the active application.
Process.List	Returns a list of all open applications.
Window.List	Returns a list of all visible, named windows. (Hidden windows and windows without a name don't appear in the list.)

Creating pop-up menu buttons

Many of the pre-made OneClick buttons behave as pop-up menus. You can create your own pop-up menu buttons using the PopupMenu function. PopupMenu accepts a list of menu items as a parameter and returns the chosen item as a string. A dash (–) in the list appears as a divider line in the menu.

This sample button uses the built-in pop-up menu border style and the button's text is "Personality".




```
Variable theChoice
theChoice = PopupMenu "Viki↵–↵Niki↵Tommy↵Jean↵Princess↵Tori"
Message "You chose " & theChoice
```

When you choose an item from the pop-up menu, the text of the item chosen is assigned to the variable theChoice. The script continues by showing a message box containing the item you picked. If you don't choose an item from the menu, then PopupMenu returns the empty string ("").

When you run a script that contains the PopupMenu function, the pop-up menu appears while you hold the mouse down on the button. Any statements that appear before the PopupMenu function execute normally, so it's a good idea to keep the number of statements prior to the PopupMenu statement to a minimum. Doing so allows the menu to pop up more responsively.

You can also use a list function or property (such as File.List) to return a list for the PopupMenu function to use. Here's a script that shows a pop-up menu of all the files in the Control Panels folder.

```
Variable theChoice
theChoice = PopupMenu File(FindFolder "ctrl").List
Open (FindFolder "ctrl") & theChoice
```

In this example, File.List returns a list of all the files in the Control Panels folder (FindFolder returns the path to the Control Panels folder). Choosing a file from the pop-up menu assigns the chosen file name to the variable theChoice; the Open command then opens the chosen file in the Control Panels folder.

Getting input while a script runs

Several functions let you add dialog boxes to your scripts to get input during script execution.

To do this in a script	Use this function
Display a message with one to four buttons and return the button clicked	AskButton
Display a message with an edit box and return the text typed in the box	AskText
Display a message with a list box and return the selected item(s)	AskList
Display a directory dialog box and get the path of the chosen file or folder	AskFile

The Script Editor's online help and Chapter 5, "EasyScript Reference," show examples of how you can use these functions in your own scripts.

Accessing the Clipboard

The Clipboard system variable lets you access the contents of the Clipboard. The benefits of being able to access the Clipboard contents from a script include the following:

- storing the Clipboard contents in variables
- assigning a new value to the Clipboard
- manipulating the Clipboard contents using commands and functions

Manipulating the Clipboard contents

By using EasyScript's commands and functions to manipulate the Clipboard variable, you can easily add new functionality to an application. Here's an example script for a Sort Lines button you can use in a word processor:

```
Variable UnsortedLines SortedLines
SelectMenu "Edit", "Copy"
UnsortedLines = Clipboard
SortedLines = ListSort UnsortedLines
Clipboard = SortedLines
SelectMenu "Edit", "Paste"
```

The script works by copying the current selection (a few lines of text) to the Clipboard. The UnsortedLines variable stores the contents of the Clipboard, which the SortList function sorts alphabetically. The result of the SortList function (the sorted lines of text) is stored in the SortedLines variable. To put the sorted lines on the Clipboard, the script simply assigns the SortedLines variable to the Clipboard variable. The last statement pastes the contents of the Clipboard into the active document, replacing the selection of unsorted text lines with the sorted text lines.

You could achieve the same results from the previous script by sorting the Clipboard text directly:

```
SelectMenu "Edit", "Copy"
Clipboard = ListSort Clipboard
SelectMenu "Edit", "Paste"
```

Storing Clipboard data in static variables

Assigning the contents of the Clipboard to a static variable lets you create a somewhat "permanent" Clipboard. Consider the following script:

```
Variable Static ClipContents
If OptionKey
    SelectMenu "Edit", "Copy"
    ClipContents = Clipboard
    Exit
End If
Clipboard = ClipContents
SelectMenu "Edit", "Paste"
```

When you select some text or graphics and Option-click the button, the script copies the current selection to the Clipboard and stores the contents in ClipContents, a static variable. When you click the button without the Option key, the script puts the ClipContents variable back on the Clipboard and then pastes it into the active application. Because the Clipboard is stored in a static variable, you can go back and access it any time, even after cutting or copying other material.

The script's usefulness becomes even more apparent when you duplicate the button containing the script several times. In the ManyClip palette at left, each of the eight Clipboard buttons contains a copy of the above script. Because each button has its own local, static ClipContents variable, the palette effectively gives you eight separate Clipboards—letting you store different selections of text, graphics, or other data in each button. To set the contents of a button's Clipboard, select some text or other

material in a document, then Option-click the button. To paste a button's Clipboard contents into a document, simply click the button.

Using public and private Clipboard formats

In certain applications, you'll need to use the ConvertClip command before accessing data on the Clipboard. Applications that store Clipboard data in a *private* format normally convert their Clipboard's contents when you switch applications; the Clipboard data is converted to *public* format that's usable by other applications. Because an EasyScript script may need to access the Clipboard's data without switching applications, the ConvertClip command tells the application to convert the Clipboard data to a public format as if you were about to switch to another application.

If the Clipboard system variable doesn't appear to contain the correct information when you access it, try using a ConvertClip statement before the Clipboard statement:

```
SelectMenu "Edit", "Copy"
ConvertClip
ClipContents = Clipboard
```

For more information, see "Clipboard" on page 145 and "ConvertClip" on page 103.

Creating tear-off palettes

Use the PopupPalette command to create pop-up and tear-off palettes. When a button's script contains the PopupPalette command, clicking the button displays the specified palette as a pop-up palette.

```
PopupPalette "System Folders"
```

The PopupPalette command takes one parameter, the name of the palette to display. The above script pops up the System Folders palette when you click the button. You can choose a button from the pop-up palette, or tear the palette off into a separate palette by dragging away from the pop-up palette.

Calling scripts as subroutines

When you click a button to run a script, OneClick normally executes only the statements contained in the button's script. A single script works as a self-contained program. You can use the Call command to run scripts in other buttons, either on the

same palette or on a different palette. When a called script finishes running, the calling script resumes executing with the statement following the Call statement.

```
// This is the main script; it calls "PlaySounds" as a subroutine
If Menu("Mail", "Read New Mail").Enabled
      Call "PlaySounds"
      Message "You have new mail."
Else
      Message "No mail."
End If

// This is the subroutine script in the button named "PlaySounds"
// The script plays three sounds and can be called by any other script
Sound "Sosumi"
Sound "Eep"
Sound "Indigo"
```

Calling scripts as subroutines lets you create large, complex scripts that are broken down into smaller, modular pieces. Once you've written a subroutine script, any other script can call the subroutine with just one Call statement—you don't need to copy and paste the entire subroutine into every script that uses it. When you make changes to the subroutine script, you don't need to make any changes to the scripts that call the subroutine.

Calling scripts as functions

EasyScript doesn't let you write functions that actually return a value when called, but you can easily mimic functions by using subroutines and global variables. To pass parameters to a function script, declare the same global variables in both the function script and the script that calls it. To simulate a return value, declare another global variable (such as "Result") in each script. Here's an example of a a function script that returns a value and another script that calls it:

```
// This is the calling script. It passes the number 134 to the
// function "MakeWords" and types the result.
Variable Global Parameter, Result
Parameter = 134
Call "MakeWords"
Message Result
```

The global variables Parameter and Result are accessible to both scripts. By assigning a value to Parameter in the calling script, then assigning another value to Result in the

function script, you can simulate passing a parameter to a function and retrieving a result.

The following script does some processing on the parameter passed to it in the Parameter variable. It converts the parameter (a number) to a string value, then looks at the text one character at a time and builds a new string containing words that represent each digit in the number. The While statement loops through each digit (character) in the number and the If, Else If, End If statement determines which words to add to the Result string based on the digit in the number.

```
// This is the function script in a button named "MakeWords". It
// takes a number parameter and returns a text string containing
// the names of each digit in the number.
Variable Global Parameter, Result
Variable X, L, C, T
T = MakeText Parameter
L = Length T
X = 1
Result = ""
While X <= L
      C = SubString T, X, X
      If C = "1"
            Result = Result & "one "
      Else If C = "2"
            Result = Result & "two "
      Else If C = "3"
            Result = Result & "three "
      Else If C = "4"
            Result = Result & "four "
      Else If C = "5"
            Result = Result & "five "
      Else If C = "6"
            Result = Result & "six "
      Else If C = "7"
            Result = Result & "seven "
      Else If C = "8"
            Result = Result & "eight "
      Else If C = "9"
            Result = Result & "nine "
      Else If C = "0"
            Result = Result & "zero "
      End If
      X = X + 1
End While
```

Like subroutine scripts, the advantage of writing function scripts is modularity. Once you've written a function script, any other script can call the function and get a result with just a few statements.

Getting a list of the installed fonts or sounds

Fonts on the Macintosh are stored as resources, and the GetResources function lets you get a list of resources of any particular type. GetResources requires one parameter, a four-character resource type, and returns a list of all the available resources of the given type.

Font resources have the "FOND" resource type, so creating a pop-up Font menu is as simple as the following:

```
Variable Choice
Choice = PopupMenu (GetResources "FOND")
SelectMenu "Font", Choice
```

The first line of the script declares a variable, Choice, to hold the result from the PopupMenu function. The second line creates a pop-up menu that lists all the available fonts. When you click the button, a pop-up Font menu appears; the name of the font you choose is stored in Choice. The third line in the script uses the SelectMenu command to choose from the menu bar's Font menu the font you selected.

Creating a pop-up Sound menu is also just as easy, because sounds are stored as resources of type "snd " (note the trailing space).

```
Variable Choice
Choice = PopupMenu (GetResources "snd ")
Sound Choice
```

Note that four-character resource types ("FOND", "snd ", and so on) are case-sensitive.

Using Drag and Drop

If you're using System 7.5 or newer (or System 7.1 with the Macintosh Drag and Drop extensions), you can drag information from an application onto a button and have the button's script act on the dropped information. Buttons can receive either plain text or Finder icons.

To support Drag and Drop in a button, you write a DragAndDrop handler in the button's script. Only buttons containing a DragAndDrop handler can receive dropped information. The DragAndDrop handler runs whenever you drop information on the button.

You use the GetDragAndDrop function to find out what was dropped on the button. When you drop a text selection, GetDragAndDrop returns the dropped text. When you drop one or more Finder items, GetDragAndDrop returns a list containing the full paths of all the dropped items.

Note Not all applications support Drag and Drop. For example, you can drag text from WordPerfect 3.1, SimpleText, or BBEdit 3.0, but not from Microsoft Word 5.1. To drag and drop Finder icons, you need Finder version 7.1.3 or newer.

Working with dropped text

The following script is a variation on the ManyClip script shown earlier in this chapter. Instead of using the Option key to store selected text in the button, the script uses a DragAndDrop handler to receive and store dropped text.

Dropping text on the button stores the dropped text in a static variable. Clicking the button pastes the stored text in the active application.

```
On DragAndDrop    // this runs only when something is dropped on the button
      Variable Static theText
      theText = GetDragAndDrop    // store the dropped text in theText
End DragAndDrop

On MouseUp        // this runs only when you click the button
      Variable Static theText
      Variable tempClip
      tempClip = Clipboard      // temporarily store the Clipboard's contents
      Clipboard = theText
      SelectMenu "Edit", "Paste"      // paste theText in the active application
      Clipboard = tempClip      // restore the original Clipboard
End MouseUp
```

Because static variables are local, they need to be declared in both the MouseUp handler and the DragAndDrop handler. When the same static variable is declared in different handlers within the same script, the variable has the same value in each handler.

Working with dropped Finder items

The following script creates a pop-up menu that launches items. To add an item to the menu, drop a Finder icon on the button. To launch an item, click the button and choose an item from the pop-up menu.

```
On DragAndDrop    // this runs only when something is dropped on the button
    // the list of items is stored in a static variable so we don't lose it when we restart
    Variable Static filePathList

    // get the path of the dropped item and add it to the path list
    filePathList = filePathList & GetDragAndDrop
End DragAndDrop

On MouseDown      // this runs only when you click the button
    Variable theChoice
    Variable Static filePathList

    // Option-click the button to clear the pop-up menu
    If OptionKey
        filePathList = ""
        Exit
    End If

    // show a pop-up menu of the stored paths
    theChoice = PopupMenu filePathList
    If theChoice = ""      // nothing was chosen
        Exit
    End If
    Open theChoice      // open the chosen item
End MouseDown
```

The script adds an item to the pop-up menu by getting the path of the dropped item from the GetDragAndDrop function. GetDragAndDrop returns a list of one or more paths, so the script simply adds the path list to the existing filePathList variable. When you click the button, the PopupMenu function uses the filePathList variable to display a pop-up menu of paths. The Open command then opens the path chosen from the menu.

Creating launch buttons using Drag and Drop on a palette

Normally when you drop a Finder icon on a palette (not on a button), nothing happens. To add Drag and Drop support to a palette (to create launch buttons, for example), create a button on the palette and name it "PaletteDrop."

When you drop a Finder icon on a palette with a PaletteDrop button, OneClick first creates a new, invisible button the same size as the PaletteDrop button. The new button is positioned at the same location where you dropped the icon.

After creating the new button, OneClick then calls the DragAndDrop handler in the PaletteDrop button. In the DragAndDrop handler, you can change the new button's icon, name, or other properties, and add a script to the button. Here's an example DragAndDrop handler for a PaletteDrop button that creates a new launch button for an item dropped on the palette.

```
On DragAndDrop
    Variable Quote, thePath
    Quote = Char 34     // quotation mark (") character
    thePath = GetDragAndDrop 1
    ListDelimiter = ":"
    // Change some of the new button's properties
    With Button(Button.Count)
        .Script = "Open " & Quote & thePath & Quote
        .Icon = 1, thePath, 16
        .Name = ListItems thePath, –1
        .Visible = 1
    End With
End DragAndDrop
```

The Button.Count property returns the number of the last button added to the palette; that's how we figure out which button to change. The variable thePath (from the GetDragAndDrop function) contains the full path to the item dropped on the palette.

The With Button statement changes some of the new button's properties, including its script, icon, name, and visibility:

- The .Script property (which contains the new button's script) is set to the Open command, followed by the item's path in quotation marks. For example, if the dropped item was Mac HD:SimpleText, then the new button's script would be:

  ```
  Open "Mac HD:SimpleText"
  ```

- The new button's icon is set to the small (16-pixel) icon of the dropped item.
- The new button's name is set to the name of the dropped item (just the name, not the full path).
- After setting the other properties, the new button is made visible so it appears on the palette and can be used.

This is a fairly simple example of how to write a DragAndDrop handler for a PaletteDrop button. The script (as it is written here) does not create multiple launch buttons if you drop multiple items on the palette. It also does not add Drag and Drop support to the launch buttons it creates.

Determining how long the mouse is held down

A button's MouseDown handler runs immediately when you click the button, before you release the mouse. You can use the Ticks system variable to determine how long the mouse was held down on the button and perform different actions based on that length of time. For example, consider a button that opens a folder: If you quickly click and release the button, the folder opens, but if you hold the mouse down on the button for a specified period of time (3/4ths of a second in this example), then a pop-up menu of the folder's contents appears, letting you select a file to open.

```
On MouseDown
    Variable theFolder beginningTicks delayTime
    theFolder = FindFolder "ctrl"    // Control Panels folder
    beginningTicks = Ticks
    delayTime = 45
    While Ticks < beginningTicks + delayTime
        If NOT IsMouseDown
            // Do the following if mouse is released before delay time elapses
            Open theFolder
            Exit
        End If
    End While
    // Do the following if mouse is held down beyond delayTime
    Open theFolder & (PopupMenu File(theFolder).List)
End MouseDown
```

In this script, the beginningTicks variable contains the time (in 60ths of a second) when the button was clicked. The While loop repeatedly checks to see if the button was held down for more than 45 ticks (3/4ths of a second, the delay time). If the button wasn't held down for more than 45 ticks (meaning the button was clicked and immediately released), then the Control Panels folder opens. If the button is held down longer than 45 ticks, then a pop-up menu listing all the control panels appears; choosing an item from the menu opens a control panel.

Making a script run when an application starts

Some of the palettes that come with OneClick use startup scripts—scripts that run as soon as their application starts. Startup scripts are useful for a variety of reasons because they can perform a task whenever you start a certain application. Common startup tasks include the following:

- Opening one or more documents
- Moving the palette to a default location on the screen
- Changing the monitor's color depth (for games or graphics programs)
- Scheduling a script to run periodically (see the next section)

Use a Startup handler in a script to specify that the script should run whenever the application starts. Here's an example script from an Adobe Photoshop palette that changes the monitor's color depth whenever Adobe Photoshop is run.

```
On Startup
    // Switch the monitor to millions of colors
    Screen.Depth = 32
    // Show the Scanner Tools palette and move it down to the corner of the screen
    Palette("Scanner Tools").Visible = 1
    Palette("Scanner Tools").Location = 0, ScreenHeight – PaletteHeight
End Startup
```

Startup handlers run even if the palette is closed when the application starts up. If you don't want a script's startup handler to run when its palette is closed, you can use the following technique:

```
On Startup
    If NOT Palette.Visible
        Exit
    End If
    Screen.Depth = 32
    Palette("Scanner Tools").Visible = 1
    Palette("Scanner Tools").Location = 0, ScreenHeight – PaletteHeight
End Startup
```

Scheduling a script to run periodically

Many of the buttons on the pre-designed OneClick palettes can change their text or icon's appearance based on a menu command's state or other information. For example, a button on the System Bar periodically updates itself to show the current

time and date, and the style buttons in the SimpleText library update themselves to indicate the current styles selected in the Style menu.

All of these buttons use the Schedule command in their scripts. A scheduled script runs periodically to check the state of something (such as whether a menu command is enabled or disabled) and then change their appearance based on the current state. To update a button's appearance in real time, a scheduled script must run quite often—usually every second or half second. The Schedule command lets you specify (in 1/10 second increments) how often a script should run.

A scheduled script typically contains three parts:

- **The Startup handler** runs when the palette's application starts up. (Use the On Startup handler to indicate the script is a startup script.) The Startup handler should use the Schedule command to add the script to OneClick's list of scheduled scripts.
- **The Scheduled handler** runs whenever OneClick runs the script as a result of it having been scheduled with the Schedule command. When OneClick runs a scheduled script, it executes statements in the Scheduled handler. Statements in the Scheduled handler usually check the status of something, such as whether a menu command is enabled or disabled, then change the button's appearance based on the status.
- **The default handler** (usually MouseUp or MouseDown) runs only when you click the button. This handler contains the usual statements that perform the action of the button, such as choosing the menu command that's being monitored in the Scheduled handler.

Note The name of the command that initiates scheduling is "Schedule" and the name of the handler is "Scheduled" (with a "d" at the end).

The following script shows a simple scheduled script. The Startup handler schedules the script to run every half second.

```
// This is the Startup handler. It turns on scheduling for this script.
On Startup
      Schedule 5
End Startup
```

```
// The Scheduled handler sets the button's text to the name of the font that
// appears checked in the Font menu.
On Scheduled
    Menu.Update
    Button.Text = Menu("Font").Checked
End Scheduled
```

The next script shows how the three parts of a scheduled script work together to create a dynamically changing Get Info button. The script works by monitoring the Get Info command in the Finder's File menu; when the command is enabled, the script uses Button.Mode = 0 to change the button's icon to Normal appearance. If the Get Info command is disabled (dimmed because no Finder icon is selected), the script uses Button.Mode = 2 to give the button a disabled appearance.

```
On Startup
    Schedule 5
End Startup

// This Scheduled handler runs every half second to check the status of the File
// menu's Get Info command and change the button's appearance accordingly.
On Scheduled
    If Menu("File", "Get Info").Enabled
        Button.Mode = 0
    Else
        Button.Mode = 2
    End If
End Scheduled

// This statement is executed only when the button is clicked.
SelectMenu "File", "Get Info"
```

For more examples of scheduled scripts, see the scripts for the Font, Size, and Style buttons in the SimpleText button library. Also see the descriptions of the Startup, Schedule, and Scheduled keywords in Chapter 5, "EasyScript Reference".

Tips for making scheduled scripts run more efficiently

You should write scheduled scripts to run as efficiently as possible since they usually run very often. Scheduled scripts run only when you're not interacting with the computer (when there is no keyboard or mouse activity); however, the more time the computer spends running scheduled scripts, the less time there is available for background processes such as PrintMonitor.

Following are a few suggestions for improving efficiency.

- **Make the Scheduled handler the first handler in the script.** When OneCLick runs a script, it searches through the script to locate the appropriate handler. If the handler is at the beginning of the script, the search goes slightly faster, especially with very large scripts.
- **Try to avoid using a large number of variables, especially global variables.** It takes a small amount of time to allocate the memory required for each variable, and global variables need to be looked up in OneClick's global variable table each time the script runs. Variables aren't actually allocated until the Variable statement is executed in the script, so declare only those variables at the beginning of a Scheduled handler that are necessary to determine if further processing is needed. Others may be declared later on in the handler.
- **Try to avoid script statements that cause screen drawing to occur.** Changes to palettes or buttons, such as changing a button's .Icon property, can slow down the script because the button gets redrawn each time the script runs.
- **Try to avoid using Menu.Update to force the application to update its menus.** If you do use it, however, it doesn't hurt to use it several times or in several different scripts. OneClick will not process this statement more than twice a second, no matter how many times it is called.

Testing and debugging a script

Writing a moderately complex script usually takes some time to get the script working the way you want it to. You may run into logic errors when writing and testing a script—the script doesn't behave the way you think it should because of a mistake in the logical flow of your script. This section provides some tips and techniques to help you get your scripts working flawlessly faster.

