# OneClick Authoring Guide

WestCode Software, Inc. • http://www.westcodesoft.com
619–487–9200 • fax 619–487–9255

## The OneClick Product Team

Alan Bird
Leonard Rosenthol
Rob Renstrom
John Oberrick
Jeff Jungblut
Mark Brooks

## Manual and Layout

Jeff Jungblut

## Cover and Package Design

Steve Sharp, Sharp Advertising & Design

## Copyright

Version 2.0 by Leonard Rosenthol

## Trademarks

Third Printing, November 1999
Printed in the United States of America.

**Disclaimer of Warranty on Software.** You expressly acknowledge and agree that use of the software is at your sole risk. The Software and related documentation are provided "AS IS" and without warranty of any kind and WestCode and WestCode's Licensor(s) expressly disclaim all warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. WestCode does not warrant that the functions contained in the Software will meet your requirements, or that the operation of the Software will be uninterrupted or error-free, or that defects in the Software will be corrected. Furthermore, WestCode does not warrant or make any representations regarding the use or the results of the use of the Software or related documentation in terms of their correctness, accuracy, reliability, or otherwise. No oral or written information or advice given by WestCode or a WestCode authorized representative shall create a warranty or in any way increase the scope of this warranty. Should the Software prove defective, you (and not WestCode or a WestCode authorized representative) assume the entire cost of all necessary servicing, repair or correction. Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

**Limitation of Liability.** Under no circumstances including negligence shall WestCode be liable for any incidental, special or consequential damages that result from the use or inability to use the Software or related documentation, even if WestCode or a WestCode authorized representative has been advised of the possibility of such damages. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

In no event shall WestCode's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Software.

# Contents

# Chapter 1
# Introduction

This manual shows you how to create custom palettes using the OneClick Editor and 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 the standard palettes listed in the OneClick menu. If you haven't already done so, read Chapter 3, "Getting Started with OneClick" in the *OneClick User's Guide*.

## About this manual

This manual is divided into two parts. Chapters 2 through 6 cover how to create palettes and buttons using the OneClick Editor. The remaining chapters cover EasyScript, OneClick's scripting language.

- Chapter 2, "Using the Button Library," shows you how to add pre-designed buttons to a palette by selecting from a button library.
- Chapter 3, "Using the Palette Editor," shows you how to create, modify, import, and export palettes.
- Chapter 4, "Using the Button Editor," shows you how to create and modify buttons.
- Chapter 5, "Using the Script Editor," describes all the options available in the Script Editor, including the script recorder, editor, compiler, and online help.
- Chapter 6, "Using the Icon Editor and Icon Search," shows you how to edit button icons and grab icons from files.
- Chapter 7, "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 8, “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, “Integration with AppleScript,” shows how you can integrate AppleScript scripts with OneClick scripts and provides pointers to other sources of information for AppleScript users.

# Why script?

It’s not necessary to learn how to write scripts to make use of OneClick. The Make a Shortcut commands and the standard palettes 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 creating your own palettes and scripts can extend OneClick’s capabilities for virtually any need.

## Technical information for Macintosh developers

The OneClick manuals do not include technical information regarding the development of OneClick extensions (external script keywords and 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, visit WestCode Software’s web site at http://www.westcodesoft.com.

# Opening the OneClick Editor

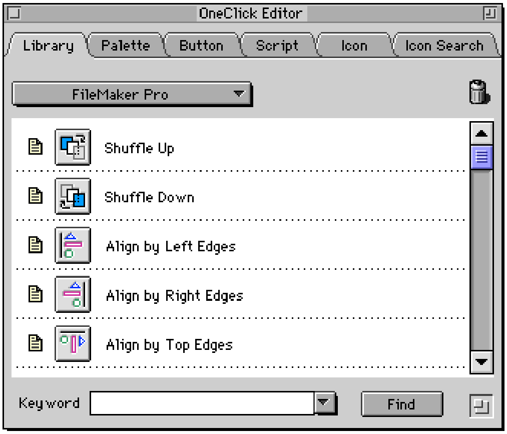
All the palette customizing features are available in the OneClick Editor window. The OneClick Editor is always available; you don’t need to run an application to access it. To open the window, choose OneClick Editor from the OneClick menu.