Using message boxes to inspect variables

The Message command is a convenient way to check the value of a variable within a script while the script runs. If your script isn't running correctly because a variable doesn't contain a value you think it should, use a Message statement to show the value in a message box. Here's an example:

```
Variable fileCount, theFileList
theFileList = File("Mac HD:Data:Reports:").List
fileCount = ListCount theFileList
```

```
// We're not sure at this point what FileCount's value might be,
// so we'll show it in a message box and then exit
Message fileCount
Exit
```

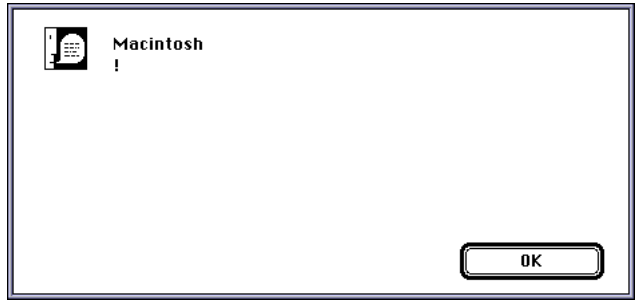
When you run the script, a message box appears showing the value of fileCount when the script reaches the Message statement. When you have the script working the way you expect and no longer need the message box, simply delete the Message statement.

You don't need to use an Exit statement after a Message statement unless you want the script to stop after the message box appears. For brevity, we've omitted the rest of the script following the Exit statement.

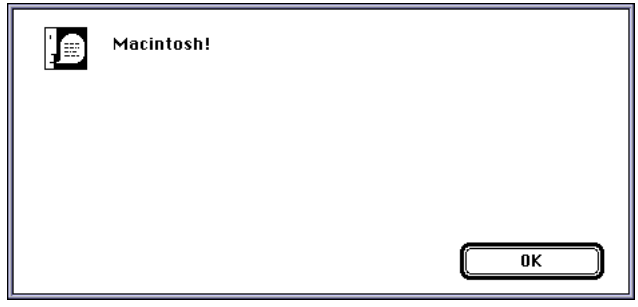
Sometimes you might be working with a string that has an extra white space character (a space, tab, or return) at the end of it. You can use the Message command to find out:

```
Message myStringVar & "!"
```

The & (concatenation) operator joins the two strings into the one string that appears in the message box:



If the string has a carriage return appended, the exclamation point appears on the second line below the string as shown above, otherwise it appears at the end of the string:



The exclamation point is just an example character used to show this technique. You can use any character you want.

Using text buttons to monitor the values of variables

If you have more than a couple of variables you want to monitor continuously, or you don't want message boxes interrupting your script (using the previous technique), you can use buttons as text displays to show the current values of variables. The Button.Text property changes a button's text label; by changing a button's text to a variable's value, you can monitor as many variables as you wish.

Here's a sample script that uses Button.Text to monitor two variables on buttons named A and B:

```
Variable X, Y
X = 0
While X <= 5
      X = X + 1
      Y = 5
      While Y >= 0
            Y = Y – 1
            Button("B").Text = Y
            Button.Update
      End While
      Button("A").Text = X
      Button.Update
End While
```

The Button.Update statements cause OneClick to update the text on the buttons while the script runs. Without the Button.Update statements, the buttons' text wouldn't get updated until the script ends.

The values of X and Y appear in two buttons on this example Testing palette during each pass through the While loops:

For global variables, this technique is even easier. Just create a button with a script that declares the global variable, then sets its text to the variable's value, such as the following:

```
Variable Global X
Button.Text = X
```

Whenever you click the button, the button's text label updates to show the current value of X.

Using sounds to determine what's being executed

When writing If and While statements, it's possible that your script might execute statements unexpectedly (or not at all)—this usually happens because the condition being tested in the If or While statement doesn't evaluate to the value you think it should. If you're not sure if a group of statements is being executed, then using a Sound statement is a quick way to find out. For example, here's part of a script that contains an If statement, and we're not sure if the expression evaluates to True:

```
If indexNumber = 3
    theData = reportTotal & reportSummary
    Sound "Quack"
End If
```

If the expression evaluates to True, the Quack sound plays, indicating that statements following the If statement are being executed. If the expression doesn't evaluate to True (or False) as we think it should, then we know whether or not there's a problem based on whether or not the Quack sound played.

Checking for run-time errors

A *run-time error* is a scripting error that occurs while a script runs, as opposed to a logic or syntax error in a script. Several commands can generate run-time errors due to a variety of reasons: invalid parameters; interface items that couldn't be found (windows, menus, buttons, and so on); a file or folder not found; out of memory; and other conditions.

Normally when a run-time error occurs, OneClick doesn't display any kind of error message—it just skips the offending statement and continues executing the rest of the script. It's up to you, the script writer, to determine whether or not a run-time error has occurred.

The Error system variable contains the error result of the last command, function, or object that reported an error. There are four possible numeric values for Error.

- 0—No error
- 1—General error (out of memory or miscellaneous errors)
- 2—Not Found error
- 3—Parameter error

See the error table on page 149 for a list of keywords and their associated error values and meanings.

Specifications and Limits

The following table summarizes the memory requirements, capacities, and data limitations for the EasyScript language.

Number range	–2,147,483,648 to 2,147,483,647
Maximum string length	Limited only by memory (4GB maximum)
Maximum length for a variable name	255 characters
Maximum number of variables in a script	Limited only by memory
Length of one line in a script	Varies; each line must compile to less than 256 bytes (keywords take two bytes each)
Length of a script	Limited only by memory
Maximum number of nested While, Repeat, or For loops	Loops can be nested up to 20 levels deep
Maximum number of nested If/Else/Else If statements	No limit
Maximum number of nested scripts (using Call to run another button's script and then return)	Scripts can be nested up to 8 levels deep
Number of buttons on a palette	65535
Number of global or application palettes	Limited only by memory

Memory usage

The OneClick control panel requires about 335K of memory when it's installed. Each palette takes an additional 250 bytes of RAM, plus about 100 bytes for every button. Button resources such as icons, scripts, and button text are purgeable so that any memory they occupy is recovered by the system if it is needed.

Global palettes occupy memory in the system heap (memory used by system software and extensions). Application palettes use memory in the application's memory partition. If you have an unusually large number of palettes or buttons for a particular application and that application is usually tight on memory, you may need to increase the application's memory size (using Get Info in the Finder). It's extremely unlikely that you'll need to do this, however.

Chapter Five

EasyScript Reference

Using the EasyScript Reference

This chapter contains detailed information about each keyword in the EasyScript language. The chapter is divided into five sections:

- Commands
- Functions
- System Variables
- Objects
- Handlers

Keywords are listed alphabetically within each section.

For information on values, expressions, operators, iteration, and other EasyScript language concepts, see Chapter 4, "Using EasyScript."

In all syntax descriptions, items in italics are values you supply. Items in square brackets are optional.

Commands

See "Commands" on page 47 for a general description of how to use commands.

AppleScript

Syntax

AppleScript *compiled-script-file*

```
AppleScript
    applescript-statements
End AppleScript
```

Description

Runs a compiled AppleScript script file, or indicates the beginning and end of AppleScript code.

Compiled-script is a path to a compiled AppleScript script. If you've written and compiled a script using Apple's Script Editor, use the AppleScript command to run the compiled script.

You can also use the AppleScript command to embed AppleScript statements within an EasyScript script. Put the AppleScript statements between the AppleScript and End AppleScript commands. When you save the button's script, OneClick tells the AppleScript extension to compile the embedded AppleScript code. Compiling AppleScript code may take a few moments, compared to the near-instantaneous compiling of EasyScript code.

You must have the AppleScript software installed to use this command. AppleScript is included in System 7 Pro and System 7.5 and is also available as a separate product.

Examples

```
// Run the Start File Sharing script included with System 7.5.
AppleScript "Mac HD:Automated Tasks:Start File Sharing"

// Open the Finder's "About This Macintosh" window.
// Requires the Scriptable Finder included in System 7.5.
AppleScript
    tell application "Finder"
        activate
        open about this macintosh
    end tell
End AppleScript
```

See Also

Appendix B, "AppleScript Information"

Beep

Syntax Beep

Description Plays the system alert sound.

Examples Beep

See Also BeepLevel (page 144), Sound (page 118), SoundLevel (page 155)

Call

Syntax Call *button-name* [, *palette-name*]

Description Runs another button's script as a subroutine of the active script. After the called script finishes running, the calling script continues running at the statement following the Call command.

Button-name is the name of the button to run. If the button is on a different palette, you must supply the palette's name in *palette-name*.

Examples
```
// Run the script in the Choose Font button, then run the script in
// the Open E-mail button on the Launcher palette
Call "Choose Font"
Call "Open E-mail", "Launcher"
```

See Also Calling scripts as subroutines (page 82), Calling scripts as functions (page 83)

Click

Syntax Click [Global] [Command] [Option] [Control] [Shift] *X*, *Y* [, *toX*, *toY*]

Description Simulates clicking the mouse at the specified location of *X* and *Y*.

The coordinates are local to the active window or dialog box. For example, Click 50, 100 indicates 50 pixels from the left edge and 100 pixels down from the top edge of the window contents (not including the title bar). Adding the Global keyword causes the coordinates to be global to the entire screen. For example, Click Global 50, 100 indicates 50 pixels from the left edge and 100 pixels down from the top edge of the screen.

Add the *toX* and *toY* coordinates to simulate clicking and dragging. When you use *toX* and *toY*, the Click statement clicks the mouse at *X*, *Y*, then drags to *toX*, *toY* and releases the mouse button.

Negative *X* coordinates measure left from the right edge of the window or screen and negative *Y* coordinates measure up from the bottom of the window or screen.

To simulate holding down a modifier key while clicking, use one or more of the following keywords in any order after the Click keyword: Command, Option, Control, or Shift.

Note To click buttons or select items from menus, use the SelectButton, SelectMenu, or SelectPopUp commands. Use Click to click other types of controls.

Examples

```
// Click 200 pixels down and 30 pixels from the right edge of the window.
Click –30, 200

// Click the right arrow of the scroll bar at the bottom of the window.
Click –20, –5

// Clicks somewhere in the upper right corner of the screen with the Shift key held down.
Click Shift Global –50, 75

// Drag from 50, 50 to 200, 200 in the active window.
Click 50, 50, 200, 200
```

See Also SelectButton (page 115), SelectMenu (page 116), SelectPopUp (page 118)

CloseWindow

Syntax CloseWindow

Description Closes the active (front) window.

Examples

```
CloseWindow
```

See Also Window (page 191)

ConvertClip

Syntax ConvertClip

Description Forces the active application to convert its Clipboard contents from a private data format to a public format. You don't need to use this command if the application you're using always stores its Clipboard contents in a public format, such as TEXT, PICT, or RTF.

If you want to store the Clipboard data in one application and use it in another application, you may need to use ConvertClip to convert the Clipboard's data prior to storing it in a variable. Normally, the Clipboard contents are converted when you switch from one application to another so that the application you're switching to can use the Clipboard data. But if you get the contents of the Clipboard and store it in a variable, then switch applications, this conversion doesn't occur in the variable. That's when you may need to use ConvertClip.

Examples

```
Variable Static theClip
ConvertClip
theClip = Clipboard
```

See Also Clipboard (page 145)

Dial

Syntax Dial *telephone-number* [, *port*]

Description Dials the specified telephone number. If you don't specify a *port*, the number is sent to the modem port. Values for *port* are:

- 0: internal speaker
- 1: modem port
- 2: printer port

If you have no modem, use port 0 (the speaker) and hold the mouthpiece of the phone close to the computer's speaker.

If *telephone-number* begins with "AT", the entire string is sent as a command to the modem.

Any non-digits in *telephone-number* except the comma are ignored. A comma indicates a pause.

Note The Dial command is an extension (external command). It's not available if the OneClick Extensions file is not installed in the Extensions folder inside the OneClick Folder (in Preferences).

Examples

```
// Dials through modem port
Dial "555-7427"

// Dials through the speaker
Dial "555-7427", 0

// Sends a command to the modem
Dial "ATS0=0S11=40DT5557427"
```

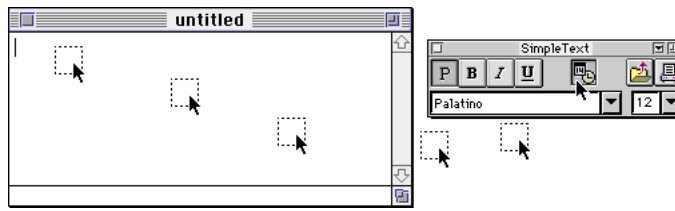
DragButton

Syntax DragButton *text*

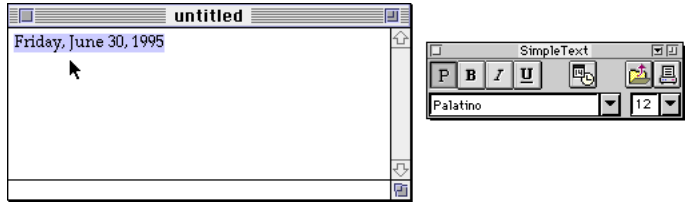
Description Drags the specified text from the button and lets you drop the text in a Drag-and-Drop-aware application. The DragButton command works only inside a MouseDown handler.

Examples

```
// Drag and drop the current date into a document. Option-drag to drag the current time.
On MouseDown
    If OptionKey
        DragButton Time
    Else
        DragButton Date
    End If
End MouseDown
```



Click the button and drag it to a Drag-and-Drop-aware application



Dropping the button in the document inserts the DragButton parameter value

```
// Option-click to choose a text file from the pop-up menu. Then drag the button
// to insert the file's contents in a document.
On MouseDown
    Variable theFolder
    Variable Static theFile
    theFolder = "Mac HD:Boilerplate Text:"
    If OptionKey
        theFile = PopupMenu File(theFolder, "TEXT").List
        Button.Text = theFile       // show the file's name on the button
    Else
        DragButton File(theFolder & theFile).Text
    End If
End MouseDown
```

DrawIndicator

Syntax DrawIndicator *progress* [, *color*]

Description Draws a progress indicator (a thermometer or pie graph) on the button. *Progress* is a percentage (0–100) that indicates how full to draw the indicator. *Color* is the color value (1–256) of the indicator. To draw a pie graph instead of a thermometer, specify a negative color value. If you omit the color parameter, DrawIndicator draws a thermometer using black as the default color.

The indicator is proportional in size to the button and fills almost the entire button. To make an indicator appear directly on the palette (not inside a button), set the button's border style to None, set its appearance to Disabled, and uncheck its color checkbox.

Examples

```
// Draw a black progress bar that's half full
DrawIndicator 50
```

```
// Draw a light purple pie graph that's 1/3 full
DrawIndicator 33, –43
```

See Also DrawButton (page 198)

Exit

Syntax Exit

Description Exits from the script, even if the script hasn't reached the end yet. If the script was called from another script, execution continues in the calling script after the Call statement.

Examples Exit

See Also Call (page 101)

FinderCopy

Syntax FinderCopy [*path-list*,] *destination*

Description Tells the Finder to copy the items specified in *path-list* to the destination folder. If you omit *path-list*, then the Finder copies the selected icon(s) to the destination folder.

Path-list may contain one or more files, folders, or a combination of files and folders. If you specify a folder, FinderCopy copies the folder and all of its contents.

If the destination folder is the same as the source folder, then the Finder simply duplicates the item and adds "copy" to the end of the new file's name.

You can use FinderCopy or FinderMove with FindFolder to copy or move items to special folders such as the System Folder, Desktop Folder, Trash, and so on.

Note FinderCopy and FinderMove work only with the Finder included in System 7.5 and System 7 Pro. To determine if you're running this version of the Finder, use the Gestalt function (see below).

Examples

```
// Copy the selected Finder icons to the folder "For Review"
// First determine if the proper Finder version is running
If Gestalt "fndr", 3
      FinderCopy "Mac HD:For Review:"
Else
      Beep
      Message "Requires newer version of Finder."
End If

// Copy the file "Status Report" on the desktop to the folder "Backups"
FinderCopy "Mac HD:Desktop Folder:Status Report", "Mac HD:Backups:"

// Copy all the items in the "Stuff to Post" folder to the "Items Posted" folder
FinderCopy File("Mac HD:Internet:Stuff to Post:").List, "Mac HD:Internet:Items Posted:"
```

See Also FinderMove (page 107), FindFolder (page 129)

FinderMove

Syntax FinderMove [*path-list*,] *destination*

Description FinderMove works just like FinderCopy, except it moves files instead of copying them. See the description of FinderCopy above.

Examples

```
// Move the selected Finder icons to the Trash
// First determine if the proper Finder version is running
If Gestalt "fndr", 3
      FinderMove FindFolder "trsh"
Else
      Beep
      Message "Requires newer version of Finder."
End If

// Move all PICT files in the folder "Downloads" to the folder "Pictures"
Variable theFile, theFileList, X
theFileList = File("Mac HD:Downloads:").List
For X = 1 to ListCount theFileList
      theFile = ListItems theFileList, X
      If File(theFile).Kind = "PICT"
            FinderMove theFile, "Mac HD:Pictures:"
      End If
End For
```

See Also FinderCopy (page 106), FindFolder (page 129)

For, Next For, Exit For, End For

Syntax

```
For index-variable = start To end
    statements
    [Next For]
    [Exit For]
End For
```

Description

Repeats statements between For and End For a number of times. When the script first enters the For…End For loop, *index-variable* is set to the value of *start*. Each time through the loop, *index-variable* is incremented by 1 and compared to the *end* value. If index-variable is greater than the end value, the loop terminates and execution continues with statements following End For. The total number of times the loop runs is *end* – *start* + 1.

You can use Next For to skip to the next iteration of the For loop, and you can use Exit For to prematurely exit the loop and continue executing the statements following End For.

For loops can be nested (you can have a For loop within a For loop).

Examples

```
Variable i
For i = 1 to 5
    Message i
End For

Variable j
For j = 15 to 20
    If j = 17
        Next For        // don't display a message for #17
    End If
    Message j
End For
```

See Also

Repeat, Next Repeat, Exit Repeat, End Repeat (page 113)

If, Else, Else If, End If

Syntax

```
If condition
      statements
[Else If condition
      statements]
[Else
      statements]
End If
```

Description

Causes the script to execute different statements depending on the value of *condition*, which is usually an expression that evaluates to true (non-zero) or false (zero). A numeric expression is true if it does not equal zero; a string expression is true if it does not equal the null string (“”).

If *condition* is true (is not equal to zero or the null string), the script continues following the If command. If it is false, all statements up to the next Else, Else If or End If are skipped. If there is an Else statement and the If condition is false, only those commands after the Else will be executed until the next End If.

If statements can be nested (there can be an If statement inside of an If statement).

Else If is a shortcut for multiple Else and If statements.

Examples

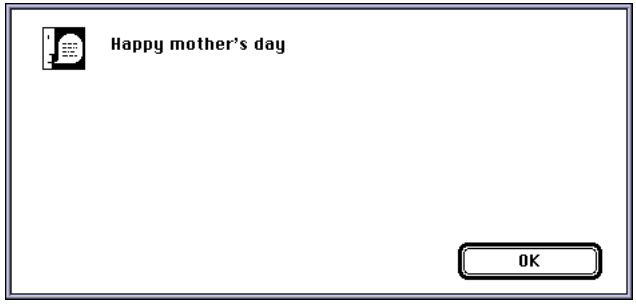
```
If LineVar < 100      // Only types LineVar if it is less than 100
      Type LineVar
End If

// Shows if FlagVar is or is not equal to zero. Could also use: If FlagVar <> 0
If FlagVar
      Type "Is not zero"
Else
      Type "Is zero"
End If

// The following types something different depending on what was chosen from the menu
TheMenu = PopupMenu "Red↵Green↵Blue"
If TheMenu = "Red"
      Type "Roses"
Else If TheMenu = "Green"
      Type "Grass"
Else If TheMenu = "Blue"
      Type "Ocean"
End If
```

Message

Syntax Message *string*

Description Displays string in a dialog box with an OK button. Use Message when you want to display a message on the screen while a script runs. The script stops and waits until you click the OK button, then closes the dialog box and resumes running.

Examples
```
Message "Happy mother's day"
Message (Substring "Macintosh", 1, 3)
```



Sample Message dialog box

See Also AskButton (page 123)

Open

Syntax Open *file-list [, application]*

Description Opens the specified application, document, control panel, desk accessory, or folder. (The Open command can open anything the Finder can open.) *File-list* is usually a full path or a list of paths. If you specify just a file name (not a full path), then Open assumes the file is in the current directory.

If you include the optional *application* parameter, OneClick opens the file using the specified application instead of the application used to create the file. (It's the same as opening the document by dropping it onto the application's icon.)

Note If there isn't enough memory to open an item, then the item does not open and the Error system variable is set to 1 (out of memory error).

Examples

```
// Open a single document
Open "Macintosh HD:Quicken 4 Folder:My Accounts"

// Open three documents in the current directory
Open "Status Report↵Month-End↵Budget"

// Open the File Sharing Monitor control panel
Open (FindFolder "ctrl") & "File Sharing Monitor"

// Open Picture 1 using Adobe Photoshop. Display a message if there's not enough memory
// to open Photoshop.
Open "Picture 1", "Adobe Photoshop"
If Error = 1
    Message "Not enough memory to open Photoshop."
End If
```

See Also Directory (page 148)

PaletteMenu

Syntax PaletteMenu

Description Shows the OneClick menu as a pop-up menu. This is the same menu as the one available in palette title bars, the menu bar, and the Apple menu.

If you've turned off the OneClick menu in the Apple menu and the menu bar and you've turned off the title bars on your palettes, you can use the PaletteMenu command to add the OneClick menu to a palette.

Examples

```
PaletteMenu
```

Pause

Syntax Pause *tenths*

Description Stops script execution for the specified period of time. Tenths is a number that specifies length of time to pause, expressed in tenths of a second.

Examples
```
Pause 5      // pauses for half a second (5/10)
Pause 600    // pauses for one minute
Pause 20     // pauses for two seconds
```

See Also Wait (page 120)

PopupPalette

Syntax PopupPalette *palette-name*

Description Displays the palette named *palette-name* as a pop-up palette. You can choose a button from the popped-up palette, or drag away from the palette to tear it off into a floating palette.