**Shortcut** Press Command-Option-` to open or close the OneClick Editor.

# A quick tour of the OneClick Editor

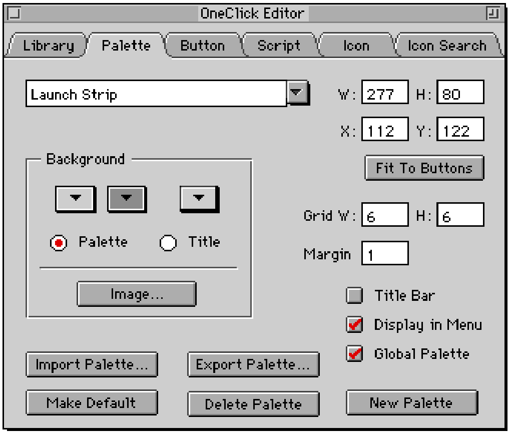
Copyright © 1995-1999 Alan L. Bird and WestCode Software, Inc.

**The OneClick Editor window** contains six tabs across the top. To choose a category of options, click the appropriate tab.



FileMaker Pro

- Shuffle Up
- Shuffle Down
- Align by Left Edges
- Align by Right Edges
- Align by Top Edges



Launch Strip

W: 277 H: 80
X: 112 Y: 122
Grid W: 6 H: 6
Margin 1

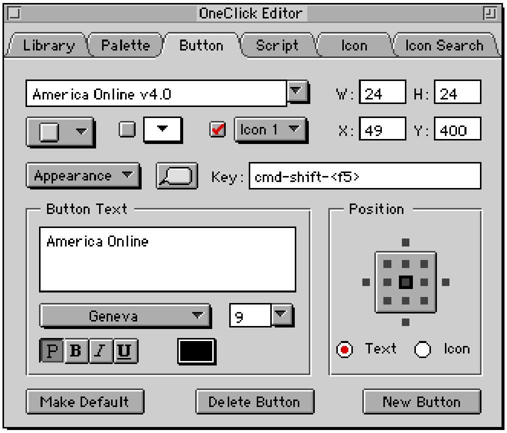
● Palette ○ Title

□ Title Bar
☑ Display in Menu
☑ Global Palette

**The Button Library** holds collections of pre-designed buttons, organized by application. You can add new buttons to a palette by dragging them from the Button Library to the palette.

**The Palette Editor** lets you create a palette and change the palette's appearance. You can create palettes that appear system-wide (global palettes) and palettes that appear only when a specific application is active.

## *Opening the OneClick Editor*



America Online v4.0 W: 24 H: 24

☑ Icon 1 X: 49 Y: 400

Key: cmd-shift-<f5>

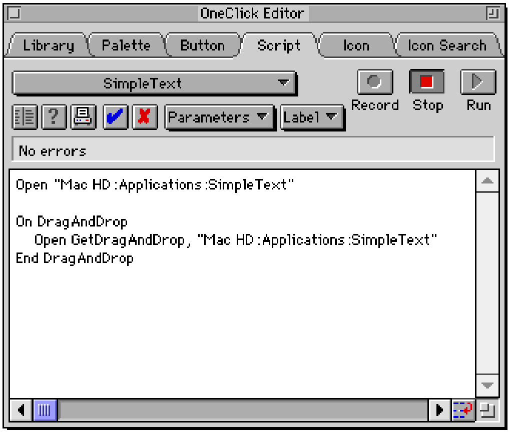
America Online

Geneva 9

P B I U

● Text ○ Icon

**The Button Editor** lets you create new buttons on a palette. You can change a button's icon, size, color, text label, border style, keyboard equivalent, and other options.