Examples
```
PopupPalette "Launcher"
PopupPalette "Calendar"
```

See Also PopupMenu (page 139)

QuicKey

Syntax QuicKey *QuicKey-name*

Description Plays the specified QuicKeys™ shortcut or shortcut sequence. To use this command, you must have QuicKeys (version 2.0 or later) from CE Software installed. You don't need to run CEIAC (from QuicKeys 2.0) or QuicKeys Toolbox (from QuicKeys 3.0).

Note The QuicKey command is an extension (external command). It's not available if the QuicKey Extension file is not installed in the Extensions folder inside the OneClick Folder (in Preferences).

Examples
```
QuicKey "QuickReference Card"
QuicKey "Mount Imaging Server"
```

Repeat, Next Repeat, Exit Repeat, End Repeat

Syntax

```
Repeat count
    statements
    [Next Repeat]
    [Exit Repeat]
End Repeat
```

Description

Repeats the script statements between the Repeat and End Repeat *count* number of times.

You can use Next Repeat to skip to the next iteration of the Repeat loop, and you can use Exit Repeat to prematurely exit the loop and continue executing the statements following End Repeat.

Examples

```
// Type 75 dashes
Repeat 75
    Type "–"
EndRepeat

// Get input five times. Play the Quack sound unless the input is "shut up"
Variable theText
Repeat 5
    theText = AskText "Type some text:"
    If theText = "shut up"
        Message "Okay!"
        Next Repeat
    End If
    Sound "Quack"
End Repeat
Message "All done"
```

See Also

For, Next For, Exit For, End For (page 108)

Schedule

Syntax Schedule *tenths* [, *run-always*]

Description Causes the script to automatically run its Scheduled handler every *tenths* tenths of a second. Use Schedule 0 to turn off the automatic execution of the script.

A script's Scheduled handler runs only if its palette is visible. To have the scheduled handler run even if the palette is hidden, include 1 (True) in the *run-always* parameter.

It's useful to have a Schedule command in a script's Startup handler so that the schedule starts as soon as the application starts. Otherwise, the schedule won't start until you click the button.

Examples

```
// Play the Quack sound every two seconds.
Schedule 20
On Scheduled
    Sound "Quack"
End Scheduled

// Check for new e-mail every ten minutes from the time the application starts up.
// Run even if the palette is hidden.
On Startup
    Schedule 6000, 1
End Startup
On Scheduled
    SelectMenu "Mail", "Check Mail"
End Scheduled
```

See Also Scheduled (page 199), Startup (page 200)

Scroll

Syntax Scroll [Page] Up | Down | Left | Right [, *window-specifier*]

Description Simulates clicking a window's scroll bar. Add the Page keyword to scroll a page at a time.

The Scroll command scrolls the active window by default. To scroll a different window, specify a window name or number in the optional *window-specifier* parameter.

Examples

```
// Scroll down one line
Scroll Down

// Scroll up one page
Scroll Page Up

// Click the right scroll arrow on the horizontal scroll bar
Scroll Right
```

SelectButton

Syntax SelectButton [Command] [Option] [Control] [Shift] [Check | Uncheck] *button-name*

Description Simulates clicking a button in a dialog box. *Button-name* is the name of the button to click. SelectButton lets you click regular pushbuttons, radio buttons, and checkboxes.

When SelectButton clicks a checkbox, the checkbox is toggled either on or off. Use the Check or Uncheck keywords to force a checkbox on or off. (SelectButton Check turns a checkbox on if it's off; SelectButton Uncheck turns a checkbox off if it's on.)

To simulate holding down a modifier key while clicking the button, add one or more of the following keywords in any order: Command, Option, Control, or Shift.

You can use wildcard characters to match button names. '?' matches a single character and '*' matches zero or more characters.

The '*' wildcard is useful for clicking buttons whose names end in an ellipsis (…) character: while most programs use a true ellipsis (Option-semicolon), some programs use three periods instead. If your script uses three periods when specifying a button name, then SelectButton won't find the button if its name ends in an ellipsis. Use the '*' wildcard to click a button without caring whether it has an ellipsis or three periods.

Note Some applications use non-standard button controls that may look like regular buttons. To click these kinds of buttons, use the Click command instead and specify the button's X and Y coordinates.

If you click a non-standard button control while recording a script, OneClick records the click as a Click statement instead of a SelectButton statement.

Examples

```
SelectButton "OK"
SelectButton "Cancel"

// Click the Generate... button
SelectButton "Genera*"

// Uncheck the Smooth Text checkbox if it's checked
SelectButton Uncheck "Smooth Text"
```

See Also DialogButton (page 167)

SelectMenu

Syntax SelectMenu [Check | Uncheck] *menu*, [, *menu*, ...] *item*

Description Selects *item* from the specified *menu*.

Menu can be either a menu name or number. Menu 1 is the first menu on the left (the Apple menu). Menu 2 is the second menu from the left (normally the File menu). Specifying a negative number goes from the right. Menu –1 is the rightmost menu (the Application menu).

SelectMenu also understands the following pseudo menu names for certain icon menus in the menu bar. You can use these menu names in place of menu numbers.

Menu	Pseudo menu name
Apple menu	[Apple]
OneClick menu	[OneClick]
Help (or Guide) menu	[Balloon]
Application menu	[Process]

Item can be either a menu item name or number. Item 1 is the first item in the menu. Dividing lines in the menu are included in the count. Negative numbers go from the bottom. Menu –1 is the last item in the menu.

To choose an item from a hierarchical menu, specify two or more menus (the path to the menu) before the menu item.

If you specify menu 0 (zero), OneClick searches through all of the application's menus (including hierarchical menus) to find the specified menu item. When you use 0 as the menu specifier, the menu item to search for must be a text string, not a number. This searching ability is great for global palette buttons that don't know in which menu the item may appear.

You can use wildcard characters to match menu and item names. '?' matches a single character and '*' matches zero or more characters.

The '*' wildcard is useful for choosing menu items that end in an ellipsis (…) character: while most programs use a true ellipsis (Option-semicolon), some programs use three periods instead. If your script uses three periods when specifying a menu item, then SelectMenu won't find the menu item if it ends in an ellipsis. Use the '*' wildcard to select a menu item without caring whether it has an ellipsis or three periods.

Note Some applications don't update their menus (enable, disable, check or uncheck menu items) until you click in the menu bar. Because OneClick selects menu items without clicking the menu bar, SelectMenu may not work correctly when it tries to choose a menu item that appears disabled, or choose an unchecked menu item that appears checked. To get around this problem, use Menu.Update to force the application to update its menus.

Examples

```
SelectMenu "File", "Print"
SelectMenu "Edit", "Copy"
SelectMenu 3, 4              // selects Copy from the Edit menu
SelectMenu –1, "Finder"      // selects Finder from the Application menu
SelectMenu "Window", –1      // selects the last window from the Window menu

// Force the application to update the checkmarks in its menus
Menu.Update
// Choose Plain Text from the Style menu only if it's not already checked in the menu
SelectMenu Check "Plain Text"

// Search every menu for the Bold command
SelectMenu 0, "Bold"

// Choose Definitions... from the Color submenu in the View menu
// It doesn't matter if Definitions... ends in an ellipsis or three periods
SelectMenu "View", "Color", "Defini*"
```

See Also SelectPopUp (page 118), Menu (page 173)

SelectPopUp

Syntax SelectPopUp [Global] *X*, *Y*, *item*

Description Chooses *item* from the pop-up menu at the coordinates *X* and *Y*. If the pop-up menu is in a window or dialog box, specify the menu's location using the window or dialog box's local coordinates. If the pop-up menu is somewhere else on the screen (not in a window or dialog box), use global coordinates and add the Global keyword.

Examples
```
SelectPopUp 28, 382, "Center"
SelectPopUp 66, 389, "Top of Right Page"
SelectPopUp 114, 390, "Column First"
```

See Also SelectMenu (page 116)

Set

Syntax Set *variable* = *value*

Description Stores a value in a variable.

The Set keyword is optional; you can omit it if you want.

Examples
```
Variable A, X
Set A = "Monday"
Set X = 5 * 7
```

See Also Variable (page 120)

Sound

Syntax Sound *sound-name*

Description Plays the specified sound. The sound must be stored in the System file or the active application. To see what sounds are available in the system and the active application, use the Sound submenu in the Script Editor's Parameters pop-up menu.

Examples
```
Sound "Quack"
```

See Also Beep (page 101)

Speak

Syntax Speak *text* [, *voice*]

Description Speaks the value of *text* (a string) using the default voice. The optional *voice* lets you specify which voice file to use when speaking. (Voices are stored in the Voices folder in the Extensions folder.)

The Speak command requires the Speech Manager extension, included with AV-capable computers and with System 7.5.

Note The Speak command is an extension (external command). It's not available if the OneClick Extensions file is not installed in the Extensions folder inside the OneClick Folder (in Preferences).

Examples

```
Speak "OneClick is the Killer App of the Nineties!"
Speak "I want my mother", "Princess"
```

Type

Syntax Type [Command] [Option] [Control] [Shift] *text* [, *text*, ...]

Description Types the value of *text* (a string) as if you had typed it from the keyboard. You can specify one or more text values to type.

To simulate holding down a modifier key, include one or more of the following keywords in any order: Command, Option, Control, or Shift.

Note When typing in menu key equivalents from a script, it's best to use lowercase letters instead of uppercase. Some third-party extensions that modify menu equivalent behavior do not expect an uppercase letter, because you don't usually hold down the Shift key when typing a menu equivalent.

Examples

```
Type "Hello there."

// types 28
Type 3 + 25

// types 2/20/97, 5:00 PM
Type Date, ", ", Time
```

```
// types Command-B
Type Command "b"

// type Command-S, then Command-P
Type Command "sp"
```

Variable

Syntax Variable [Global | Static] *variable-name* [, *variable-name*, ...]

Description Declares variables for use in a script or handler. Variables are local by default; add the Global or Static keywords to declare global or static variables. A variable can be static or global, but not both.

Variable names can contain only letters, numbers, and the underscore (_) character and must start with a letter. They can contain both upper and lowercase letters. (It's a good idea to begin variable names with a lowercase letter to differentiate them from other script keywords.) When a variable name consists of two or more words, you can improve the name's readability by capitalizing the first letter of each word or separating the words with underscore characters.

Examples

```
// Declare two local variables named X and Y
Variable X, Y

// Declare a local variable named dayName and two global variables named dayNum and dayVal
Variable dayName Global dayNum dayVal

// Declare a static variable named lastRunDate
Variable Static lastRunDate
```

See Also Variables (page 50)

Wait

Syntax Wait *expression*

Description Stops script execution until expression evaluates to true (non-zero). Execution continues with the statement following Wait after the expression becomes true.

You can continue using the computer while the script waits.

Examples

```
// Waits until you click the mouse, then displays a message box
Wait IsMouseDown
Message "All done"

// Waits until you press any key
Wait IsKeyDown

// Waits until the time is 12:15 PM, then plays a sound
Wait (Time 0) = "12:15 PM"
Sound "Quack"
```

See Also Pause (page 112)

While, Next While, Exit While, End While

Syntax

```
While condition
    statements
    [Next While]
    [Exit While]
End While
```

Description Executes all of the statements between While and EndWhile while *condition* is true (non-zero). When *condition* becomes false, execution continues with the statement following End While. If *condition* is false the first time it's tested, then the loop is skipped.

You can use Next While to skip to the next iteration of the While loop, and you can use Exit While to prematurely exit the loop and continue executing the statements following End While.

Examples

```
// Type 1 to 10 separated by commas
Index = 1
While Index <= 10
      Type Index & ", "
      Index = Index + 1
End While

// Type 1 to 10 separated by commas, skipping number 5
Index = 1
While Index <= 10
      If Index = 5
            Next While
      End If
      Type Index & ", "
      Index = Index + 1
End While
```

See Also Repeat, Next Repeat, Exit Repeat, End Repeat (page 113); For, Next For, Exit For, End For (page 108); Repeating statements while a condition is true (page 62)

With

Syntax

```
With object-specifier
      .property = value
End With
```

Description Lets you omit the object specifier when getting or setting the values of many of the same object's properties. Use the With command to make a script more readable and save yourself some typing.

When you specify a property without an object, the object is assumed to be the same object specified in the With statement. You can access other objects and properties within the With statement as long as you specify the object you want to access.

Examples

```
With Button("Program Launcher")
      .Width = 40
      .Height = 22
      .Color = 43                  // light purple
      .Text = "Launcher"
      Palette.Name = .Name         // sets the palette name equal to the button name
      .Mode = 0                    // sets the button appearance to Normal
End With
```

See Also Objects (page 65)

Functions

See "Functions" on page 49 for a general description of how to use functions in scripts.

Absolute

Syntax Absolute *number*

Description Returns the absolute value of *number*.

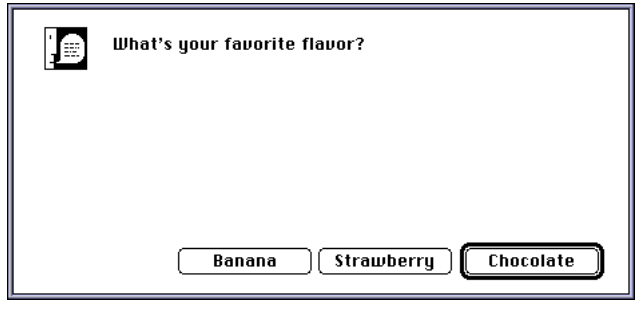
Examples
```
// Each statement types the value 25
Type Absolute –25
Type Absolute 25
```

AskButton

Syntax AskButton [*message*] [*button1* [*button2*...]]

Description Displays a dialog box with the specified *message* and lets you click one of up to four buttons. If you don't specify any buttons, the dialog box has a single OK button. The AskButton function returns the number (1–4) of the button clicked.

Examples
```
Variable theAnswer
theAnswer = AskButton "What's your favorite flavor?", "Chocolate", "Strawberry", "Banana"
// Types 1, 2, or 3 depending on the button clicked (1=Chocoloate, 2= Strawberry, 3=Banana)
Type theAnswer
```



See Also Message (page 110), AskList (page 125)

AskFile

Syntax AskFile [*type-list*]

Description Displays a directory dialog box and returns the full path to the selected file or folder. If you click Cancel in the dialog box, AskFile returns the empty string (“”).

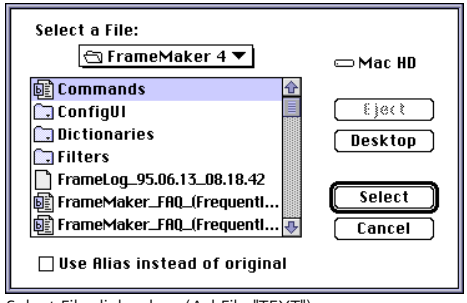
Type-list is a list of four-character file type codes, such as “TEXT”, “PICT”, “WDBN”, or “APPL”. To permit choosing a folder, use the pseudo file type “fold”. If you omit “fold”, the dialog box permits the selection of a file only, not a folder.

To set the default directory that appears in the dialog box, set the Directory system variable to the desired path before calling AskFile.

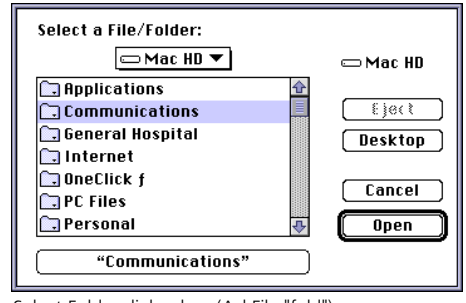
In the AskFile dialog box, check the “Use Alias instead of original” checkbox to return the path to an alias instead of the path to the file or folder the alias refers to.

Examples

```
Variable theFile, theFolder
theFolder = AskFile "fold"
theFile = AskFile "TEXT"
```



Select File dialog box (AskFile "TEXT")



Select Folder dialog box (AskFile "fold")

See Also Directory (page 148)

AskList

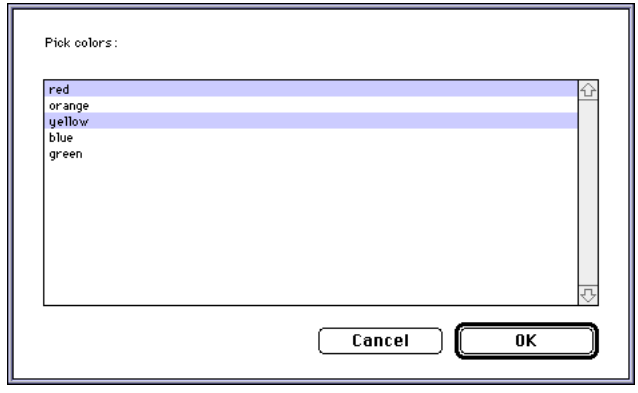
Syntax AskList *item-list* [, *prompt*] [, *selected-list*]

Description Displays a message (*prompt*) and a list of items (*item-list*) in a dialog box and returns a list containing the selected items. If you want one or more items in the list to appear selected by default, include the list of selected items in *selected-list*.

When the dialog box appears, click an item in the list to select it. To select more than one item, hold down the Command key and click additional items. To select a contiguous group of items, hold down the Shift key and click the first and last items in the group.

Examples

```
Variable theResponse
// Show a list of three colors with red and yellow already selected
theResponse = AskList "red↵orange↵yellow↵blue↵green", "Pick colors:", "red↵yellow"
Type "You picked:" & Return & theResponse
```



Sample AskList dialog box

See Also AskText (page 126), AskButton (page 123), AskFile (page 124)

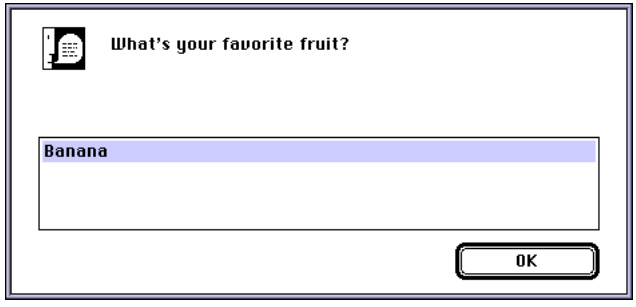
AskText

Syntax AskText [*string*] [, *default-value*]

Description Displays a dialog box with an edit box and prompts you to type a line of text. When you click OK, the function returns the typed text. The optional ***string*** is the prompt in the dialog box. The second parameter, ***default-value***, is an optional default string that appears pre-entered in the edit box.

Examples

```
// Assigns whatever is typed to Result. The dialog box says: Type something
Variable Result, favoriteFruit
Result = AskText "Type something"

// The default response is Banana
favoriteFruit = AskText "What's your favorite fruit?", "Banana"
Message "You typed " & Result & " and " & favoriteFruit
```



Sample AskText dialog box

See Also Message (page 110), AskButton (page 123), AskList (page 125), AskFile (page 124)

Char

Syntax Char *ascii-code*

Description Returns a string containing the character specified by the ***ascii-code*** number. This is the opposite of the Code function.

Examples

```
// Type the capital letter A
Type Char 65
```

```
// Presses the Return key
Type Char 13
```

See Also Code (page 127)

Code

Syntax Code *text*

Description Returns (as a number) the ASCII code of the first character in *text*. This is the opposite of the Char function.

Examples

```
// Types 97
Type Code "a"

// Types 65
Type Code "Apple"
```

See Also Char (page 126)

Date

Syntax Date [*format*]

Description Returns the current date as a string. The optional *format* specifies which of several date formats to use. If you don't specify a format, Date uses the default short format. The default types use the format specified by the Date and Time control panel.

You can use the Date command in the Script Editor's Parameters pop-up menu to choose different format settings and insert the proper format number in the script.

Format	Type	Example
0	Default short*	4/22/94
1	Default long*	Thursday, April 22, 1994
2	Default abbreviated*	Thu, Apr 22, 1994
3	Short MDY	4/22/94
4	Short DMY	22/4/94

*format may vary depending on the settings in the Date and Time control panel

Format	Type	Example
5	Short YMD	94/4/22
6	Abbreviated MDY	Apr 21, 1994
7	Abbreviated DMY	21 Apr, 1994
8	Long MDY	April 21, 1994
9	Long DMY	21 April, 1994
10	Abbreviated WMDY	Thu, Apr 21, 1994
11	Abbreviated WDMY	Thu, 21 Apr, 1994
12	Long WMDY	Thursday, April 21, 1994
13	Long WDMY	Thursday, 21 April, 1994
+16	Include leading zeros	04/03/94, April 03, 1994

The following apply only to the short type:

+32	Include century	4/22/1994
+0	Use '/' separator	4/22/94
+64	Use '-' separator	4-22-94
+128	Use '.' separator	4.22.94
+192	Use space separator	4 22 94

*format may vary depending on the settings in the Date and Time control panel

Examples Assume it's Tuesday, September 26, 1995.

```
Type Date          Types: 9/26/95
Type Date 0        Same as above
Type Date 1        Types: September 26, 1995
Type Date 3        Types: 9/26/95
Type Date 19       Types: 09/26/95
Type Date 179      Types: 09.26.1995
Type Date 8        Types: September 2, 1995
Type Date 26       Types: Tue, Sep 26, 1995
```

See Also Time (page 142)

Find

Syntax Find *find-text*, *in-text*

Description Returns the character position of *find-text* in *in-text*. If *find-text* is not found, Find returns zero (false).

Examples

```
// Types 5
Type Find "is", "Now is the time"

// Types 0
Type Find "and", "Now is the time"
```

See Also Replace (page 141)

FindFolder

Syntax FindFolder *folder-code*

Description Returns the full path to the specified folder. *Folder-code* is a four-character folder abbreviation.

Use the FindFolder function instead of typing the paths for special folders such as Desktop Folder, System Folder, or Trash in your scripts. Not only might it save you some typing, but it also allows your scripts to work with non-English versions of the system software. (Folders have different names in different languages.) Also, future versions of the system software may put the folders in different locations; FindFolder will still be able to determine the correct path to the folder given the folder's abbreviation.