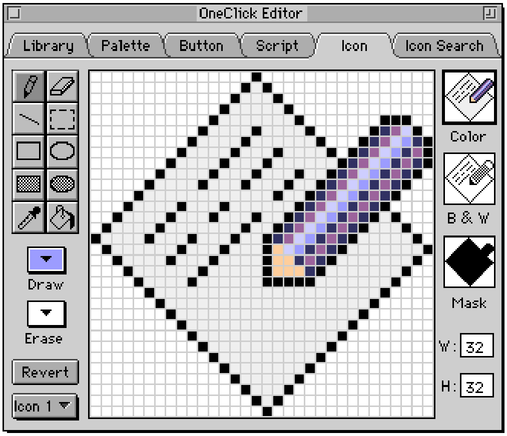


SimpleText

No errors

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

On DragAndDrop
  Open GetDragAndDrop, "Mac HD:Applications:SimpleText"
End DragAndDrop
```

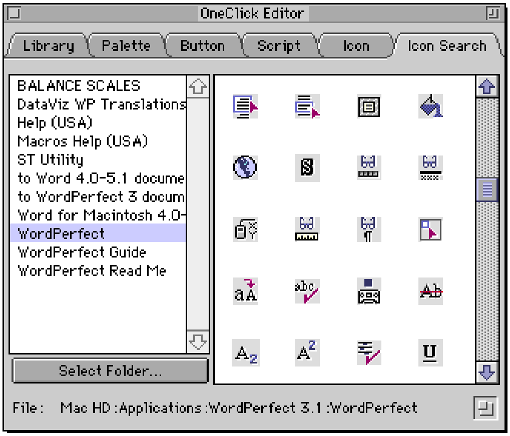
**The Script Editor** lets you record and edit button scripts. Using OneClick's scripting language, EasyScript, you can add advanced functionality to your recorded scripts or write new scripts from scratch.



W: 32

H: 32

**The Icon Editor** lets you change the appearance of a button's icon. You can use the tools in the Icon Editor to change the icons of existing buttons and to make new icons.



BALANCE SCALES
DataViz WP Translations
Help (USA)
Macros Help (USA)
ST Utility
to Word 4.0-5.1 docume
to WordPerfect 3 docum
Word for Macintosh 4.0-
WordPerfect
WordPerfect Guide
WordPerfect Read Me

File: Mac HD:Applications:WordPerfect 3.1:WordPerfect

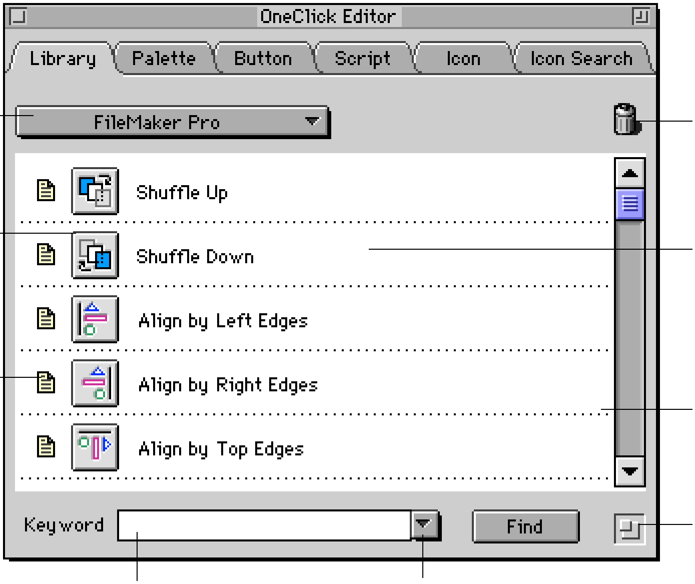
**The Icon Search** lets you "raid" icons from applications or other files containing icons. If you don't want to create icons from scratch, this is an easier method—just drag icons found in other files to your palettes.

## Chapter 2

# Using the Button Library

The Button Library is a storage area for buttons. Buttons are organized into various button library files. You can easily copy buttons from a library to any palette, and you can create your own libraries to store buttons.

Many of the buttons available on WestCode's web site are stored in library files. If you download a library file, you can open the file in the Button Library and then drag the buttons from the library to one of your palettes.



Choose a library from the pop-up menu

Drag a button to a palette to add it to the palette

Drag a script icon to copy just the button's script to another button

Drag a button to the trash to delet it from the libra

Click and type to edit a button's Balloon Help message

Drag a divider line to change the height of list items

Drag to resize th library window

Type a word to search for, then click Find to show all the buttons containing the word

Choose a search word or add a word to the pop-up menu

## Choosing a library of buttons

▶ **To open the Button Library**

1. Open the OneClick Editor.
2. Click the **Library** tab.

   If a library for the active application exists, that library appears in the list box. The list shows all the buttons in the library with a brief description of each button. The active library's name appears in the pop-up menu above the list.

3. To choose a different library, choose a library name from the pop-up menu.

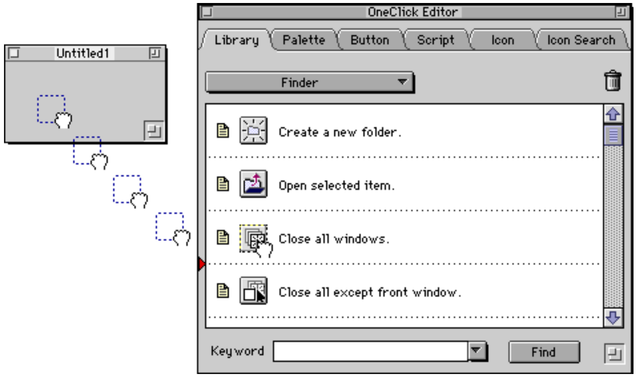
### The Universal library

In addition to several application-specific libraries, OneClick includes a Universal library with buttons you can add to any palette. The Universal library contains useful generic controls that you can use in any application.

## Copying buttons and scripts from the library to a palette

▶ **To copy a button from the library to a palette**

■ Drag the button's icon to an empty space on the palette.



Create a new folder.

Open selected item.

Close all windows.

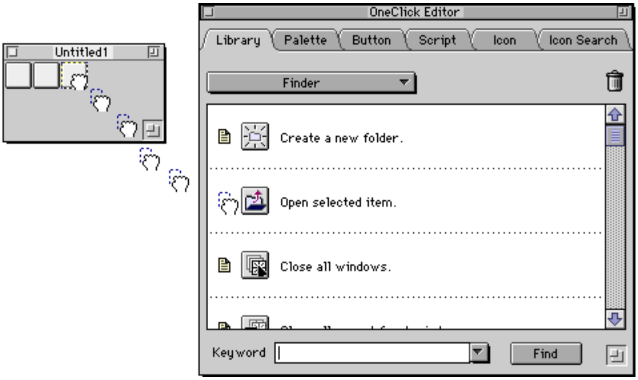
Close all except front window.