Following are the folder codes to use:

Code	Folder name	Code	Folder name
“macs”	System Folder	“font”	Fonts
“desk”	Desktop Folder	“fonD”	Fonts (disabled)
“trsh”	Trash	“pref”	Preferences
“amnu”	Apple Menu Items	“prnt”	PrintMonitor Documents
“amnD”	Apple Menu Items (disabled)	“shdf”	Shutdown Items
“ctrl”	Control Panels	“shdD”	Shutdown Items (disabled)
“ctrD”	Control Panels (disabled)	“strt”	Startup Items
“sdev”	Control Strip Modules	“strD”	Startup Items (disabled)
“extn”	Extensions	“macD”	System Extensions (disabled)
“extD”	Extensions (disabled)		

Items in the Desktop Folder appear on the desktop.

Note Not all System versions support the folder codes listed here. (This table lists only the folder codes supported in System 7.5.1.) Later versions of system software may support additional folder codes; FindFolder will automatically support the new codes. To find out which folder codes your System version supports, open the System file with ResEdit and view the “fld#” resource.

Examples

```
// Make an alias of the selected icon, then move the alias to the Apple Menu Items folder.
// After choosing Make Alias, the new alias is automatically selected and gets moved.
SelectMenu "File", "Make Alias"
FinderMove FindFolder "amnu"

// Store the name of the startup disk in the global variable HD.
// FindFolder "macs" returns a path to the System folder.
// The startup disk name is the first item in the path.
Variable Global HD
ListDelimiter = ":"
HD = ListItems FindFolder "macs", 1
```

Gestalt

Syntax Gestalt *selector* [, *bit*]

Description Returns information about the hardware and system software configuration. You can use Gestalt to find out if a particular hardware or software component is available.

Selector is a four-character string that identifies the category of information you want to retrieve. Use the optional *bit* specifier (0–31) to get an individual bit in the result code as a True (1) or False (0) value.

Gestalt is based on the Macintosh toolbox call of the same name. This function is intended for use by programmers and advanced scripters.

Examples

```
If (Gestalt "kbd ") = 4
	Message "You are using an Extended keyboard"
Else
	Message "You are not using an Extended keyboard"
End If

If Gestalt "ascr", 0
	Message "AppleScript is available."
End If

If Gestalt "drag", 0
	Message "Drag and Drop support is available."
End If

If Gestalt "fndr", 3
	Message "OSL Compliant Finder is running."
End If
```

See Also *Inside Macintosh*, Volume VI, pages 3-38 to 3-53

GetDragAndDrop

Syntax GetDragAndDrop [*item*]

Description Returns a list of paths of icons dropped on the button. If you drop more than one icon on the button, you can get individual paths from the list using the optional *item* (the index number of the dropped item).

If you drag and drop text on the button, GetDragAndDrop returns the dropped text as a string.

You must use the GetDragAndDrop function within a DragAndDrop handler to retrieved the dropped information. You cannot drop icons or text on a button that doesn't have a DragAndDrop handler.

Examples

```
// Changes the dropped text to all uppercase and pastes it back into the
// application, replacing the current selection (the dropped text)
On DragAndDrop
    // save the current contents of the Clipboard
    Variable tempClip
    tempClip = Clipboard
    // change dropped text to uppercase and put it on the Clipboard
    Clipboard = Upper GetDragAndDrop
    // paste the uppercase text (replacing the dropped text) and restore the Clipboard
    SelectMenu "Edit", "Paste"
    Clipboard = tempClip
End DragAndDrop
```

```
// Changes the creator of all dropped TEXT and PICT files to "ttxt" (SimpleText).
// This causes SimpleText to open the TEXT or PICT file when you double-click the file's icon.
On DragAndDrop
    Variable fileCount, index, theFile
    // get the number of files dropped
    fileCount = ListCount GetDragAndDrop
    For index = 1 to fileCount
        // get the path of the dropped file from the list of dropped files
        theFile = GetDragAndDrop index
        If (File(theFile).Kind = "TEXT") OR (File(theFile).Kind = "PICT")
            File(theFile).Creator = "ttxt"
        End If
    End For
End DragAndDrop
```

See Also Using Drag and Drop (page 85), DragAndDrop (page 197)

GetResources

Syntax GetResources *resource-type*

Description Returns a list of names of all available resources of the specified type. *Resource-type* is a 4-character resource type (the type must be **exactly** 4 characters). Suggested types are “FONT”, “DRVR”, and “snd ” (note the trailing space).

Examples

```
// Types the names of all available fonts.
Type GetResources "FONT"

// Types the names of all items in the Apple menu.
Type GetResources "DRVR"

// Shows a pop-up menu of available sounds and plays the selected sound.
Sound PopupMenu GetResources "snd "
```

Length

Syntax Length *text*

Description Returns the number of characters in *text*.

Examples

```
// Types: 6
Type Length "Banana"

// Types: 0
Type Length ""
```

ListCount

Syntax ListCount *list*

Description Returns the number of items in the list.

Examples

```
// Types: 5
Type ListCount "Red↵Orange↵Yellow↵Green↵Blue↵Purple"

// Types the number of fonts in the Font menu
Type ListCount Menu("Font").List
```

ListItems

Syntax ListItems *list*, *start* [, *end*]

Description Returns a portion of the specified *list*. *Start* indicates the list item to start from and *end* indicates the last list item. If *end* is not supplied, ListItems returns the single list item at *start*. If *Start* and *end* are negative numbers, then ListItems returns items from the end of the list instead of the beginning.

Examples

```
// Types: Green, Blue, Yellow (all on separate lines)
Type ListItems "Red↵Green↵Blue↵Yellow↵Brown", 2, 4

// Types: Blue, Yellow, Brown (all on separate lines)
Type ListItems "Red↵Green↵Blue↵Yellow↵Brown", –3, –1

// Assuming the SystemFolder path is "Mac HD:System Folder", types "Mac HD"
ListDelimiter = ":"
Type ListItems SystemFolder, 1
```

See Also ListDelimiter (page 152)

ListSort

Syntax ListSort *list*

Description Sorts all list items in alphabetical order.

Examples

```
// Types: Blue, Green, Red (all on separate lines)
Type ListSort "Red↵Green↵Blue"

// Lets you choose a window from a list of all windows sorted alphabetically,
// then makes the chosen window the active (front) window.
Variable theChoice
theChoice = PopupMenu ListSort Window.List
Window(theChoice).Front
```

ListSum

Syntax ListSum *list*

Description Returns the sum of all the numbers in the list.

Examples

```
// Types:30
Type ListSum "10↵35↵–15"

// Sums all the currently selected numbers and types the
// sum on the line after the last number
SelectMenu "Edit", "Copy"
// Move to the next line
Type "→↵"
// Type the sum of all the numbers on the Clipboard
Type ListSum Clipboard
```

Lower

Syntax Lower *text*

Description Returns *text* with all letters changed to lowercase.

Examples

```
// Types "this is it."
Type Lower "This is IT."
```

See Also Upper (page 143), Proper (page 140)

MakeNumber

Syntax MakeNumber *text*

Description Returns *text* as a numeric value. This is the opposite of MakeText.

OneClick normally converts a string value to a number when the value is passed to a command that expects a numeric parameter. However, some commands (such as SelectMenu) can accept both string and numeric parameters; the value is interpreted differently depending on whether it is a string or a number. In cases like these, you'll need to use MakeNumber to force the command to interpret the string value as a number.

When converting text to a number, MakeNumber observes the following rules:

- any leading spaces are ignored
- the number can have a leading '–' or '+'
- there cannot be a space after the '–' or '+' or between the digits
- conversion stops when any non-numeric characters are encountered

Following are some examples of how MakeNumber converts text to numbers.

Text	Number	Text	Number	Text	Number
"12"	12	"Twelve"	0	"12–42"	12
"12 point"	12	"–42"	–42	"1 2 3"	1

Examples

```
Variable theMenu
theMenu = AskText "Type a menu number:"

// AskText returns a string value, so we'll convert it to a number before
// passing it to the Menu object. For example, if we type 3 in the AskText
// dialog box and pass that value to the Menu object, Menu will think
// we're referring to the menu named "3" instead of the 3rd menu in the menu bar.
theMenu = MakeNumber theMenu

// Types a list of all the menu items for the specified menu
Type Menu(theMenu).List
```

See Also MakeText (page 136)

MakeText

Syntax MakeText *number*

Description Returns *number* as a string value. This is the opposite of MakeNumber.

OneClick normally converts a numeric value to a string when the value is passed to a command that expects a string parameter. However, some commands (such as SelectMenu) can accept both string and numeric parameters; the value is interpreted differently depending on whether it is a string or a number. In cases like these, you'll need to use MakeText to force the command to interpret the numeric value as a string.

Examples

```
Variable theWindow
theWindow = 3

// Selects the 3rd item from the Window menu
SelectMenu "Window", theWindow

// Selects an item named "3" from the Window menu
SelectMenu "Window", MakeText theWindow
```

See Also MakeNumber (page 135)

PopupFiles

Syntax PopupFiles [*folder* [, *file-type-list*]

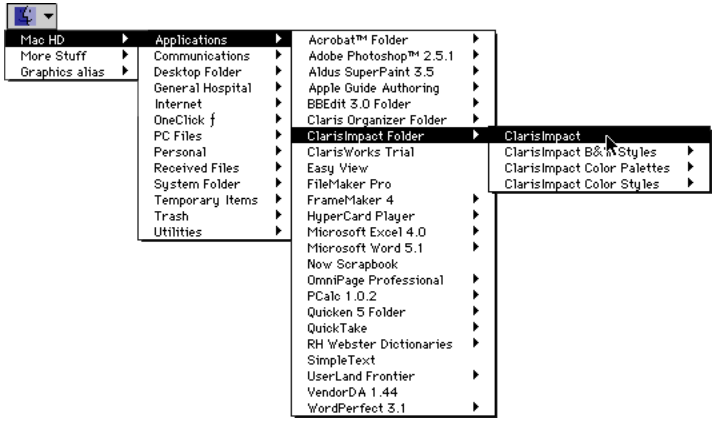
Description Pops up a hierarchical menu of all the files, folders, and volumes on the desktop and returns the full path to the chosen file or folder.

Folder is a path to a volume or folder; if you specify *folder*, the pop-up menu shows the files and folders in *folder* and lists higher-level folders and volumes at the bottom of the menu, below the separator line.

File-type-list is a list of four-character file type codes, such as “TEXT”, “PICT”, and “WDBN”. If you specify *file-type-list*, PopupFiles lists only folders and files of the specified types.

Examples

```
// Choose a Text or Microsoft Word file from within the Data folder, then open the chosen file
Open PopupFiles "Mac HD:Data", "TEXT↵JWDBN"

// Choose a file or folder from any volume and open it
Open PopupFiles
```



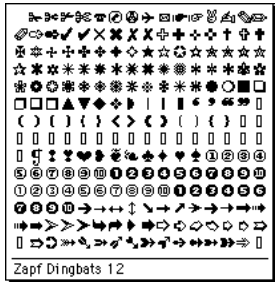
Sample PopupFiles menu

See Also PopupMenu (page 139)

PopupFont

Syntax PopupFont [*font-name* | *font-id* [, *size*]]

Description Displays a pop-up menu of all the characters in a font and returns the chosen character as a one-character string.



Sample PopupFont menu

You can specify a font by its name or ID number and optionally specify the font size (in points). If you don't specify a font, the font defaults to Geneva; if you don't specify a size, the size defaults to 14.

Note The PopupFont function is an extension (external function). It's not available if the OneClick Extensions file is not installed in the Extensions folder inside the OneClick Folder (in Preferences).

PopupFont works only in a MouseDown handler.

Examples

```
Type PopupFont "Times"
Type PopupFont "Symbol", 12
Type PopupFont Menu("Font").Checked, MakeNumber Menu("Size").Checked
```

PopupMenu

Syntax PopupMenu *menu-list* [, *checked-item*, ...]

Description Returns the item selected from a pop-up menu. The *menu-list* parameter is the list of text items that you want to appear in the menu; each item in the list is separated by the Return (↵) delimiter.

To include a divider line in the menu, include a hyphen (-) as an item in the menu list.

To indicate that a menu item is disabled (is gray in the menu and cannot be selected), start the item's name with a tilde (~).

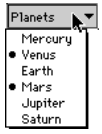
The *checked-item* parameter is the name or number of an item to appear checked in the menu. You can include multiple *checked-item* parameters to check more than one menu item.

Examples

```
// Types the selected item from the Red, Green, Blue menu.
Type PopupMenu "Red↵Green↵Blue"
```


```
// Types the selected item from Red and Blue. Green is disabled.
Type PopupMenu "Red↵~Green↵Blue"
```

```
// Choose a planet from the pop-up menu. Venus (item #2) and Mars are checked.
Variable theChoice
theChoice = PopupMenu "Mercury↵Venus↵Earth↵Mars↵Jupiter↵Saturn", 2, "Mars"
```




```
// Shows a pop-up menu of the Font menu.
SelectMenu "Font", PopupMenu Menu("Font").List
```

See Also PopupFiles (page 137)

Proper

Syntax Proper *text*

Description Returns *text* with each word capitalized.

Examples

```
// Displays "Sunday, Monday, Tuesday" in a message box
Message Proper "sunday, monday, tuesday"
```

See Also Lower (page 135), Upper (page 143)

Random

Syntax Random [*value*]

Description Returns a random number. If *value* is supplied, the number is in the range of 1 to *value*. If *value* is not supplied, the range is 1 to 65536.

Examples

```
// Types a number between 1 to 65536 inclusive
Type Random

// Types a number between 1 to 10 inclusive
Type Random 10
```

Replace

Syntax Replace *find-text*, *in-text*, *replace-text [, replace-all]*

Description Returns the string procured by finding *find-text* in *in-text* and replacing it with *replace-text*. If *find-text* is not found, it returns *in-text* unchanged.

The function replaces only the first occurrence of *find-text*, not all occurrences. To replace all occurrences, pass 1 (True) in the optional *replace-all* parameter.

Examples

```
// Types "Snippy"
Type Replace "oo", "Snoopy", "ip"

// Types "Hellothere"
Type Replace " ", "Hello there", ""

// Types "Middiddippi"
Type Replace "iss", "Mississippi", "idd", 1
```

See Also Find (page 129)

Return

Syntax Return

Description Returns the carriage return character (the character generated by pressing the Return key).

Examples

```
// Types three blank lines
Type Return Return Return
```

See Also Tab (page 142)

SubString

Syntax SubString *string*, *start* [, *end*]

Description Returns a portion of string. *Start* indicates the position of the character to start from and *end* indicates the position of the last character. If *end* is not supplied, SubString returns the single character at *start*. Both *start* and *end* may be negative numbers indicating their position from the end of the string.

Examples

```
// Types: is
Type SubString "This is it.", 6,7

// Types: SS
Type SubString "Annuities (SS)", –3, –2

// Types: e
Type SubString "Hello", 2
```

See Also ListItems (page 134)

Tab

Syntax Tab

Description Returns the tab character (the character generated by pressing the Tab key).

Examples

```
// Insert three tab characters at the current cursor position
Type Tab Tab Tab
```

See Also Return (page 141)

Time

Syntax Time [*format*]

Description Returns the current time as a string. The optional *format* specifies which of several time formats to use. If you don't specify a format, Time uses the default short format. The default time types use the format specified by the Date and Time control panel.

You can use the Time command in the Script Editor's Parameters pop-up menu to choose different format settings and insert the proper format number in the script.

Format	Type	Example
0	Default	1:07 PM
1	Default with seconds*	1:07:45 PM
2	12 hour	1:07 PM

*format may vary depending on the settings in the Date and Time control panel

Format	Type	Example
3	12 hour with seconds	1:07:45 PM
4	24 hour	13:07
5	24 hour with seconds	13:07:45
+16	Include leading zeros before hour	01:07 PM

*format may vary depending on the settings in the Date and Time control panel

Examples Assume it's 4:09:13 PM.

```
Type Time          Types: 4:09 PM (format may vary)
Type Time 18       Types: 04:09:13 PM
Type Time 5        Types: 16:09:13
```

See Also Date (page 127)

Trim

Syntax Trim *text*

Description Returns *text* with extra spaces removed. Extra spaces are spaces at the beginning or at the end of the string or more than one space in a row. Trim removes spaces only, not Tab or Return characters.

Examples

```
// Types "Alan L. Bird"
Type Trim "    Alan L.      Bird  "
```

Upper

Syntax Upper *text*

Description Returns *text* with all letters changed to uppercase.

Examples

```
// Types "THIS IS IT."
Type Upper "This is IT."
```

See Also Lower (page 135), Proper (page 140)

System Variables

See "System variables" on page 54 for a general description of how to use system variables in scripts.

ASResult

Description Contains the AppleScript result variable.

Every time an AppleScript statement is executed, the returned information is put into the AppleScript variable "result." If no information is returned, the result variable is set to empty. The ASResult system variable lets you retrieve the value of AppleScript's result variable.

Examples

```
// Display a message box showing the name of Scriptable Text Editor's front window
AppleScript
	– puts the window name in result
	get the name of window 1 of application "Scriptable Text Editor"
End AppleScript
Message ASResult
```

See Also Integrating OneClick and AppleScript (page 210), AppleScript (page 100)

BeepLevel

Description Returns the current alert sound volume, or sets the alert volume to a new value. Changing the BeepLevel volume affects only the default beep sound, not the volume of other system sounds. It's the same as adjusting the volume in the Alert Sounds panel of the Sound control panel (version 8.0 or later).

The alert sound level can be zero (no sound) to 7 (highest sound level).

Changing BeepLevel works only when running System 7.5 or Sound Manager 3.0 or later. Under earlier Sound Manager versions, BeepLevel is the same as SoundLevel.

Examples

```
// If the Option key is held down, turn off the alert sound, otherwise set the volume to 3.
If OptionKey
    BeepLevel = 0
Else
    BeepLevel = 3
End If
Beep
```

See Also SoundLevel (page 155), Beep (page 101), Sound (page 118)

Clipboard

Description Returns the contents of the Clipboard, or puts the specified value on the Clipboard.

Use the Clipboard system variable when you want to store the Clipboard's contents in a variable for later use, or when you want to manipulate data on the Clipboard and later paste it back into the same application or another application.

You can store any type of Clipboard data in a variable, including plain or styled text, graphics, spreadsheet cells, and other data types. If you want to manipulate the Clipboard's contents, however, you can change only the plain text on the Clipboard. You can use EasyScript's string- and list-handling commands to manipulate Clipboard text as you would do with regular string variables.

Note If the Clipboard variable doesn't contain the data you expect after you copy something to the Clipboard in an application, you may need to use the ConvertClip command (see page 103).

Examples

```
// Types the Clipboard contents. A slow paste.
Type Clipboard

// Puts My Name on the Clipboard as plain text.
Clipboard = "My Name"

// Copy this script to several buttons to make multiple Clipboards. Because the script doesn't
// actually manipulate the Clipboard data (it just gets data from and puts data onto the
// Clipboard), it works with any type of data on the Clipboard (text, pictures, and so on).
Variable Static clipContents
// If the button is Option-clicked, copy the selection and put it in a static variable.
If OptionKey
    SelectMenu "Edit", "Copy"
    clipContents = Clipboard
Else
```

```
// When clicked without the Option key, put the contents back on the Clipboard and paste.
    Clipboard = clipContents
    SelectMenu "Edit", "Paste"
End If
```

See Also Accessing the Clipboard (page 80), ConvertClip (page 103)

CommandKey

Description Returns True (1) if the Command key was held down when the button was clicked to run the script. You cannot set this system variable.

Use CommandKey, ControlKey, OptionKey, and ShiftKey to create buttons that perform different actions based on the modifier key (or keys) that are pressed when you click the button.

Examples

```
// If the Command key was pressed, select SuperScript, otherwise select SubScript
If CommandKey
    SelectMenu "Style", "SuperScript"
Else
    SelectMenu "Style", "SubScript"
End If
```

See Also ControlKey (page 146), OptionKey (page 154), ShiftKey (page 154)

ControlKey

Description Returns True (1) if the Control key was down when the button was clicked to run the script. You cannot set this system variable.

Use CommandKey, ControlKey, OptionKey, and ShiftKey to create buttons that perform different actions based on the modifier key (or keys) that are pressed when you click the button.

Examples

```
// If the Control key was pressed, dial GEnie, otherwise display the dialing directory.
If ControlKey
    SelectMenu "Dial", "GEnie"
Else
    SelectMenu "Dial", "Directory..."
End If
```

See Also CommandKey (page 146), OptionKey (page 154), ShiftKey (page 154)

Cursor

Description Returns the ID number of the cursor (mouse pointer). You cannot set this system variable.

Use Cursor to determine which cursor is active. It's useful in conjunction with the Wait command: have a script start some long process that causes the watch cursor (⌚) to appear, then stop and wait for the standard arrow cursor (↖) to appear before continuing. You can use the Cursor submenu in the Script Editor's Parameters menu to see the ID numbers of the cursors in the active application. The ID number of the standard system arrow cursor is always –1.

In most applications, the cursor changes to the standard system arrow when you move the cursor over a palette. In some applications, however, the cursor doesn't change to the arrow, especially if the cursor is animated and the application doesn't give any time to background processes. If you want a script to reliably check for the system arrow cursor, don't move the mouse over a palette while the script runs.

Examples

```
// Sign on to America Online, then wait until we're signed on before opening Stuffit Deluxe
Open "Mac HD:Communications:America Online v2.5.1:America Online v2.5.1"
Wait (Window.Name = "Welcome")
SelectButton "Sign On"
Wait (Cursor = –1)
Open "Mac HD:Utilities:Compression:Stuffit Deluxe"
```

Dialogs

Description Enables or disables display of dialog boxes while a script runs. False (0) means don't show dialog boxes, True (non-zero) means show dialog boxes (the default).

Set Dialogs to 0 (False) to keep dialog boxes from flashing up on the screen while the script runs. It makes script execution look smooth and clean. The script can still type information and click buttons in dialog boxes, even when they're not visible.

Only certain types of dialog boxes are affected by setting the Dialogs system variable. Specifically, only modal dialog boxes and alert boxes are hidden. Movable dialog boxes (both modal and non-modal), windows, and windows disguised as dialog boxes are not hidden.