All the button's attributes are copied, including the icon, script, Balloon Help message, and other settings.

If you want, you can copy only the button's script from the library to another button on a palette. Copying a script to another button replaces the button's previous script, if any.

▶ **To copy a script from the library to a button on a palette**

■ Drag a button's script icon (🗎) from the library to a button on a palette.



Create a new folder.

Open selected item.

Close all windows.

When you drag a script icon over a button, the button highlights to show which button will receive the new script.

## Searching for specific buttons

In a library with a lot of buttons, you can use the Find feature to quickly search for the buttons you want. OneClick displays only the buttons whose descriptions contain the keyword you specify.

▶ **To search for buttons**

1 Choose the library you want to search from the Library pop-up menu.

2 Type the word or phrase you want to search for in the Keyword box.

3 Click **Find**.

Only the buttons containing the search keyword appear in the library list.

4 To redisplay all the buttons in the library, click **Show All**.

You can add frequently used keywords to the Keyword pop-up menu.

- To add a keyword to the pop-up menu, type a word or phrase in the Keyword box, then choose **Add**.
- To remove a keyword from the pop-up menu, choose the keyword, then choose **Remove**.

When you choose a keyword from the menu, OneClick automatically finds and displays the buttons containing the keyword (you don't need to click **Find**).

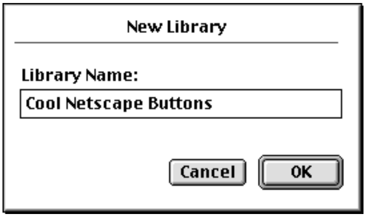
## Creating a library and adding buttons

You can create new libraries to store buttons you create and for applications that don't already have their own library. Putting buttons in a library lets you share your buttons with other users; they can open your library and drag its buttons to their palettes.

▶ **To create a new library**

1 Choose **New Library** from the Library pop-up menu.

2 Type a name in the Name box, then click **OK**.



New Library

Library Name:

Cool Netscape Buttons

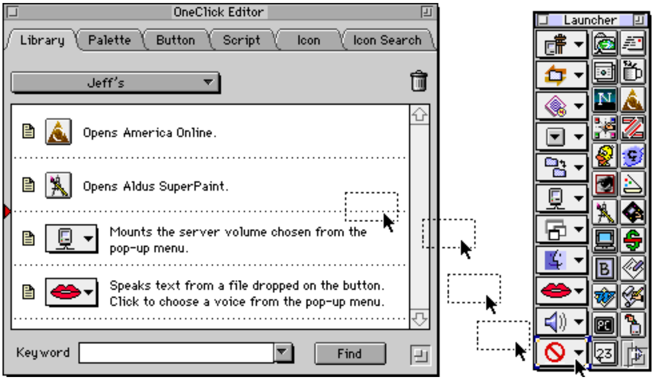
You can type up to 31 characters for the name.

An empty library appears in the list box, and you can now add buttons to the library. The library file is stored in the Libraries folder within the OneClick Folder (in Preferences).

You can add buttons to your own libraries or any of the pre-designed libraries that come with OneClick.

### ▶ To add buttons to the library

- ■ Select and drag one or more buttons from a palette to the library list.



Opens America Online.

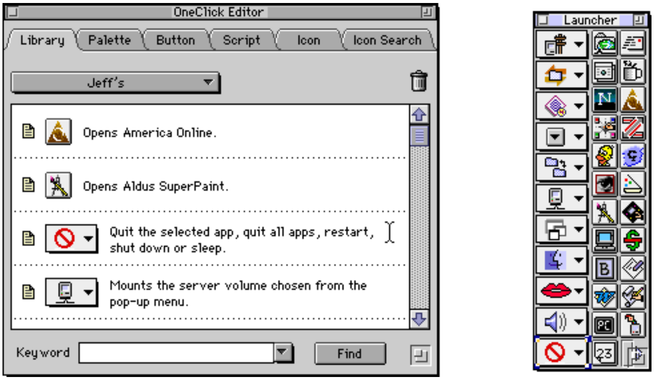
Opens Aldus SuperPaint.

Mounts the server volume chosen from the pop-up menu.

Speaks text from a file dropped on the button. Click to choose a voice from the pop-up menu.

Dragging the ⊘ button from a palette to the library

A red triangle on the left edge of the list shows where the button will be inserted when you release the mouse button. Dragging a button to the library doesn't remove it from the palette; it makes a new copy of the button in the library.



Opens America Online.

Opens Aldus SuperPaint.

Quit the selected app, quit all apps, restart, shut down or sleep.

Mounts the server volume chosen from the pop-up menu.

### ▶ To rearrange the order of buttons

- Drag buttons up or down in the list.
- To move a button to a position that isn’t in view, drag the button above or below the list until the list starts scrolling. When the desired position appears, drag the button back into the list and drop it in the list.

### ▶ To change the height of list items

- Move the pointer over one of the dotted divider lines between list items.
- When the cursor changes to the resize cursor (‡), drag the line up or down.

  All list items change to the new height.

The descriptions next to each button are the buttons’ Balloon Help messages. You can edit the Balloon Help text directly in the list; the help text is the same text that appears in the Balloon Help dialog box in the Button Editor.

### ▶ To edit a button’s Balloon Help message

- Click the button’s help text (to the right of the button) and type. You can use Command-X, Command-C, and Command-V to cut, copy, and paste text in the Balloon Help messages.

Changing the help text in the library does *not* change the help text for any copies of the buttons already placed on palettes.

## Managing library files

The Library pop-up menu lists all the library files found in the Libraries folder. You can use the Open Library command in the pop-up menu to open a library file from another disk or folder, which lets you use a library created by other OneClick users or supplemental libraries provided by WestCode.

### ▶ To open a library file

1. Choose **Open Library** from the Library pop-up menu.
2. Use the directory dialog box to find and open the library file.

**Note** You can make the library available at all times by dropping the library file in the Libraries folder within the OneClick Folder.

### ▶ To delete a button from the active library

- Drag the button to the trash can icon in the upper-right corner of the library window.

### ▶ To rename a library

1. Choose the library you want to rename from the Library pop-up menu.
2. Choose **Rename Library** from the pop-up menu.
3. Type a new name in the Library Name box, then click **OK**.

### ▶ To delete a library

1. Choose the library you want to delete from the Library pop-up menu.
2. Choose **Delete Library** from the pop-up menu.

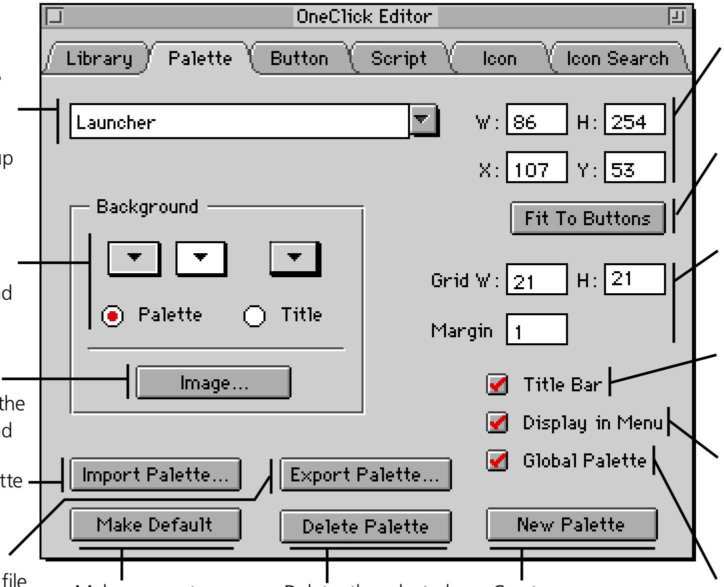
Click **Delete** when the confirmation message appears.

## Chapter 3

# Using the Palette Editor

The Palette Editor lets you create new palettes and change palette characteristics. The features of a palette that you can change include the window title, size, location, color, pattern, background picture, and button spacing.

**Shortcut** To quickly open the Palette Editor, Control-click the background of the palette you want to edit, then choose **Palette Editor** from the contextual menu.



Launcher

W: 86 H: 254

X: 107 Y: 53

● Palette ○ Title

Grid W: 21 H: 21

Margin 1

☑ Title Bar

☑ Display in Menu

☑ Global Palette

Type to rename the selected palette; choose a palette to edit from the pop-up menu

Choose colors and patterns for the palette's background and title bar

Place a picture or desktop pattern in the palette's background

Imports a new palette from a palette file