When a script ends or is cancelled with Command-period, Dialogs is automatically set back to True. You don't need to explicitly set Dialogs to True at the end of the script. (If Dialogs was left False, then you wouldn't be able to see any dialog boxes while working in your applications.)

Examples

```
// Check the Substitute Fonts checkbox in Page Setup Options, but don't show the two dialogs
Dialogs = 0
SelectMenu "File", "Page Setup..."
SelectButton "Options"
If NOT DialogButton("Substitute Fonts").Checked
      SelectButton "Substitute Fonts"
End If
SelectButton "OK"
SelectButton "OK"
Dialogs = 1
```

Directory

Description Returns the directory where file open and save operations start from, or sets the current directory for file open and save operations.

Set the Directory system variable to the path of the folder you want to appear in Open and Save As dialog boxes.

Examples

```
// Save the current directory, then set the directory to where we want to save a file.
Variable saveDir
saveDir = Directory
Directory = "Mac HD:Data:"
// Select Save As from the File menu, type a file name, then click Save and Replace.
SelectMenu "File", "Save As..."
Type "My Notes File"
SelectButton "Save"
SelectButton "Replace"
// Set the directory back to the previous directory.
Directory = saveDir
```

Error

Description Contains a run-time error number if an error occurred in the last statement executed, or contains 0 (zero) if there was no error. For example, SelectMenu sets the Error variable to 2 (Not Found) if it wasn't able to find the specified menu or menu item.

Normally when a run-time error occurs in a script, the offending statement is skipped and execution continues with the next statement. If you plan to share your scripts with others, it's a wise idea to anticipate and check for possible errors that could occur while the script runs.

For example, assume your hard disk is named "Mac HD" and you wrote a script that opens a file on the hard disk, then does some processing on the file. The script uses the Open command and a full path, including the hard disk name, to open the file. If you give the script to your co-worker, whose hard disk is named "Centris," the script won't run correctly on his computer because the Open command won't be able to find the file (the path is different). The best place to check for this potential error is right after the Open command.

Most commands (including If) set or clear the Error variable after they run, so the value of Error is likely to change from one statement to the next. Because of this, you should always store Error in a temporary variable and then refer to the temporary variable, not Error, when dealing with the error condition.

Error 1: General Error (out of memory or resource problem)

AskButton	Unable to load dialog
Message	Unable to load dialog

Error 2: Not Found error

Button(*button-name*)	Button not found
Call	Button not found
DialogButton(*button-name*)	Button not found
File(*path*)	File, folder, or volume not found
Menu(*menu-name*)	Menu not found
Palette(*palette-name*)	Palette not found
Process(*process-name*)	Application not open
PopupPalette	Palette not found
Scroll	Window not found
SelectButton	Button not found

Error 2: Not Found error

SelectMenu	Menu not found
SelectPopUp	Menu not found
Sound	Sound not found
Volume(*volume-name*)	Volume not found
Window(*window-name*)	Window not found

Error 3: Parameter error (generally means missing parameter)

/ (division)	Divide by zero
Button.Help	Invalid help
Button.Icon	Invalid icon number or path
Button.Mode	Invalid mode
Button.Text	Invalid text
Find	Invalid string
GetResources	Invalid type
ListCount	Invalid list
ListItems	Invalid list
ListSort	Invalid list
ListSum	Invalid list
PopupMenu	Invalid menu values
PopupPalette	Invalid palette
Proper	Invalid string
Replace	Invalid string
Schedule	Invalid time
Trim	Invalid string

Examples

```
// Open the file "Status Report" on the desktop
// If the Open fails, show a message box explaining why the file wasn't opened.
// Declare a temporary variable for storing the error code.
Variable theProblem
Open "Mac HD:Desktop Folder:Status Report"
theProblem = Error
// If Error is non-zero, then some kind of error occurred.
If theProblem
    Message "An error occurred."
    If theProblem = 2
        // Error 2 is the generic "Not Found" error. Because we were trying to open a file,
        // we can deduce that a Not Found error means that a volume, folder, or file
        // wasn't found.
        // Now display a more descriptive message for the specific error that occurred.
        Message "Status Report: File or path not found."
    End If
    // Stop script execution if an error occurred. (Otherwise the script continues.)
    Exit
End If
```

See Also Testing and debugging a script (page 93), Checking for run-time errors (page 96)

IsKeyDown

Description Returns True (non-zero) when any key on the keyboard is pressed or held down. "Any key" includes all keys on the keyboard except the Power On key. As soon as the pressed key is released, IsKeyDown returns False (0). You cannot check to see which key was pressed.

You can use IsKeyDown to check to see if a key was pressed, then take some other action. Or, use the Wait command to wait for a keystroke, then continue. You may need to hold down a key slightly longer than you would for a simple keystroke to give IsKeyDown time to recognize that a key is down.

Note When Caps Lock is on, the Caps Lock key is considered pressed even if it's not physically held down.

Examples

```
// Wait for a keypress, then start quacking while the key is down.
// Stop quacking when the key is released. Show the status on the button.
Button.Text = "Waiting..."
Wait IsKeyDown
While IsKeyDown
      Sound "Quack"
End While
Button.Text = "Done"
```

See Also IsMouseDown (page 152)

IsMouseDown

Description Returns True (non-zero) if the mouse button is down (pressed), otherwise returns False.

You can use IsMouseDown to check to see if the mouse was clicked, then take some other action. Or, use the Wait command to wait for a mouse click, then continue. You may need to hold down the mouse button slightly longer than you would for a normal mouse click to give IsMouseDown time to recognize that the mouse button is down.

Examples

```
// Wait for the mouse button to be pressed, then start quacking while the button is down.
// Stop quacking when the button is released. Show the status on the button.
Button.Text = "Waiting..."
Wait IsMouseDown
While IsMouseDown
      Sound "Quack"
End While
Button.Text = "Done"
```

See Also IsKeyDown (page 151)

ListDelimiter

Description Returns the character used to separate items in a list, or sets the character that separates list items. The default ListDelimiter is the carriage return character (↵).

Set ListDelimiter to a different character to work with regular text strings as lists. For example, you could change ListDelimiter to a space to treat a sentence as a list of

words. Or change ListDelimiter to a colon (:) to treat a file's path as a list containing the volume, folder(s), and file name.

Note When you change ListDelimiter, all functions that work with lists (and object properties that return lists) use the new ListDelimiter value. If you change ListDelimiter to a colon or another character, remember to change it back to the default character (↵) before using a list delimited by carriage returns. The ListDelimiter value is reset to ↵ when the script ends or is cancelled with Command-period, so you don't need to explicitly change it back at the end of the script.

Examples

```
// Show a list box containing a list of all words in the sentence.
Variable X, theSentence
theSentence = "Monday is my favorite day of the week."
// Change ListDelimiter to a space (a space character separates words in the sentence).
ListDelimiter = " "
X = AskList theSentence

// Show a message box containing the volume name and file name of the chosen file.
// The first item in the path is the volume name, the last item is the file name
// A colon separates items in a path.
Variable thePath, volumeName, fileName
thePath = AskFile
ListDelimiter = ":"
volumeName = ListItems thePath, 1
fileName = ListItems thePath, –1
Message "You chose something named " & fileName & " on the disk named " & volumeName

// Type a list of sounds, separated by commas instead of carriage returns
ListDelimiter = ","
Type GetResources "snd "
```

See Also Manipulating lists (page 75)

OptionKey

Description Returns True (1) if the Option key was held down when the button was clicked to run the script. Cannot set this system variable.

Use CommandKey, ControlKey, OptionKey, and ShiftKey to create buttons that perform different actions based on the modifier key (or keys) that are pressed when you click the button.

Examples

```
// Normal click quits the active application, Option-click quits all open applications
Variable listOfApps, appCount, theApp, Index
If OptionKey
    // Get a list of running applications and the number of apps in the list
    listOfApps = Process.List
    appCount = Process.Count
    // Loop through all the active applications and quit all the apps not named Finder
    For Index = 1 to appCount
        theApp = ListItems appList, index
        If Process(theApp).Name <> "Finder"
            Process(theApp).Quit
        End If
    End For
Else
    // Normal (not Option) click: if the active app is something other than Finder, then quit it
    If Process.Name <> "Finder"
        Process.Quit
    End If
End If
```

See Also CommandKey (page 146), ControlKey (page 146), ShiftKey (page 154)

ShiftKey

Description Returns True (1) if the Shift key was held down when the button was clicked to run the script. Cannot set this system variable.

Use CommandKey, ControlKey, OptionKey, and ShiftKey to create buttons that perform different actions based on the modifier key (or keys) that are pressed when you click the button.

Examples

```
// When clicked, show a pop-up menu of apps. If Shift-clicked, open all apps in the menu.
Variable theAppList, netFolder, Index
netFolder = "Mac HD:Internet:"
theAppList = "Eudora↵NewsWatcher↵Netscape"
If ShiftKey
    For Index = 1 to ListCount theAppList
        Open netFolder & (ListItems theAppList, Index)
    End For
Else
    Open netFolder & (PopupMenu theAppList)
End If
```

See Also CommandKey (page 146), ControlKey (page 146), OptionKey (page 154)

SoundLevel

Description Returns the current system sound level (volume), or sets the sound level to a new value. It's the same as adjusting the volume in the Volumes panel of the Sound control panel (version 8.0 or later).

The sound level can be zero (no sound) to 7 (highest sound level).

Examples

```
// Save the current sound level and set it to the highest value before running Maelstrom.
// Option-clicking the button restores the previous sound level.
Variable Static savedVolume
If NOT OptionKey
    savedVolume = SoundLevel
    SoundLevel = 7
    Open "Mac HD:Games:Maelstrom 1.4.0"
Else
    SoundLevel = savedVolume
End If
```

See Also BeepLevel (page 144), Beep (page 101), Sound (page 118)

SystemFolder

Description Returns the path to the System Folder on the startup disk. You cannot set this system variable.

Using the SystemFolder variable is a shortcut to typing the actual path in your scripts (or using FindFolder "macs"). Using SystemFolder (or FindFolder) is recommended if you plan on sharing your scripts with others.

Examples

```
// Open the Views control panel
Open SystemFolder & "Control Panels:Views"
```

See Also FindFolder (page 129)

Ticks

Description Returns the number of ticks (1/60th of a second) since the computer was started. You cannot set this system variable.

Use Ticks to measure time intervals in 1/60ths of a second.

Examples

```
// Quack for 10 seconds
Variable saveTicks
saveTicks = Ticks
While (Ticks – saveTicks < 600)       // 60 ticks per second
      Sound "Quack"
End While
```

See Also Pause (page 112), Time (page 142), Wait (page 120)

Objects

This section describes all the objects, properties, and messages you can use in scripts. Except where noted, the syntax for accessing and changing a property is the same for all properties and objects.

▶ **To retrieve the value of a property:**

value = Object[*(specifier)*].Property

▶ **To change the value of a property:**

Object[*(specifier)*].Property = *value*

For general information on how to use objects, specifiers, properties, and messages, see "Objects" on page 65.

Button

Description A Button object is a button on a OneClick palette. The Button properties let you access or change nearly all the properties of a button that you normally set in the Button Editor. You can also access or change a button's script using the .Script property, and you can create and delete buttons using the .New and .Delete messages.

You can specify a button either by name or by number. Specifying by name lets you perform an operation on a specific button that you already know the name of. Specifying by number lets you loop through all the buttons on a palette and perform operations on each of them in sequence. Buttons are numbered 1 to N, where N is the total number of buttons on the palette. The numbering sequence is the same as the creation order of the buttons (1 being the oldest, N being the newest).

Button(–1) refers to the button currently under the cursor. A scheduled script could check to see which button is under the cursor, if any, and perform some action (such as display a help message) for the button you're pointing to.

If you omit the specifier part of the Button object, then the object is assumed to be the button that was clicked (the button containing the running script). Also, OneClick assumes the specified button is on the same palette as the button containing the

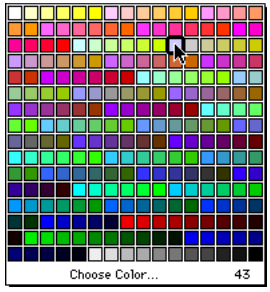
running script. To specify a button on a different palette, specify a palette object using this format:

Palette(*palette-specifier*).Button(*button-specifier*).Property

.Border Gets or sets the button's border style. There are 12 border styles numbered 0 through 11 which correspond to the choices in the Border pop-up menu:

Setting the .Border value to 3 removes the border. To make the button appear without a face or border (so only the icon or text appears), set .Border to 3 and .Color to 0.

.Color Gets or sets the button's color. Colors are numbered 1–256. To determine a color's number, drag the mouse over a color in a Color pop-up menu (in the Button Editor) and look at the number in the bottom-right corner of the menu:



Setting the color to 0 removes the button's color and makes the button transparent, allowing the palette's background to show through (the same as unchecking the Color checkbox in the Button Editor). Setting the button's .Color to a value 1–256 also checks the Color checkbox in the Button Editor.

.Count Gets the total number of buttons on the palette (a shortcut for using ListCount Button.List). No button specifier is necessary. This property is read-only.

```
// Loop through all the buttons and change them to a light purple color.
Variable Index
For Index = 1 to Button.Count
    Button(Index).Color = 43
End For
```

.Delete The .Delete message permanently deletes the specified button from the palette. The button specifier is the name of the button to delete. If you don't specify a button, the button containing the script is deleted and the script stops running. To delete a button on another palette, include a palette specifier before the button specifier.

```
// Delete the button named "Tile Windows"
Button("Tile Windows").Delete

// Delete the button named "Open Text Editor" on the palette named "Tools"
Palette("Tools").Button("Open Text Editor").Delete

// Delete all the buttons on the palette named "Cool Buttons"
Variable Index, numButtons
numButtons = Palette("Cool Buttons").Button.Count
For Index = 1 to numButtons
    Palette("Cool Buttons").Button(Index).Delete
End For
```

.Exists Returns True (1) if the specified button exists, otherwise False (0). You can use this property to determine if a button exists before performing some other action on the button.

```
// If the button named "Switcher" exists, then change its color to red, otherwise
// show an error message
If Button("Switcher").Exists
    Button("Switcher").Color = 36
Else
    Message "Can't find the Switcher button"
End If

// If the button named "Current Task" doesn't already exist, then create it
If NOT Button("Current Task").Exists
    Button("Current Task").New
End If
```

.Height Gets or sets the button's height. You can set the height to 1 to draw a button as a horizontal line.

```
// Change the height of the button named Quicken to 22
Button("Quicken").Height = 22
```

.Help Gets or sets the button's Balloon Help message. There is no limit on the length of the help message.

```
Button("Quicken").Help = "To open Quicken, click here."
```

.Icon Gets or sets the button's icon. Specify a number 1–4 to set the button's icon to one of the four stored icons. Setting .Icon to 0 causes no icon to appear on the button. (The icon isn't deleted from the button; it just doesn't appear.)

```
// Set the button's icon to icon #1
Button.Icon = 1

// Remove the button's icon
Button.Icon = 0
```

You can also copy an icon from one button to another button using the following syntax:

Button.Icon = *icon-number*, Button(*button-specifier*).Icon

The statement copies the currently visible icon from the specified button to the current button. *Icon-number* is a number 1–4 that specifies which icon you want to copy to in the current button.

```
// Copy the icon currently appearing on the Chooser button to this button's icon #2
Button.Icon = 2, Button("Chooser").Icon
```

You can copy a 16- or 32-pixel icon from a file's Finder icon to a button's icon using this syntax:

Button.Icon = *icon-number, path* [, *icon-size*]

Path is a path to an application, document, folder, or other Finder item. If the file you're copying an icon from has a custom icon, OneClick copes the custom icon, not the file's original icon. *Icon-size* indicates whether you want to copy the small icon (16) or large icon (32). The default is the small icon if you don't specify *icon-size*.

```
// Set the button's icon #1 to the SimpleText application's 32-pixel icon
Button.Icon = 1, "Mac HD:Applications:SimpleText", 32
```

.IconAlign Gets or sets the alignment of an icon on the button. The numbers 0 and 5-12 correspond to the 9 positions on the Position grid in the Button Editor. (These values are the same for the .TextAlign property, except .IconAlign doesn't use values 1–4.)

.Left Gets or sets the button's horizontal location on the palette. The left side of the palette is coordinate 0.

```
// Move the button to the left edge of the palette
Button.Left = 0
```

.List Gets a list of the names of all the buttons on the palette. Specify a palette object to get a list of button names from another palette. No button specifier is necessary. This property is read-only.

```
// Display a list box containing the names all the buttons on the palette
Variable theSelection
theSelection = AskList Button.List
```

```
// Display a list box containing the names of all the buttons on the palette named "Cool Tools"
Variable theSelection
theSelection = AskList Palette("Cool Tools").Button.List
```

.Location Changes the button's location on the palette. The .Location property requires two parameters (left and top) and is write-only. Using .Location is the same as using .Left and .Top, except it redraws the button only once instead of twice.

```
// Move the Quicken button to the upper-left corner of the palette
// Leave a 2-pixel margin between the edges of the button and the palette
Button("Quicken").Location = 2, 2
```

.Mode Gets or sets the appearance of the button. .Mode is a number 0–7 that corresponds to a setting in the Appearance pop-up menu in the Button Editor: A button's default appearance is Normal (0).

0: Normal	1: Pushed	2: Disabled
3: Inverted	4: Lighter	5: Darker
6: Pushed/Darker	7: Disabled/Lighter	

```
// Set the button's appearance according to the appearance of a menu command
// If Bold is dimmed in the Style menu, then make the button lighter and disabled
If NOT Menu("Style", "Bold").Enabled
      Button.Mode = 7
// If Bold is checked, make the button look pushed in and darker
Else If Menu("Style", "Bold").Checked
      Button.Mode = 6
// Bold isn't checked or dimmed, so make the button look normal
Else
      Button.Mode = 0
End If
```

.Name Gets or sets the button's name. A button's name is limited to 31 characters.

```
// Types a list of button names
Variable Index
For Index = 1 to Button.Count
      Type Button(Index).Name
End For

// Quacks if the current button's name is "Quack Button"
If Button.Name = "Quack"
      Sound "Quack"
End If

// Renames all the buttons to "Button" and a number
Variable Index
For Index = 1 to Button.Count
      Button(Index).Name = "Button" & MakeText Index
End For
```

.New The .New message creates a new button on the palette. The button specifier is the name of the new button. The button is created with all the default properties specified in the Button Editor, except the button is invisible. This lets your script

change other properties (size, location, text, icon, color, and so on) before making the button visible.

You can create a copy of an existing button by assigning another button to the new button using the following syntax:

Button(*button-name*).New = Button(*button-to-copy*)

To copy a button from another palette, add a palette specifier using this syntax:

Button(*button-name*).New = Palette(*palette-name*).Button(*button-to-copy*)

All the original button's properties (including its script) are copied to the new button. By creating new buttons in this manner, you don't need to copy all the individual properties one at a time from the original button to the new button.

```
// Make a new purple button named "Open Unread Mail" with the text "Unread" in the
// palette's upper-left corner, then turn the button on after all the properties have been set
Button("Open Unread Mail").New
With Button("Open Unread Mail")
    .Color = 43
    .Text = "Unread"
    .Location = 2, 2
    .Visible = 1
End With

// Create a copy of the button named "Toggle" and name it "Toggle 2"
Button("Toggle 2").New = Button("Toggle")
```

.Script Gets or sets a button's script. Use the .Script property to get a button's script as a string, or to assign a string containing a script to another button, replacing the existing script. Keep the following in mind when assigning a script to a button:

- When assigning a literal string to a button's .Script property, replace any quotation marks (") in the script text with single apostrophes ('). Use the Return symbol (↵ or <return>) to separate lines in the script text.
- If you assign a script to a button and the script contains a Startup handler, the current script stops and the Startup handler runs immediately.
- If you assign a script to the button containing the currently running script (replacing the active script), all script execution stops.
- Special characters in scripts, such as Return, arrow keys, and function keys, are converted to their text equivalents in angle brackets when you get the .Script

property of a button. You must use these same text equivalents when you assign a literal text string (or text from a text file) to a button's .Script property:

<return>	**<enter>**	**<tab>**
<esc>	**<delete>**	**<help>**
<fwddelete>	**<home>**	**<end>**
<pageup>	**<pagedown>**	**<leftarrow>**
<rightarrow>	**<uparrow>**	**<downarrow>**
<f1>	**<f2>**	**<f3>**
<f4>	**<f5>**	**<f6>**
<f7>	**<f8>**	**<f9>**
<f10>	**<f11>**	**<f12>**
<f13>	**<f14>**	**<f15>**

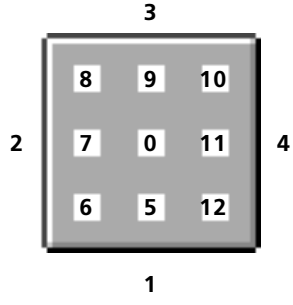
```
// Create three new buttons and assign scripts to them. The scripts assigned to each button
// are actually identical; only the method used to represent the Return character is different.
Button("New Button 1").New
Button("New Button 2").New
Button("New Button 3").New
Button("New Button 1").Script = "Message 'Hello there.'↵Sound 'Quack'"
Button("New Button 2").Script = "Message 'Hello there.'<return>Sound 'Quack'"
Button("New Button 3").Script = "Message 'Hello there.'" & Return & "Sound 'Quack'"

// Assign a script that types the Home and Down Arrow keys
Button("Another Button").Script = "Type '<home><downarrow>'"

// Copy the script from Button 1 to Button 2
Button("Button 2").Script = Button("Button 1").Script

// Take the script stored in the text file named "Launch Script" and assign it to a button
Button("New Button 1").Script = File("Mac HD:Launch Script").Text

// Store the Quicken button's script in a text file named My Quicken Script
File("Mac HD:My Quicken Script").Text = Button("Quicken").Script
```

.Size Changes the button's size. The .Size property requires two parameters (width and height) and is write-only. Using .Size is the same as using .Width and .Height, except it redraws the button only once instead of twice.

```
// Change the size of the Quicken button to 40 wide by 22 tall
Button("Quicken").Size = 40, 22
```

.Text Gets or sets the button's text (the label that appears on the button). There is no length limit on the button text. The text wraps inside the button if it's too long to fit on one line.

.TextAlign Gets or sets the alignment of text within or outside the button. The numbers 0–12 correspond to the 13 positions on the Position grid in the Button Editor.




```
// Place the button's text outside the right edge of the button
Button.TextAlign = 4
```

.TextColor Gets or sets the color of the button's text. Colors are numbered 1–256. See the Button.Color property for information on determining the number of a color.

.TextFont Gets or sets the font of the button's text. You can specify a font either by its name (as it appears in the Button Editor's font menu) or by its font ID number. The .TextFont property returns the font name.

```
// Set the button's font to Palatino 12
Button.TextFont = "Palatino"
Button.TextSize = 12

// Set the button's font to Courier (font ID 22)
Button.TextFont = 22
```

.TextSize Gets or sets the font size (in points) of the button's text.

.TextStyle Gets or sets the font style of the button's text. Add style numbers together to combine styles.

0: Plain Text	1: Bold	2: Italic	4: Underline
8: Outline	16: Shadow	32: Extend	64: Condense

```
// Set style to Plain Text (removes all other style attributes)
Button.TextStyle = 0

// Set style to Bold and Underline
Button.TextStyle = 5
```

.Top Gets or sets the button's vertical location on the palette. The top of the palette is coordinate 0.

```
// Move the button ten pixels down from the top of the palette
Button.Top = 10
```

.Update The .Update message tells OneClick to immediately redraw the specified button, instead of waiting until the script stops executing before redrawing. When a button is redrawn, its DrawButton handler is also called (if the handler exists).

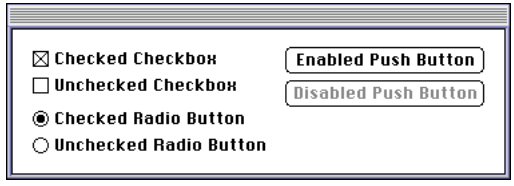
.Visible Gets or sets whether or not the button appears on the palette. This property corresponds to the Visible setting in the Appearance pop-up menu in the Button Editor. Set .Visible to 0 to hide a button or set .Visible to 1 to show it. All buttons are visible when the OneClick Editor window is open; invisible buttons don't disappear until you close the editor window.

.Width Gets or sets the button's width. You can set the width to 1 to draw a button as a vertical line.

```
// Set the width of the button named Quicken to 40, then set the width of
// the current button to match the width of the Quicken button.
Button("Quicken").Width = 40
Button.Width = Button("Quicken").Width
```

DialogButton

Description A DialogButton object is any push button, radio button, or checkbox in an application window or dialog box. You can tell whether a button is checked or enabled by looking at the button's .Checked or .Enabled property, then click the button (using SelectButton) if necessary to check or uncheck the button. You specify a DialogButton object using the button's name (such as "OK" or "Cancel") as the specifier.



In the picture above, the following statements describe the state of the buttons in the dialog box. Checkboxes and radio buttons can also be disabled, although they aren't pictured here.

```
DialogButton("Checked Checkbox").Checked = 1
DialogButton("Unchecked Checkbox").Checked = 0
DialogButton("Checked Radio Button").Checked = 1
DialogButton("Unchecked Radio Button").Checked = 0
DialogButton("Enabled Push Button").Enabled = 1
DialogButton("Disabled Push Button").Enabled = 0
```

You can use wildcard characters to match button names. '?' matches a single character and '*' matches zero or more characters.

All DialogButton properties are read-only.

.Checked Returns 1 if the specified button is checked, or 0 if it's unchecked.

```
// Open the Page Setup dialog, click Options, and uncheck Substitute Fonts if it's checked
SelectMenu "File", "Page Setup..."
SelectButton "Options"
If DialogButton("Substitute Fonts").Checked
      SelectButton "Substitute Fonts"      // click the checked button to uncheck it
End If
```

.Enabled

Returns 1 if the specified button is enabled (not dimmed), or 0 if it's dimmed.

```
// Display a message and stop the script if the "Subscribe" button isn't available,
// otherwise click the button and continue.
If NOT DialogButton("Subscribe").Enabled
    Message "The Subscribe button isn't available. (Probably because no edition is selected.)"
    Exit
Else
    // click the Subscribe button
    SelectButton "Subscribe"
    // set some subscriber options
    SelectMenu "Edit", "Publishing", "Subscriber Options..."
End If
```

.Exists

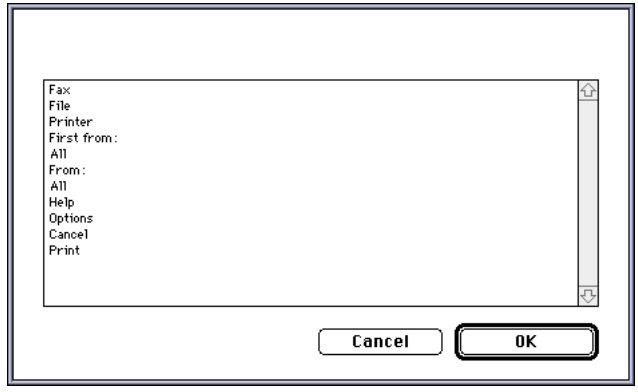
Returns True (1) if the specified dialog box button exists, otherwise False (0). Use this property to determine if a button exists before performing some other action.

```
// If the Cancel/Replace dialog box appears, then click the Replace button
If DialogButton("Replace").Exists
    SelectButton "Replace"
End If

// If it looks like an Open or Save dialog box is on the screen, then show the
// Directory Assistance palette, otherwise hide the palette
If DialogButton("Desktop").Exists AND DialogButton("Eject").Exists
    Palette("Directory Assistance").Visible = 1
Else
    Palette("Directory Assistance").Visible = 0
End If
```

.List

Returns an unsorted list of names on all the buttons in the frontmost window or dialog box.

```
// Displays a list box listing all the buttons in the Finder's Print dialog box.
Variable theResponse
SelectMenu –1, "Finder"
SelectMenu "File", "Print Window..."
// Display a list box and get a response. The chosen list items are put in theResponse.
theResponse = AskList DialogButton.List
```



AskList dialog box listing buttons from the Print dialog box

```
// Choose Save As, type a file name, click Save, and then click Replace if necessary
SelectMenu "File", "Save As..."
Type "Personal Assets 6/2/95"
SelectButton "Save"
// If the Replace/Cancel dialog box appears, then DialogButton.List will contain
// "Cancel↵Replace" and Find will return a non-zero value. Otherwise, DialogButton.List
// will contain nothing and Find will return 0, causing the SelectButton statement to be skipped.
If Find "Replace", DialogButton.List
      SelectButton "Replace"
End If
```

File

Description The File object lets you work with files and folders. You can get a list of all files in a folder, or just the files of a specified type; get or change a file's type and creator codes; and read and write information in a text file.

The specifier for a File object is the file or folder's path. When specifying a path to a volume, include a colon (:) at the end of the volume name.

.Creator Gets or sets the specified file's creator code. A creator code is a four-character code that indicates which application created the file; for example, the creator code for any SimpleText file is "ttxt" and the creator code for any Photoshop file is "8BIM". The Finder uses a file's creator code to figure out which icon to show in the Finder and which application to open when you double-click the icon.

If you use .Creator to change a file's creator code, the Finder does not immediately update the display of the file's icon in an open Finder window. Close and re-open the window to make the Finder show the correct icon.

```
// Change the creator code of all TEXT and PICT files in Mac HD:Data to "ttxt" (for SimpleText).
Variable index, theFile, fileList
fileList = File("Mac HD:Data:", "TEXT↵PICT").List
For index = 1 to ListCount fileList
    theFile = ListItems fileList, index
    File(theFile).Creator = "ttxt"
End For

// Change the creator code of the dropped TEXT files to "R*ch" (for BBEdit).
// Skip the file if its type is something other than TEXT.
On DragAndDrop
    Variable index, theFile, fileList
    fileList = GetDragAndDrop
    For index = 1 to ListCount fileList
        theFile = ListItems fileList, index
        If File(theFile).Kind = "TEXT"
            File(theFile).Creator = "R*ch"
        End If
    End For
End DragAndDrop
```

.Exists Returns True (1) if the specified file or folder exists, otherwise False (0). Use this property to determine if a file or folder exists before performing some other action on the file or folder.

```
// The path to the SimpleText application is stored in the static variable simpleTextPath.
// If the file can't be found, display a directory dialog so the user can locate SimpleText
// and store the new path. The While loop ensures that the user chooses a valid path to
// an application.
Variable Static simpleTextPath
While NOT File(simpleTextPath).Exists
    Message "SimpleText can't be found. Please locate it."
    simpleTextPath = AskFile "APPL"
End While
Open simpleTextPath
```

.Kind Gets or sets the specified file's file type code. A file type code is a four-character code that indicates the file's format; for example, the file type code for a text file is "TEXT" and the file type code for an application is "APPL". When you're not sure of a file's type

code, use the File Type command in the Script Editor's Parameter menu to choose a file and insert it's type code into the script.

The .Kind property returns the pseudo type "fold" if you specify a folder.

The most common use for the .Kind property is to get a file's type and do something with the file based on its type.

```
// Move all PICT files in the folder "Downloads" to the folder "Pictures"
Variable theFile, theFileList, X
Directory = "Mac HD:Downloads:"
theFileList = File.List
For X = 1 to ListCount theFileList
      theFile = ListItems theFileList, X
      If File(theFile).Kind = "PICT"
            FinderMove theFile, "Mac HD:Pictures:"
      End If
End For
```

Normally you won't want to change a file's type unless you know what you're doing. Changing a file's type does not translate the file's contents into another format: if you change a Microsoft Word document (type "WDBN") into a ClarisImpact report (type "iRpt"), you can then open the file in ClarisImpact, but the program won't understand the file's contents and will probably give an error message. The file still contains Microsoft Word data in the file, not ClarisImpact data.

.List Returns a list of files in the specified folder. A second file type specifier is optional: specify a list of file type codes to get a list that contains only files of the specified type(s).

Remember to use a colon at the end of the folder's path to indicate the path is a folder, not a file.

The .List property is read-only.

```
// Type a list of all the files in the folder Mac HD:Data.
Type File("Mac HD:Data:").List

// Type a list of the TEXT and PICT files in the folder Mac HD:Data.
Type File("Mac HD:Data:", "TEXT↵PICT").List
```

```
// Open all the TIFF documents in the folder Mac HD:Scans.
Variable theFile, theListOfFiles
theListOfFiles = File("Mac HD:Scans:", "TIFF").List     // put the list of TIFFs in theListOfFiles
For theFile = 1 to ListCount theListOfFiles              // loop through the list and open each file
      Open "Mac HD:Scans:" & (ListItems theListOfFiles, theFile)
End For
```

.NewFolder The NewFolder message creates a new folder at the specified path. A colon (:) at the end of the folder name is optional.

If OneClick can't find the volume or folder where the new folder should be created, then no folder is created.

```
// Create a new folder named Received Files in the current directory
File("Received Files").NewFolder

// Create a Documents folder inside the WordPerfect folder. The Applications and
// WordPerfect folders must already exist (only the last folder in the path gets created).
File("Mac HD:Applications:WordPerfect 3.1:Documents:").NewFolder

// Create a new, untitled folder on the desktop
File((FindFolder "desk") & "untitled").NewFolder
```

.Text Reads text from the specified file or writes text to a text file. You can read text (actually, the data fork) from any file and store it in a variable, allowing you to work with text in a file just like any other string value.

You can also write text to a text file by assigning a value to the file's .Text property. To prevent accidently overwriting a non-text file's data fork, you can only write text to a text file (a file whose type code is "TEXT"). No text is written if you try to write text to a non-text file.

If you write text to a file that doesn't exist, OneClick creates a new SimpleText file and writes the text to it. You can then change the type and creator codes, if you prefer, after the new file is created. If you create a new file in this manner, the folder containing the file (if specified) must exist or else OneClick won't create the file. OneClick will not create any non-existent folders in the file's path.

```
// Create a new file on the desktop called "My Text File" and put the text "Hello there" in it
// If "My Text File" already exists, the text in it is overwritten
File("Mac HD:Desktop Folder:My Text File").Text = "Hello there"
```

```
// Copy the text from "File A" to "File B", overwriting the text already in "File B"
File("File B").Text = File("File A").Text

// Append the contents of "Mac HD:File B" to "Mac HD:File A"
File("Mac HD:File A").Text = File("Mac HD:File A").Text & File("Mac HD:File B").Text

// Display a list box containing interesting information from the System file.
// This just shows the System file's data fork in the list. There will be some garbage in the text.
Variable theList, theResponse
theList = File(SystemFolder & "System").Text
theResponse = AskList theList

// Search all files in a directory for a text string, then display a list box showing only the files
// that contain the search string
Variable theDirectory, theTotalFileList, theFoundFileList, theResponse, theSearchString
theDirectory = AskFile "fold"                                    // choose the directory to search
theSearchString = AskText "Type a string to search for:" // get the string to find
theTotalFileList = File(theDirectory).List
For index = 1 to ListCount theTotalFileList
      theCurrentFile = theDirectory & (ListItems theTotalFileList, index)
      If Find theSearchString, File(theCurrentFile).Text
            theFoundFileList = theFoundFileList & theCurrentFile & "↵"
      End If
End For
theResponse = AskList theFoundFileList, "Search results:"
```

Menu

Description A Menu object is any menu or submenu in the menu bar. You can use a Menu object to determine if a menu item is checked or enabled, or to get a list of all the menu items in a specified menu. The .Checked and .Enabled properties work the same way as the DialogButton object's .Checked and .Enabled properties.

The specifier for a Menu object is a menu or menu item name. You can also specify a menu by number: 1 is the first menu in the menu bar (usually the Apple menu), 2 is the second menu (usually File), and so on. Use a negative number to specify a menu starting from the right side of the menu bar: –1 is the Application menu, –2 is the Help menu, and so on.

You can also specify menu items by number. Like menus in the menu bar, the first item in the menu is 1, the second is 2, and the last is –1. A divider line in the menu also counts as a menu item.

To specify a menu item in a menu, specify both the menu and menu item using this syntax:

Menu(*menu*, *menu-item*).Property

For example, to see if the Copy command in the Edit menu is enabled, you could use either of these statements:

```
If Menu("Edit", "Copy").Enabled       // check to see if Copy in the Edit menu is enabled
If Menu(3, 4).Enabled     // check to see if the 4th command in the 3rd menu is enabled
```

You can specify menu items in hierarchical menus using a similar syntax. Just include any submenu names in the path to the menu or menu item:

Menu(*menu*, *submenu*, *menu-item*)

For example, you could use the following to see if the Bold menu item is checked in the Style submenu of the Format menu:

```
If Menu("Format", "Style", "Bold").Checked
```

You can use wildcard characters to match menu or menu item names. '?' matches a single character and '*' matches zero or more characters.

All Menu properties are read-only.

.Checked Returns True (1) if the specified menu item is checked, or False (0) if it's unchecked.

```
// Switch between Body Pages and Master Pages. The current choice appears checked
// in the View menu; only one choice appears checked at a time.
Menu.Update         // force the application to update the checkmarks in its menus
If Menu("View", "Body Pages").Checked
      SelectMenu "View", "Master Pages"
Else
      SelectMenu "View", "Body Pages"
End If
```

.Enabled Returns True (1) if the specified menu or menu item is enabled (not dimmed), or False (0) if it's dimmed.

.Exists

Returns True (1) if the specified menu or menu item exists, otherwise False (0).

```
// Check to see if the Format menu exists before opening the Paragraph Designer.
// (The Format menu exists only if a document is open.) If the menu doesn't exist, then
// display a message box and exit.
If Menu("Format").Exists
    SelectMenu "Format", "Paragraphs", "Designer..."
Else
    Message "Can't open the Paragraph Designer. Perhaps no document is open."
    Exit
End If
```

.List

Returns an unsorted list of menu items in the specified menu or submenu. Use Menu.List without a specifier to get a list of the menus in the menu bar.

```
// Type a list of all the menus in the menu bar
Type Menu.List

// Type a list of all the commands in the File menu
Type Menu("File").List

// Type a list of all the commands in the Style submenu of the Format menu
Type Menu("Format", "Style").List

// See if the Palatino font is available in the Font menu
If NOT Find "Palatino", Menu("Font").List
    Message "You don't have Palatino installed."
    Exit
End If
```

.Name

Returns the name of the specified menu or menu item. Use .Name when you want to get the name of an icon menu that doesn't have a name, such as the Apple menu or Help menu. .Name returns a pseudo name if it knows what the menu is. If the specified menu already has a name, then .Name just returns the menu's name.

For this menu:	.Name returns:
Apple menu	[Apple]
OneClick menu	[OneClick]
Help (or Guide) menu	[Balloon]
Application menu	[Process]

If .Name is unable to determine a menu's name, then .Name may return garbage or nothing at all, depending on how the application defines its menu names for icon menus.

```
// Type the name of the Application menu
Type Menu(–1).Name
```

.Update

Forces the active application to update the status of checked, unchecked, enabled, and disabled items in its menus.

Some applications don't update their menus (enable, disable, check or uncheck menu items) until you click in the menu bar. Because OneClick accesses and selects menu items without clicking the menu bar, the SelectMenu command (and the .Checked and .Enabled properties of menu items) may not work correctly when the script tries to access a menu item that appears disabled. To get around this problem, use Menu.Update before a SelectMenu statement and before statements that access a menu item's .Checked or .Enabled property.

```
// Check the status of the Bold item in the Style menu, then set the button's icon appropriately
// Make sure the Style menu shows the correct status of the Bold item first
Menu.Update
If Menu("Style", "Bold").Checked
    Button.Icon = 1
Else
    Button.Icon = 2
End If
```

Palette

Description

A Palette object refers to any OneClick palette. You can use the Palette object to manipulate or get information about any of the global and application-specific palettes available in the active application.

The specifier for a Palette object is the name of the palette as it appears in the Palette Editor and on the palette's title bar.

You can also specify a palette by number, which is useful for looping through all the available palettes and performing some operation on each palette. Palette(–1) refers to the palette under the cursor, if any.

.Color Gets or sets the color of the palette's background. Colors are numbered 1–256; see the Button.Color property (page 158) for more information.

.Count Returns the total number of available palettes. The .Count property is a shortcut for ListCount Palette.List.

.Delete Permanently removes the specified palette. If no palette is specified, the palette containing the active script is deleted.

.Drag Causes the button to act like a title bar, letting you drag the palette around on the screen when you drag the button. The .Drag message works only in a MouseDown handler.


Drag button

To create a palette with a drag button (such as the palette at left), create a button with the following script. Then simply drag the button to move the palette.

```
On MouseDown
    Palette.Drag
End MouseDown
```

.Exists Returns True (1) if the specified palette exists, otherwise False (0). A palette exists if it's available in the OneClick menu; .Exists returns True whether or not the palette is actually visible.

An application-specific palette "exists" only if its application is open and active (meaning the palette appears in the OneClick menu).

```
// Display the Styles palette. If the palette can't be found, display a message box.
If Palette("Styles").Exists
    Palette("Styles").Visible = 1
Else
    Message "The Styles palette can't be found. Make sure its application is open and active."
End If
```

.Grow Causes the button to act like a size box (sometimes called a grow box), letting you resize the palette when you click and drag the button. The .Grow message works only in a MouseDown handler.

Resizing a palette with Palette.Grow does not automatically move the button containing the Palette.Grow script. The script should check the new height and width of the palette and move the button to the palette's new lower-right corner.

It's possible to create a very small palette by dragging the grow button past the palette's left or top edge, rendering the palette almost unusable. Therefore, it's a good idea to have the script check the height and width of the palette after the Palette.Grow statement and change the palette's height or width if the size is too small.

To create a palette with a grow box, create a button in the lower-right corner of the palette and put the following script in the button:

```
On MouseDown
    Palette.Grow
    // Reset the palette's height or width if it's too small (minimum size 30 x 20 pixels)
    If Palette.Width < 30
        Palette.Width = 30
    End If
    If Palette.Height < 20
        Palette.Height = 20
    End If
    // Move this button to the new lower-right corner of the palette
    Button.Location = (Palette.Width – Button.Width), (Palette.Height – Button.Height)
End MouseDown
```

.Height Gets or sets the palette's height. Setting .Height to zero (0) adjusts the height so that all buttons fit vertically within the palette. (This is similar to clicking Fit To Buttons in the Palette Editor, except only the height changes, not the width.)

```
// Toggle the palette between 40 pixels tall and the "Fit To Buttons" height
If Palette.Height = 40
    Palette.Height = 0
Else
    Palette.Height = 40
End If
```

.Left Gets or sets the palette's horizontal position on the screen.

```
// Move the palette to the left edge of the screen
Palette.Left = 0

// Move the palette to the right edge of the screen
Palette.Left = Screen.Width – Palette.Width
```

.List Returns a list of all available palettes. Use .List in a loop to cycle through all palettes and perform some operation on each palette.

```
// Show a pop-up menu of all available palettes
Variable theChoice
theChoice = PopupMenu Palette.List

// Turn on (show) all the available palettes
Variable X, palList, thePalette
palList = Palette.List
For X = 1 To Palette.Count
      thePalette = ListItems palList, X
      Palette(thePalette).Visible = 1
End For
```

.Location Changes the palette's location on the screen. The .Location property requires two parameters (left and top) and is write-only. Using .Location is the same as using .Left and .Top, except it redraws the palette only once instead of twice.

```
// Move the palette to 40 pixels down and 10 pixels from the left edge of the screen
Palette.Location = 10, 40
```

.New The .New message creates a new palette. The palette specifier is the name of the new palette. The palette is created with all the default properties specified in the Palette Editor, except the palette is hidden. This lets your script change other properties (size, location, color, and so on) and add new buttons before making the palette visible. Adding the optional Global keyword following .New lets you create a global palette. If you omit the Global keyword, then .New creates an application-specific palette for the active application.

You can create a copy of an existing palette by assigning another palette to the new palette using the following syntax:

Palette(*palette-name*).New = *palette-to-copy*

All the original palette's properties (including its buttons) are copied to the new palette, except the new palette isn't made visible. By creating new palettes in this manner, you don't need to copy all the properties and buttons one at a time from the original palette to the new palette.