Saves the selected palette in a palette file

Makes current settings the default for new palettes

Deletes the selected palette

Creates a new, untitled global palette

Type numbers to change palette size and location

Click to auto-fit the palette around its buttons

Type numbers to adjust spacing between buttons

Click to turn the palette's title bar on or off

Uncheck to keep the palette from appearing in the OneClick menu

Uncheck to make the palette app-specific instead of system-wide

The following sections show how to create palettes and change their characteristics. For information on changing a palette's buttons or adding new buttons, see Chapter 4, "Using the Button Editor."

## Creating a new palette

You can create new palettes and add buttons to them at any time. If you create a lot of buttons for an application, you may find it useful to group the buttons on different palettes based on their function. For example, in a graphics program you could have one palette with buttons you use for file operations, and another palette with buttons you use for editing scanned images. In a database program, you could create a different palette for each database file you use.

### Global and application-specific palettes

You can create two kinds of OneClick palettes: **global** palettes which appear in all applications, and **application-specific** palettes that are available only in the application they were created for. When you quit an application, its application-specific palettes close automatically, while global palettes are always available.

The most common use you'll have for a global palette is for opening applications or documents and changing system-wide settings. You can put buttons on a global palette that open your most frequently used applications and documents so that you don't have to switch to the Finder to open them.

Another use for a global palette is for buttons that work within different applications. One convenient use would be to have a button that does the following:

- Copy selected text (such as a mailing address) to the clipboard. (Note that it doesn't matter what application the text is copied from.)
- Open an envelope-printing utility, such as Easy Envelopes+.
- Paste the address text from the clipboard.
- Print an envelope.

If you want to create buttons that work with more than one application (such as buttons that copy information between different applications), create a global palette to contain the buttons. For buttons designed to work only within a certain application, create an application-specific palette.

**Note** Adding a button to a global palette does not ensure that the button's script will work in all applications. Because most button scripts are tailored to specific applications, using a script in an application it wasn't written for could cause unpredictable results. Usually, generic buttons (such as New, Open, Print, and so on) will work in any application.

▶ **To create a new, empty palette**

1. If you want to create a palette for a specific application, switch to that application.
2. Open the OneClick Editor.
3. Click the **Palette** tab.
4. Click **New Palette**.

   A new, untitled global palette appears. You can now change the look of the palette and add buttons.