To import a palette from a OneClick palette file, use this syntax:

Button(*button-name*).New = *palette-name*, *palette-file*

This is the same as importing a palette in the Palette Editor, except the new palette isn't made visible after it's imported.

```
// Create a new palette named "Communications"
Palette("Communications").New
// Change the new palette's size and location, then make it visible
Palette("Communications").Size = 100, 22
Palette("Communications").Location = 0, (Screen.Height – Palette.Height)
Palette("Communications").Visible = 1

// Create a new global palette named "Project Documents"
Palette("Project Documents").New Global
Palette("Project Documents").Visible = 1

// Copy the palette named "Launcher" to a new palette named "Launcher copy"
Palette("Launcher copy").New = "Launcher"
Palette("Launcher copy").Visible = 1

// Import the palette named "Welcome Screen" from the palette file "Screens"
Palette("My Screen").New = "Welcome Screen", "Mac HD:Extra Palettes:Screens"
Palette("My Screen").Visible = 1
```

.Size Changes the palette's size. The .Size property requires two parameters (width and height) and is write-only. Using .Size is the same as using .Width and .Height, except it redraws the palette only once instead of twice.

Using zero (0) for either the height or width parameters fits the palette to enclose the buttons (similar to clicking Fit To Buttons in the Palette Editor).

```
// Change the palette to 100 pixels wide by 22 pixels tall
Palette.Size = 100, 22

// Resize the palette to enclose all the buttons (same as "Fit To Buttons")
Palette.Size = 0, 0
```

.TitleBar Turns the palette's title bar on or off, or gets the palette's current title bar setting. Set .TitleBar to 1 to turn on the title bar or 0 (zero) to turn it off.

```
// Toggle the palette's title bar on or off
Palette.TitleBar = NOT Palette.TitleBar
```

.Top Gets or sets the palette's vertical location on the screen. The palette's top edge starts at the top of the palette's content area, not the top of its title bar. The title bar is 12 pixels tall.

```
// Move the palette to the top of the screen, just below the menu bar
// If the palette's title bar is turned on, move the palette 12 pixels higher
If Palette.TitleBar
      Palette.Top = 33
Else
      Palette.Top = 21
End If

// Move the palette to the bottom of the screen
Palette.Top = Screen.Height – Palette.Height
```

.Update Forces the palette to redraw itself and all its buttons. Use the .Update message in a script when you want OneClick to immediately redraw a palette after you change palette or button properties. (If you change several properties of a button or palette within a script, OneClick normally redraws the affected button or palette when the script ends, not after each individual property change.)

```
// Make the two buttons named "A" and "B" flash between red and green 10 times
Repeat 10
      Button("A").Color = 36    // red
      Button("B").Color = 226   // green
      Palette.Update
      Button("A").Color = 226
      Button("B").Color = 36
      Palette.Update
End Repeat
```

.Visible Shows or hides the specified palette, or gets the palette's current visible setting. Set .Visible to 1 to show the palette or 0 (zero) to hide it. Visible palettes have a bullet (•) next to their names in the OneClick menu.

```
// Turn on (show) all the available palettes
Variable X, palList, thePalette
palList = Palette.List
For X = 1 To Palette.Count
      thePalette = ListItems palList, X
      Palette(thePalette).Visible = 1
End For

// Assign a key shortcut to this script's button to toggle the palette on or off with a keystroke
Palette.Visible = NOT Palette.Visible
```

.Width Gets or sets the palette's width. Setting .Width to zero (0) adjusts the width so that all buttons fit horizontally within the palette. (This is similar to clicking Fit To Buttons in the Palette Editor, except only the width changes, not the height.)

```
// Toggle the palette between 100 pixels wide and the "Fit To Buttons" width
If Palette.Width = 100
    Palette.Width = 0
Else
    Palette.Width = 100
End If
```

Process

Description A process is any open, running application or desk accessory, including the Finder. A Process object lets you manipulate or get information about a running application.

The specifier for a Process object is the name of an application as it appears in the Application menu in the menu bar. If you don't specify an application, then Process refers to the active (front) application.

You can also specify a Process object by number. Process(1) is the active application, Process(2) is the previous application used, and so on. Process(–1) is a shortcut for referencing the Finder.

All Process properties (except for .Selection and .Visible) are read-only.

.Count Returns the total number of open applications. The .Count property is a shortcut for ListCount Process.List.

.Creator Returns the four-character creator code for the specified application. For example, if SimpleText is the active application, Process.Creator returns "ttxt".

.Exists Returns True (1) if the specified application is open, otherwise False (0).

```
// Switch to FileMaker Pro if it's open, otherwise display a message.
If Process("FileMaker Pro").Exists
    Process("FileMaker Pro").Front
Else
    Message "FileMaker Pro isn't running!"
End If
```

.Folder Returns a path to the folder containing the specified application.

```
// Get the path to the folder containing the SimpleText application
Variable theAppPath
theAppPath = Process("SimpleText").Folder
```

.Free Returns the amount of free memory (in bytes) in the specified application's memory partition. To determine the amount of free memory in K, divide the number of bytes by 1024. To determine the amount of memory currently in use by an application, subtract the number of free bytes (.Free) from the total number of bytes allocated (.Size).

```
// Display the amount of free memory (in kilobytes) for the active application.
Message Process.Name & " has " & (Process.Free / 1024) & "K free"
```

.Front Gets the name of the active application, or switches another application to the front.

```
// Display the name of the active application on the button
Button.Text = Process.Front
```

To set the active application, use .Front as a message.

```
// Switch to the Finder
Process("Finder").Front

// Switch back and forth between the two frontmost applications
Process(2).Front
```

.Kind Returns the four-character file type code for the specified application. The type code is usually "APPL" (for applications) except for the Finder, whose type code is "FNDR". Desk accessories have the type code "dfil".

.List Returns a list of all running applications. This list includes all applications that appear in the Application menu. Background-only processes (such as File Sharing) aren't included.

.Name

Returns the name of the specified application. The .Name property is useful if you want to get the name of a process specified by number.

```
// Get the name of the active application.
Variable theActiveApp
theActiveApp = Process.Name

// Type a list of all the active applications (same as Type Process.List)
Variable X
For X = 1 to Process.Count
      Type Process(X).Name, Return
End For
```

.Quit

Quits the specified application as if you had chosen Quit from the application's File menu. The .Quit message sends an Apple Event to the specified application, telling it to quit.

```
// Quit the active application.
Process.Quit

// Quit all running applications except the Finder.
Variable appList appCount theApp X
appCount = Process.Count
appList = Process.List
For X = 1 to appCount
      theApp = ListItems appList, X
      If Process(theApp).Name <> "Finder"
            Process(theApp).Quit
      End If
End For

// Force the Finder to quit.
Process("Finder").Quit
```

.Selection

Uses Apple Events to get or set the text of the current selection in the specified application. Setting an application's selection using the .Selection property is faster than typing text (using the Type command) or setting the Clipboard's contents and pasting. To the use .Selection property, however, the application must support the Apple Events required to get and set the current selection, and most applications do not currently support these events.

Microsoft Excel and the System 7.5 Finder do support these events. To determine if another application supports them, select something in the application and then run the following script:

```
Message Process.Selection
```

If the resulting message box is empty, chances are pretty good that the application doesn't support the events required to get or set the selection.

An application's response to .Selection is usually different depending on the type of data you work with in the application. For example, if you select the range of cells A3:B5 in a Microsoft Excel worksheet named "Budget," then Process.Selection returns the string "Budget!R3C1:R5C2"—*not* the contents of the selected cells. If you select a chart object in the worksheet, Process.Selection returns the name of the selected chart.

In the Finder, Process.Selection returns the full path of the selected icon, or a list of paths if more than one icon is selected.

```
// Get a list of paths of all the selected icons
Variable thePathList
thePathList = Process("Finder").Selection

// Open the Sharing window for the startup disk
Process("Finder").Selection = Volume.Name
// Give Finder time to select the icon before choosing the menu item
Wait (Process("Finder").Selection = Volume.Name)
SelectMenu "File", "Sharing..."
```

.Size Returns the total amount of memory (in bytes) allocated to the specified application. To determine the application's memory size in K, divide the number of bytes by 1024.

```
// Display the amount of memory used and total memory allocated in Kilobytes
Variable usedMemK, memSizeK, appName
appName = Process.Name
usedMemK = (Process.Size – Process.Free) / 1024
memSizeK = Process.Size / 1024
Message appName & " is using " & usedMem & "K out of the " & memSize & "K reserved for it."
```

.Visible Returns True (1) if the specified application is visible (showing on the screen) or False (0) if it's hidden. Setting .Visible to zero (0) hides the application as if you had chosen Hide from the Application menu; setting .Visible to 1 shows the application.

Unlike choosing Hide from the Application menu, hiding the active application (using Process.Visible = 0) does *not* bring another application to the front.

```
// Hide all applications except Finder, then switch to Finder
Variable X
For X = 1 to Process.Count
      If Process(X).Name <> "Finder"
            Process(X).Visible = 0
      End If
End For
Process("Finder").Front
```

Screen

Description The Screen object lets you get information about and set options for the monitor(s) connected to your Macintosh. If you have only one monitor, then the Screen object needs no specifier. If you have more than one monitor, then using Screen without a specifier refers to the main (menu bar) monitor, Screen(2) refers to the second monitor, and so on. Screens are numbered as they appear in the Monitors control panel.

.Color Gets or sets the Colors and Grays options in the Monitors control panel. Setting .Color to 1 switches the monitor to colors and setting .Color to 0 (zero) switches it to grays.

```
// Change the monitor to grayscale mode
Screen.Color = 0
```

.Count Returns the number of monitors connected to the computer.

```
If Screen.Count = 1
      Message "This is a standard configuration."
Else If Screen.Count = 2
      Message "You're a Power User."
Else If Screen.Count >= 3
      Message "Beware of electromagnetic radiation!"
End If
```

.Depth

Gets or sets the number of colors displayed on the monitor. The .Depth property uses a number (the bit depth) to determine the number of colors. The following table shows bit depth values and the corresponding settings in the Monitors control panel.

.Depth value	Monitors control panel setting
1	Black & White
2	4
4	16
8	256
16	Thousands
32	Millions

```
// Switch to millions of colors before opening Adobe Photoshop
Screen.Depth = 32
Open "Mac HD:Applications:Adobe Photoshop:Adobe Photoshop™ 2.5.1"
```

.Exists

Determines if the specified monitor exists. This property is useful in determining if more than one monitor is connected.

```
// Display a message if only one monitor is connected
If NOT Screen(2).Exists
    Message "Can't find a second monitor"
End If
```

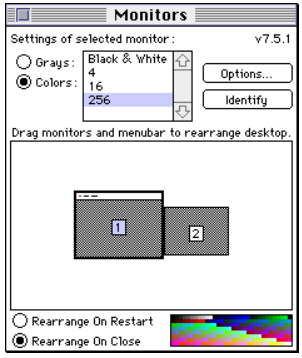
.Height

Returns the height (in pixels) of the specified monitor. Getting the screen's height is useful when you want to position palettes or windows at (or near) the bottom of the screen.

```
// Move the palette to the bottom of the screen
Palette.Top = Screen.Height – Palette.Height
```

.Left Returns the location of the left edge of the specified monitor relative to Screen(1), the main (menu bar) monitor. For single-monitor systems, .Left always returns 0. If you have a second monitor connected, Screen(2).Left returns the left edge of the second monitor relative to the left edge of the main monitor. For example, assume the following:

- you have two monitors
- the main monitor is 1152 pixels wide by 870 pixels tall
- the second monitor is 832 pixels wide by 624 pixels tall
- the second monitor is positioned to the right of the main monitor



In this setup, Screen(2).Left returns 1152 because the main monitor goes from 0 (on the left edge) to 1151 (on the right). If the second monitor was positioned to the *left* of the main monitor, then Screen(2).Left would return –832.

```
// Position the palette in the upper-right corner of the second monitor.
Palette.Left = Screen(2).Left + Screen(2).Width – Palette.Width
Palette.Top = Screen(2).Top + 12
```

.Top Returns the location of the top edge of the specified monitor relative to Screen(1), the main (menu bar) monitor. For single-monitor systems, .Top always returns 0. If you have a second monitor connected, Screen(2).Top returns the top edge of the second monitor relative to the top edge of the main monitor. (See the .Left property for an explanation of how this works.)

.Update Forces the Macintosh to redraw the entire contents of the screen, including the menu bar, the desktop, and all windows and palettes. Use Screen.Update when some other program malfunctions and leaves garbage on the desktop or in a window.

.Width Returns the width (in pixels) of the specified monitor. Getting the screen's width is useful when you want to position palettes or windows at (or near) the right edge of the screen.

```
// Move the palette to the right edge of the screen
Palette.Left = Screen.Width – Palette.Width
```

Volume

Description The Volume object lets you eject, unmount, or get information about any mounted disk—including hard disks, floppy disks, CD-ROMs, and file server volumes. You can specify a volume either by name or by number. Using Volume without a specifier refers to the startup disk. When specifying a volume by name, include a colon (:) at the end of the name to indicate the name is a path.

.Count Returns the total number of mounted volumes.

.Eject Ejects the specified volume, leaving its icon on the desktop. This is the same as choosing Eject Disk from the Finder's Special menu.

.Exists Returns 1 (True) if the specified volume is mounted, otherwise 0 (False).

```
If NOT Volume("Beavis").Exists
    // Open an alias to mount the server volume
    Open (FindFolder "amnu") & "Recent Servers:Beavis"
    Type "myPassword"
    SelectButton "OK"
End If
```

.Free Returns the number of free kilobytes (K) on the specified volume. Subtract .Free from .Size to get the total number of used kilobytes.

```
// Show a blue thermometer indicating the percentage of space used on the startup disk
Variable usedSpace percentUsed
usedSpace = Volume.Size – Volume.Free
percentUsed = (usedSpace * 100 / Volume.Size)
DrawIndicator percentUsed, 211
```

.List Returns a list of names of all mounted volumes. Each name has a colon (:) at the end to indicate the name is a path. Volumes in the list are in the order in which they were mounted; the first volume in the list is always the startup disk.

```
// Unmount the volume chosen from the pop-up menu
Variable theChoice
theChoice = PopupMenu Volume.List
Volume(theChoice).Unmount
```

.Name Returns the name of a volume specified by number.

```
// Store the name of the startup disk in global variable HD
Variable Global HD
HD = Volume.Name
```

.Size Returns the total size of the specified volume in kilobytes (K). Divide the size by 1024 to get the size in megabytes.

```
// Display a list box showing the sizes (in megabytes) of all mounted volumes
Variable volCount volList theVol sizeList X
volCount = Volume.Count
volList = Volume.List
For X = 1 To volCount
      theVol = ListItems volList, X
      sizeList = sizeList & theVol & " " & (Volume(theVol).Size / 1024) & " MB" & Return
End For
X = AskList sizeList, "Sizes of mounted volumes:"
```

.Unmount Unmounts the specified volume and removes its icon from the desktop. This is the same as dragging the volume to the Trash or choosing Put Away from the Finder's File menu.

```
// Unmount the volume named “PowerBook” that’s mounted via file sharing
Volume(“PowerBook”).Unmount
```

Window

Description The Window object lets you get information about or manipulate open windows on the screen. The specifier for a Window object is the name of the window as it appears in the window's title bar. If you don't specify a window name, then the object refers to the active (front) window. You can also specify a window by number; window 1 is the active window.

You can use wildcard characters to match window names. '?' matches a single character and '*' matches zero or more characters.

You can use the Window object to work with windows in the active application only, not in inactive applications. To work with windows in an inactive application, use Process.Front to make the application active first.

.Count Returns the total number of open windows in the active application. Use Window.Count as a shortcut for ListCount Window.List.

.Exists Returns True (1) if the specified window is open on the screen, otherwise False (0). Use .Exists to determine if a window is open before performing some other action that affects the window or its contents (such as moving the window or typing text into it).

```
// If the window named "untitled" isn't open, then open a new window
If NOT Window("untitled").Exists
      SelectMenu "File", "New"
End If
```

.Front Gets the name of the active window or switches to the specified window. Use Window.Front to switch windows from a script instead of using the Click command to click a window and make it active.

```
// Switch to the "Bookmarks" window if it's not already active
If Window.Front <> "Bookmarks"
      Window.Front = "Bookmarks"
End If
```

Note Applications that have their own floating palettes or tool bars (such as Adobe Photoshop, Microsoft Word and Excel) often consider one of the floating palettes to be the active window, *not* the active document window. (This applies only to an

application's built-in palettes, not OneClick palettes.) For example, if the active window is a Photoshop document, Window.Front usually returns "Tools" as the active window (the Photoshop tool palette). You can use Window.Kind to determine which windows are real document windows and which are palettes or other kinds of windows.

.Height

Gets or sets the height of the active window. Use .Height to resize a window vertically. The .Height property works only with the active window, so a window specifier is not necessary.

```
// Resize the active window to 100 pixels tall
Window.Height = 100
```

.Kind

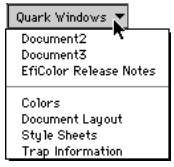
Returns an ID number identifying the kind of window specified. Applications that have different types of windows (document windows, moveable dialog boxes, tool bars, and so on) use a unique ID number for each style of window.

Use the .Kind property to determine which windows in Window.List are document windows and which are other special types of windows. By evaluating the .Kind property of each window in the list, you can create a new list that contains just the desired windows.

The following table shows the .Kind values for standard window styles.

Value	Window style
0	Document
1	Dialog box
2	Plain dialog box
3	Alternate dialog box (has a drop shadow)
4	Document with no size box
5	Moveable dialog box
8	Document with zoom box
12	Document with zoom box, no size box
16	Desk Accessory (rounded) window

```
// Displays the .Kind value for any window chosen from the pop-up menu.
// The menu lists all windows, including documents, moveable dialog boxes, tool bars, etc.
Variable theChoice
theChoice = PopupMenu Window.List
If theChoice <> ""
      Message Window(theChoice).Kind
End If
```




```
// Show a pop-up menu of document windows and floating palettes in QuarkXPress.
// Documents appear at the top of the menu, palettes are at the bottom.
// Quark's document windows have a .Kind value of 8 and palette windows all
// have a .Kind value greater than 16.
Variable WinList, theChoice, specialList, docList, theWindow, X
WinList = Window.List
For X = 1 To ListCount WinList
      theWindow = ListItems WinList, X
      If Window(theWindow).Kind > 16
            specialList = specialList & theWindow & Return
      Else If Window(theWindow).Kind = 8
            docList = docList & theWindow & Return
      End If
End For
WinList = (ListSort docList) & "-" & Return & (ListSort specialList)
theChoice = PopupMenu WinList
If theChoice <> ""
      Window(theChoice).Front
End If
```

Note For Macintosh programmers: The .Kind value is a combination of the window's variant and the resource ID of the window's WDEF. This is the same number that the application passes to the Macintosh Toolbox call "NewWindow" when it creates the window. Standard windows have a value from 0 to 16. Custom windows have a number that is 16 * WDEF Resource ID + window variant (a number greater than 16).

.Left Gets or sets the window's left edge on the screen. Use .Left to move a window horizontally.

```
// Move the window to the left edge of the screen
Window.Left = 0

// Move the window to the right edge of the screen.
Window.Left = Screen.Width – Window.Width
```

.List Returns a list of open windows in the active application. The window list includes regular windows; moveable dialog boxes; and the application's own special windows such as tool bars or floating palettes, if any (not OneClick palettes). Special windows that don't have a name (such as tool bars in some applications) aren't included in the window list.

```
// Display a pop-up menu of all open windows and switch to the window chosen from the menu
Variable theChoice
theChoice = PopupMenu Window.List
If theChoice <> ""
    Window.Front = theChoice
End If
```

.Location Sets the window's horizontal and vertical location on the screen. This property requires two parameters (X and Y coordinates) when specifying the window's location. The .Location property is read-only.

```
// Move the window named "Mac HD" to the left edge of the screen and 20 pixels down.
// (The menu bar is 20 pixels tall.)
Window("Mac HD").Location = 0, 20
```

.Size Changes the active window's size. This property requires two parameters (width and height) when specifying the size. The .Size property works only with the active window, so a window specifier is not necessary. This property works only if the window has a resize box.

The .Size property is write-only.

```
// Resize the window to 540 pixels wide by 400 pixels tall
Window.Size = 540, 400
```

.Top Gets or sets the window's top edge on the screen. Use .Top to move a window vertically .

```
// Move the active window to just below the menu bar
Window.Top= 20
```

```
// Cascade all open windows
Variable theWindow, winList, winCount, X
winList = Window.List
winCount = Window.Count
For X = 1 to winCount
      theWindow = ListItems winList, X
      Window(theWindow).Top = (20 * X) + 20
      Window(theWindow).Left = 20 * X
End For
```

.Update Forces the application to redraw the contents of the specified window.

.Visible Returns True (1) if the specified window is visible (showing on the screen) or False (0) if it's hidden. Setting .Visible to zero (0) hides the window; setting .Visible to 1 shows the window.

Making a window invisible does not close the window, it just hides it. To make an invisible window visible, you must specify it by name and not number. Invisible windows are not seen when specifying by number.

Caution Be careful when changing a window's .Visible property—some applications may not work correctly if you make their windows invisible.

.Width Gets or sets the width of the active window. Use .Width to resize a window horizontally. The .Width property works only with the active window, so a window specifier is not necessary.

```
// Resize the active window to 300 pixels wide
Window.Width = 300
```

.Zoom Gets or sets the zoom state of the specified window. When .Zoom = 1, the window is zoomed out to its full size; when .Zoom = 0, the window is not zoomed (the window's current size is different than its zoomed size). Setting .Zoom without a parameter toggles the window's zoom state.