5. If you want the palette to appear only in one application, switch to that application, then uncheck the **Global** checkbox.

## Selecting a palette to edit

Before changing a palette's characteristics, you need to select it so OneClick knows which palette you want to edit.

▶ **To select a palette to edit**

1. Open the OneClick Editor.
2. Click the **Palette** tab.

3 Click the palette you want to edit, or if the palette is hidden, choose its name from the pop-up menu next to the Palette Name box in the Palette Editor.

–Or–

■ Control-click the background of the palette you want to edit, then choose **Edit Palette** from the contextual menu. (This shortcut isn't available for palettes that override OneClick's contextual menu with a custom contextual menu.)

The name of the selected palette appears in the Palette Name box. A resize box appears in the lower-right corner of the palette to show it's selected.

**Shortcut** If a palette is open, but it's hidden behind the editor window or another palette, you can bring it to the front by holding down the Option key and choosing the palette's name from the OneClick menu.

## Changing a palette's name

The default palette name is Untitled and a number, such as Untitled1.

▶ **To change the palette's name**

■ Type a new name in the Palette Name box.

The name appears in the palette's title bar. You can type up to 31 characters for the name.

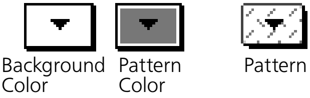
Consider naming your palettes based on the category of buttons you've placed on the palette. For example, in Adobe Photoshop, you could have three palettes named Scanning Tools, Color-Correction Tools, and Image Filters. In applications where you use only one palette, you could just name the palette after the application, such as ImageStyler Tools.

## Changing a palette's background

The palette background appears behind any buttons you place on the palette. You can choose between a solid color background (the default is light gray), two kinds of pattern backgrounds, or a picture background.

▶ **To change a palette's background color or pattern**

1 Choose a color from the Background Color pop-up menu.



The palette's background changes to the color you picked.

2 If you want the palette to have a patterned background, choose a pattern from the Pattern pop-up menu.

3 If you want a color pattern, choose a color from the **Pattern Color** pop-up menu.

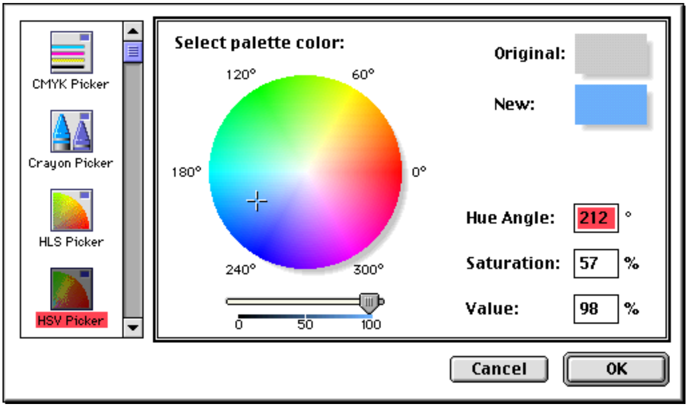
The **Palette** and **Title** buttons determine which part of the palette is changed when you choose options from the color or pattern menus. To change the color or pattern of the palette's title bar, click **Title**, then repeat the previous steps.

## Choosing a custom color

If you want to use a different color that's not included in the 256 colors available in the color menu, you can use the Color Picker dialog box to choose a custom color. Using a custom color works best if your monitor supports more than 256 colors.

▶ **To choose a custom color**

1 Select **Choose Color** from the bottom of the Color pop-up menu.



Hue Angle: 212 °

Saturation: 57 %

Value: 98 %

**2** Do any of the following:

- Drag over the color wheel to choose a hue (color) and saturation (intensity).