The .Zoom property works only with the active window, so a window specifier is not necessary.

```
// Zoom the active window to its full size
Window.Zoom = 1

// Toggle the zoom state of the active window (same as clicking in the window's zoom box)
Window.Zoom

// Zoom all unzoomed windows
Variable winList, theWindow, X
winList = Window.List
For X = 1 to ListCount winList
      theWindow = ListItems winList, X
      Window(theWindow).Front
      If NOT Window(theWindow).Zoom
            Window(theWindow).Zoom = 1
      End If
End For
```

Handlers

See "Handlers" on page 72 for a general description of how to use handlers in scripts.

DragAndDrop

Description A script's DragAndDrop handler executes when you drag and drop a Finder icon or text clipping on the button. Use the GetDragAndDrop function to retrieve the text or the paths of the dropped icons. You cannot drop items on a button that doesn't contain a DragAndDrop handler.

Examples

```
// Move all the files dropped on the button to the "Briefcase" folder
On DragAndDrop
    FinderMove GetDragAndDrop, "Mac HD:Briefcase:"
End DragAndDrop
```

```
// Store the text dropped on the button in a static variable named theStoredClipping.
// The button's text label shows the contents of the text clipping.
// You can later click and drag from the button to insert the text in another document.
On DragAndDrop
    Variable Static theStoredClipping
    theStoredClipping = GetDragAndDrop
    Button.Text = theStoredClipping
End DragAndDrop
```

```
// To insert the stored clipping in a document, simply click the button and
// drag from the button to the document.
On MouseDown
    Variable Static theStoredClipping
    DragButton theStoredClipping
End MouseDown
```

See Also GetDragAndDrop (page 132), DragButton (page 104), Using Drag and Drop (page 85)

DrawButton

Description A script's DrawButton handler executes each time OneClick draws or redraws the button. OneClick redraws a button whenever any of the following occur:

- the button gets clicked
- the button, its palette, or the screen gets updated
- a script changes a button's visual properties (text, color, icon, size, and so on)
- the button becomes visible (if it was previously hidden or obscured)

Examples

```
// Play the "Quack" sound whenever OneClick redraws the button
On DrawButton
    Sound "Quack"
End DrawButton
```

See Also DrawIndicator (page 105), Button.Update (page 166), Palette.Update (page 181)

MouseDown

Description A script's MouseDown handler executes when you click the mouse on a button, but before you release the mouse. The handler executes as soon as you click the button.

A script cannot contain both MouseUp and MouseDown handlers. If a script does contain both handlers, only the MouseDown handler runs when you click the button.

Examples

```
// Play the Quack sound when you click the button.
// The sound starts playing as soon as you click.
On MouseDown
    Sound "Quack"
End MouseDown
```

See Also MouseUp (page 199), IsMouseDown (page 152)

MouseUp

Description A script's MouseUp handler executes when you click and release the mouse on a button. The script doesn't start executing until after you release the mouse button. (This lets you cancel clicking the button by moving the pointer off of the button before releasing the mouse button.)

The MouseUp handler is the default handler for a script. If the script contains PopupMenu, PopupPalette, or PopupFiles, then MouseDown is the default handler instead.

A script cannot contain both MouseUp and MouseDown handlers. If a script does contain both handlers, only the MouseDown handler runs when you click the button.

Examples

```
// Play the Quack sound when you click and release the mouse button.
On MouseUp
      Sound "Quack"
End MouseUp
```

See Also MouseDown (page 198), IsMouseDown (page 152)

Scheduled

Description A script's Scheduled handler executes each time a Scheduled event occurs.

Use the Schedule command to turn on scheduling for a script. (If you want scheduling to run all the time, put the Schedule command in the script's Startup handler.) You can specify how often the Scheduled handler should run in increments of 1/10th of a second. Scheduled scripts run only when the Macintosh is idle (waiting for keyboard or mouse input).

Normally, a Scheduled handler runs only when its palette is visible; the scheduling stops if the palette is hidden, then resumes when the palette is made visible again. To have a Scheduled handler run even if its palette is hidden, include 1 (True) in the Schedule command's optional *run-always* parameter.

Examples

```
// Play the Quack sound every five seconds from the time the application starts up.
On Startup
    Schedule 50
End Startup

On Scheduled
    Sound "Quack"
End Scheduled

// Turn on scheduling for this script when the button is clicked.
// Run even if the palette is hidden.
Schedule 100, 1

// This handler runs once every ten seconds. It checks the contents of a folder and displays
// a message box whenever new files appear in the folder. It then moves the new files
// to a different folder (leaving the original folder empty again) and opens the folder.
On Scheduled
    Variable theServerPath, myPath
    theServerPath = "Accounting Server:Orders to Process:"
    myPath = "Accounting Server:Donna's Orders:"
    // The following statements run only if there are one or more files in "Orders to Process".
    If File(theServerPath).List <> ""
        Message "There are new files in " & theServerPath
        // Move all the files in "Orders to Process" to "Donna's Orders".
        Directory = theServerPath
        FinderMove File(theServerPath).List, myPath
        // Open "Donna's Orders".
        Open myPath
    End If
End Scheduled
```

See Also Scheduling a script to run periodically (page 90), Schedule (page 114), Startup (page 200)

Startup

Description A script's Startup handler executes automatically when the application starts up. Startup handlers on global palettes execute after the computer starts up. A Startup handler runs even if the script's button or palette is hidden.

If you make changes to a script that contains a startup handler, or duplicate a button whose script contains a startup handler, the handler runs immediately after you close the OneClick Editor window.

Examples

```
// Turn on scheduling for this script.
On Startup
      // Make the Scheduled handler run every five seconds.
      Schedule 50
End Startup

// Show the palette (if it's hidden) when the application starts up.
On Startup
      Palette.Visible = 1
End Startup
```

Appendix A

EasyScript Summary

The following tables summarize all of EasyScript's built-in and external commands, functions, and system variables. For more detailed information about a specific command and how to use it, see Chapter 5, "EasyScript Reference" or see the online help available in the Script Editor.

▶ **To get help for an EasyScript keyword**

1. Open the OneClick Editor window.
2. Click the Script tab.
3. Click the  button to display the keyword list.
4. Select a keyword from the list.
5. Click the  button to display help for the selected keyword.

Note Keywords marked with an asterisk (*) are OneClick extensions (external commands, functions, or system variables). OneClick extensions are stored in the Extensions folder inside the OneClick folder (in Preferences). If the OneClick extension files are not installed, then the extra keywords they provide are not available to use in scripts.

Command	What it does
AppleScript	Indicates that the following lines are AppleScript statements.
Beep	Plays the Macintosh system beep.
Call	Calls another script as a subroutine of the current script.
Click	Simulates clicking coordinates on the screen or within a window or dialog box.
CloseWindow	Closes the active window.
ConvertClip	Forces conversion of the Clipboard contents to plain text.
Dial*	Dials a phone number through the speaker or a modem.
DragButton	Drags the specified text from a button.
DrawIndicator	Draws a progress bar or pie indicator.
Else	Indicates statements to execute if a logical expression evaluates to False.
End	Indicates the end of a block of statements.
Exit	Stops running the current script and returns to the calling script, if any.
FinderCopy	Copies files to the specified folder.
FinderMove	Moves files to the specified folder.
For	Repeats one or more statements while incrementing the specified variable.
If	Indicates statements to execute if a logical expression evaluates to True.
Message	Displays a message in a dialog box with an OK button.
Next	Forces the next iteration of a For, Repeat, or While loop.
On	Specifies the start of a handler.
Open	Opens the specified application, document, or folder.
PaletteMenu	Displays the OneClick menu as a pop-up menu.
Pause	Pauses for the specified time interval, then resumes.
PopupPalette	Displays another palette as a pop-up palette.
QuicKey*	Runs the specified QuicKey macro. Requires QuicKeys™ from CE Software.
Repeat	Repeats one or more statements the specified number of times.
Schedule	Specifies how often a script should run automatically.

Command	What it does
Scroll	Simulates clicks in the active window's scroll bars.
SelectButton	Clicks a named button or checkbox within a window or dialog box.
SelectMenu	Simulates choosing a command from a pull-down menu.
SelectPopUp	Simulates choosing a command from a pop-up menu in a window or dialog box.
Set	Assigns a value to a variable.
Sound	Plays the specified sound.
Speak*	Speaks the specified text.
Type	Types the specified text or command keys.
Variable	Declares variable names for use in a script
Wait	Waits until the specified condition evaluates to True, then resumes.
While	Repeats one or more statements while the specified condition is true.
With	Specifies the object whose properties are referenced in the following statements.
//	Indicates that the rest of the line is a comment.

Function	What it does
Absolute	Returns the absolute value of a number.
AskButton	Displays an alert box and returns a value indicating which button was used to dismiss the alert.
AskFile	Displays a directory dialog box and returns the full path of the file or folder selected.
AskList	Displays a list box and returns the selected item.
AskText	Displays a dialog box with a text field and returns the text typed in the field.
Char	Returns the character indicated by the ASCII code parameter.
Code	Returns the ASCII code of the string parameter's first character.
Date	Returns the current date in different string formats.
Find	Returns the character position of one text string within another.
FindFolder	Returns the full path of the specified folder.
Gestalt	Returns Gestalt information for the specified selector.

Function	What it does
GetDragAndDrop	Returns a list of paths of dropped files or returns the text in a dropped text clipping.
GetResources	Returns a list of all the resources of the specified type.
Length	Returns the number of characters in a text value.
ListCount	Returns the number of items in a list.
ListItems	Returns a subset of items from the specified list.
ListSort	Sorts the specified list and returns the sorted list.
ListSum	Returns the sum of all numbers in a list.
Lower	Returns a text value as all lowercase letters.
MakeNumber	Converts a text value to a number and returns the result.
MakeText	Converts a number to a text value and returns the result.
PopupFiles	Displays a hierarchical pop-up menu of files and folders and returns the path of the chosen item.
PopupFont*	Displays a pop-up menu of all the characters in a font and returns the chosen character.
PopupMenu	Displays a list as a pop-up menu and returns the chosen item.
Proper	Returns a string with the first letter of each word capitalized.
Random	Returns a random number between 1 and the specified value.
Replace	Replaces occurrences of one string within another.
Return	Returns the carriage return character. Use with Type to press the Return key from a script.
SubString	Returns a portion of a string.
Tab	Returns the tab character. Use with Type to press the Tab key from a script.
Time	Returns the current time in different text formats.
Trim	Returns text with extra spaces removed.
Upper	Returns a text value as all uppercase letters.

Variable	What it does
ASResult	Returns the result of the last AppleScript statement.
BeepLevel	Returns or sets the system beep volume level.

Variable	What it does
Clipboard	Returns the contents of the Clipboard.
CommandKey	True when the Command key is pressed, otherwise False.
ControlKey	True when the Control key is pressed, otherwise False.
Cursor	Returns the resource ID number of the current cursor.
Dialogs	Enables or disables display of dialog boxes while a script runs.
Directory	Returns or sets the path of the last folder used in the application.
Error	Returns the error code of the most recent script error.
IsKeyDown	Returns True when a key is currently pressed, otherwise False.
IsMouseDown	True when the mouse button is clicked or held down, otherwise False
ListDelimiter	Returns or sets the delimiter character to use with lists.
OptionKey	True when the Option key is pressed, otherwise False.
ShiftKey	True when the Shift key is pressed, otherwise False.
SoundLevel	Returns the current speaker volume level (0—7) or sets the volume to a new level.
SystemFolder	Returns the path to the System folder on the startup disk.
Ticks	Returns the number of ticks (1/60th of a second) since the computer was started.

Object	What it does
Button	Accesses or manipulates the properties of OneClick buttons.
DialogButton	Accesses the properties of buttons in a window or dialog box.
File	Accesses or manipulates the properties or contents of files and folders.
Menu	Accesses the properties of menus or menu items.
Palette	Accesses or manipulates the properties of OneClick palettes.
Process	Accesses or manipulates the properties of running applications.
Screen	Accesses or manipulates the properties of monitors.
Volume	Accesses or manipulates the properties of mounted volumes (disks).
Window	Accesses or manipulates the properties of windows.

Handler	What it does
DragAndDrop	Indicates statements to execute if the button was triggered by a Drag and Drop event.
DrawButton	Indicates statements to execute upon a Button Draw event.
MouseDown	Indicates statements to execute if the button was triggered by a MouseDown event.
MouseUp	Indicates statements to execute if the button was triggered by a MouseUp event.
Scheduled	Indicates statements to execute upon a Schedule event.
Startup	Indicates statements to execute when the application starts up.

Operator	What it does
AND	Performs a logical AND.
NOT	Performs a logical NOT.
OR	Performs a logical OR.
"	Encloses a literal text string.
&	Joins the text string on the left with the text string on the right.
( )	Encloses expressions to be evaluated first.
*	Multiplies the number on the left by the number on the right.
+	Adds the number on the left to the number on the right.
–	Subtracts the number on the right from the number on the left.
/	Divides the number on the left by the number on the right.
<	Evaluates to True if the expression on the right is greater than the expression on the left.
<=	Evaluates to True if the expression on the right is greater than or equal to the expression on the left.
<>	Evaluates to True if the expression on the right and the expression on the left are not equivalent.
=	Evaluates to True if the expression on the right and the expression on the left are equivalent.
>	Evaluates to True if the expression on the right is less than the expression on the left.
>=	Evaluates to True if the expression on the right is less than or equal to the expression on the left.

Appendix B

AppleScript Information

Why use AppleScript?

AppleScript is a system-level scripting language that's part of the Mac OS. AppleScript lets you control applications that are designed to support scripting (called AppleScript-aware applications). Not all applications support AppleScript, but newer versions of most major commercial applications do support it to some degree.

The primary reason to use AppleScript is the model in which it works. Whereas EasyScript lets you control an application by driving its user interface (clicking buttons and selecting menu items), AppleScript lets you control an application by using a scripting vocabulary that's built in to the application. For example, in EasyScript, you could sort a FileMaker Pro database with the following script:

```
SelectPopUp 27, 11, "List"      // Choose the List layout from the Layout pop-up menu
SelectMenu "Select", "Sort..."  // Choose the Sort command from the Select menu
SelectButton "Clear All"        // Remove all items from the Sort Order list box
Type Down Down                  // Select the second item (Last Name) in the Field list box
SelectButton "» Move »"         // Move Last Name from the Field list to the Sort Order list
SelectButton "Sort"             // Click the Sort button
```

The script performs its work entirely by clicking options, typing, and choosing menu commands. AppleScript, by comparison, gets the same result by using some scripting keywords that are built into FileMaker Pro:

```
tell application "FileMaker Pro"
    Show Layout "List" of Document "Phone Book"
    Sort Layout "List" By Field "Last Name"
end tell
```

The phrases "Show Layout" and "Sort Layout By Field" are part of FileMaker Pro's built-in AppleScript vocabulary; they aren't part of AppleScript. Each AppleScript-aware application comes with its own vocabulary that it understands. (Most AppleScript-aware applications also support a common or "core" vocabulary.)

An application's vocabulary is usually specific to the purpose of the application, for example:

- WordPerfect 3.1's vocabulary lets you manipulate words, paragraphs, pages, and text formatting in word processing documents.
- FileMaker Pro's vocabulary lets you manipulate fields, records, and layouts within database documents.
- Microsoft Excel's vocabulary lets you manipulate cells, rows, columns, tables, and charts in spreadsheet documents.

AppleScript software is included with System 7.5 or newer and System 7 Pro, and is also available as a separate product. (It is not included with OneClick.) If you don't already have AppleScript, you can purchase it from most software stores or mail-order retailers.

Note AppleScript knowledge is not required to use OneClick or write EasyScript scripts, and OneClick works just fine without AppleScript installed.

Integrating OneClick and AppleScript

By leveraging the unique features of both scripting languages, you can achieve greater scripting control over the applications you use. You can use OneClick buttons to run either EasyScript or AppleScript scripts (or a combination of the two), and you can share data between the two languages by the use of global variables.

- Use OneClick to drive the user interface of applications and to control applications that don't support AppleScript.
- Use AppleScript to manipulate information in documents whose applications are AppleScript-aware.

Launching compiled AppleScript scripts

An EasyScript script can launch an AppleScript script that is saved as either a compiled script or as an "applet" (a compiled script that's saved as a double-clickable application). Use the AppleScript command to run the script.

```
// Run the "Start File Sharing" script included with System 7.5.
AppleScript "Mac HD:AppleScript:Automated Tasks:Start File Sharing"
```

If you launch an AppleScript "droplet" (a script application that expects you to drop something on its icon), the AppleScript script will usually use the current selection as the dropped item (just as if the selection was dropped on the icon). You cannot pass information from EasyScript's GetDragAndDrop function to an AppleScript droplet.

Embedding AppleScript code in an EasyScript script

The EasyScript language lets you extend its power by including scripts written in AppleScript directly in your EasyScript scripts. This AppleScript embedding capability allows your EasyScript scripts to contain a mixture of EasyScript and AppleScript code.

To embed AppleScript code within an EasyScript script, type or paste the AppleScript statements between AppleScript and End AppleScript commands.

```
// Eject removable disks from all drives.
// This script requires the Scriptable Finder in System 7.5.
AppleScript
    tell application "Finder"
        put away (every disk whose ejectable is true)
    end tell
End AppleScript
```

When you save or check the syntax of a script that contains embedded AppleScript code, OneClick tells the AppleScript extension to compile the AppleScript portions of the script. If the AppleScript compiler needs to report an error message, such as a syntax error in an AppleScript statement, the message appears in the status line in the Script Editor (where EasyScript compiler messages normally appear).

Accessing the AppleScript result variable

After each AppleScript statement executes, the special AppleScript variable "result" gets set to the result of the statement. Use EasyScript's ASResult system variable to access the AppleScript result variable. ASResult always returns the AppleScript result variable as a string, no matter what the original data type was in AppleScript.

Following is a sample script that performs a calculation and returns the floating-point (decimal) result as a string. OneClick displays the result in a message box.

```
// Displays a OneClick message box containing the string "1.4"
AppleScript
      get (3 + 4) / 5
End AppleScript
Message ASResult
```

Accessing OneClick variables from an AppleScript script

The OneClick Scripting Addition file (included with OneClick) lets you get and set the values of EasyScript global variables from within an AppleScript script. You can access only global variables (not local or static variables) from within AppleScript. Use the following syntax in your AppleScript scripts:

```
get OneClick variable "global-variable-name"
set OneClick variable "global-variable-name" to value
```

Note that the OneClick variable name is a string enclosed in quotes. To access an EasyScript global variable and use it in AppleScript, the easiest way to do so is to first get its value, then copy the value to an AppleScript variable of the same name.

The following is an example that passes the pathname of a dropped Finder icon to AppleScript. The AppleScript script retrieves the pathname from the global variable theFileToSend; the script then tells Anarchie (an Internet FTP client program) to send the specified file to an FTP (File Transfer Protocol) server on the Internet. The end result is that the file dropped on the OneClick button is sent to the FTP server.

```
On DragAndDrop
    Variable Global theFileToSend
    // Get the pathname of the first icon dropped on the button.
    theFileToSend = GetDragAndDrop 1
    // Tell Anarchie to upload the specified file to the FTP server.
    AppleScript
        tell application "Anarchie"
            activate
            – Get the value of theFileToSend and store it in an
            – AppleScript variable of the same name.
            get OneClick variable "theFileToSend"
            copy result to theFileToSend
            – Transfer the file to the "incoming" directory on the FTP server.
            store file theFileToSend host "crash.cts.com" path "incoming" ¬
                user "jeffmj" password "myPass" with binary
        end tell
    End AppleScript
End DragAndDrop
```

Calling a OneClick script from an AppleScript script

The OneClick Scripting Addition file lets you call the script of a OneClick button as a subroutine in AppleScript. The technique is similar to the way you can use the Call command in an EasyScript script. Use the following syntax to call a script.

```
call OneClick button "button-name" on palette "palette-name"
```

Button-name and *palette-name* are strings enclosed in quotes. Note that unlike EasyScript's Call command, a palette name is required when you call a OneClick button from within AppleScript.

Following is a sample script that calls a OneClick button. The OneClick button opens the folder named Utilities on the startup disk; the AppleScript statements then zoom and position the folder's open window.

```
tell application "Finder"
    activate
    call OneClick button "OpenUtilitiesFolder" on palette "Launcher"
    set zoomed of window of folder "Utilities" of startup disk to true
    set position of window of folder "Utilities" of startup disk to {100, 85}
end tell
```

Determining if AppleScript is installed

If you write scripts for people to use on other Macs, it's a good idea to include code in your scripts that lets you determine if AppleScript is installed or not. By doing so, you can alert the user that AppleScript needs to be present for the script to run correctly. Use the Gestalt function with the "ascr" selector to find out if AppleScript is available.

```
If NOT Gestalt "ascr", 0
    Message "This button requires AppleScript, but AppleScript is not installed."
    Exit
End If
AppleScript SystemFolder & "Scripts:Universal Scripts:Start File Sharing"
```

Gestalt "ascr", 0 returns 1 (True) if AppleScript is available, otherwise 0 (False).

AppleScript resources

Because AppleScript is a very different language, describing its syntax and use is beyond the scope of this manual. There are a number of good books on AppleScript, including the following:

- **Danny Goodman's AppleScript Handbook**, Second Edition, published by Random House. This book covers the basics of writing AppleScript scripts, and also includes intermediate and advanced topics. Several sections cover how to script many popular business applications. The book comes with a CD-ROM containing all kinds of goodies: AppleScript and sample scripts, scripting additions, some scriptable applications, documentation, and more.
- **The Tao of AppleScript**, Second Edition, published by Hayden Books. This book is better suited for scripting beginners than the previous book. It's best for people who have no programming knowledge or experience. The book comes with two disks containing AppleScript, sample scripts, scripting additions, and other files.
- **Getting Started with AppleScript, AppleScript Language Guide,** and **AppleScript Scripting Additions Guide**, all published by Addison-Wesley. These are Apple's official manuals for the AppleScript language.
- **AppleScript Finder Guide**, published by Addison-Wesley. This is Apple's reference manual for the Scriptable Finder included in System 7.5. The manual assumes you already know AppleScript. It's essential for people who want to write advanced scripts that control the Finder and manipulate Finder objects.

Index

A

B

D

E

F

G

H

I

K

N

O

P

Q

R

S

T

U

V

W

Z