- Use the scroll bar to choose a brightness.
- Type color values in the number boxes.

**3** Click **OK** when you're done.

**Note** The Color Picker may look different depending on the system software you are using. Refer to the manual for your system software if you need help.

## Placing a picture or a large color pattern in the background

You can import a picture or a large color pattern to use for the palette's background. Buttons on the palette then appear to sit on top of the picture or pattern.

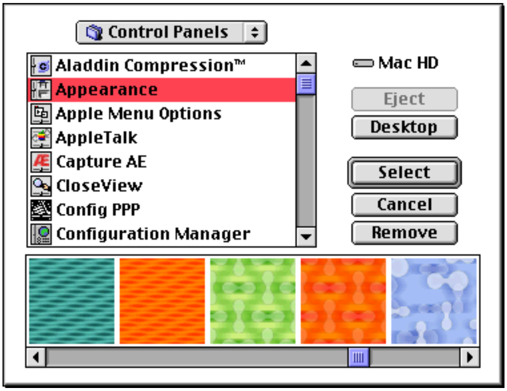
Large color patterns (sometimes called PPATs or desktop patterns) are usually 32 pixels square or larger in size and can contain more than two colors. A good source of large color patterns is the Desktop Patterns control panel included in Mac OS 7.5 and the Appearance control panel included in Mac OS 8 and later. Online services such as America Online are also a good source of pattern files.

You can import a picture from any file containing a PICT format graphic.

### ▶ To import a picture or large color pattern

1 Click **Image**.

2 Use the directory dialog box to select a file containing either color patterns or a PICT graphic. (If you're running Mac OS 8 or later, look in the Control Panels folder for the Appearance control panel.)



The selected file's images appear in the scroll box below the file list. (For PICT files, only the upper-left corner of the picture appears in the list.) Use the scroll bar to see more images.

3 To select a picture or pattern, click the image and click **Open**, or double-click the image.

The image you chose appears in the palette's background.

### ▶ To remove a picture or color pattern

1 Click **Image**.

2 Click **Remove** in the dialog box.

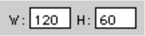
## Changing a palette's size

You can change the size of a palette three different ways.

### ▶ To resize a palette

- Drag the palette’s resize box

  –Or–



W: 120 H: 60

- Type numbers in the Height and Width boxes in the Palette Editor

  –Or–

- Click **Fit To Buttons** in the Palette Editor

Clicking **Fit To Buttons** changes the palette’s height and with so that all its buttons are visible without any empty space (except the margin) along the bottom and right edges of the palette. **Fit To Buttons** changes the palette size only; it doesn’t move any buttons.

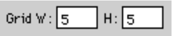
If you resize a palette smaller so that not all of its buttons are visible, the hidden buttons are not removed or disabled. Hidden buttons can still be called from scripts in other buttons; this provides an easy way to create hidden “subroutine” buttons that are used only by scripts in other buttons.

## Changing a palette’s grid settings and button spacing

A palette’s grid lets you easily align and position buttons on a palette. Buttons snap to points on an invisible grid as you drag them on the palette. When resizing a palette, the resize box also snaps to the grid points.

To determine proper settings for the grid, consider the size of the buttons on the palette and how much empty space you want between each button. For example, if your buttons are 42 pixels wide by 20 pixels tall and you want two pixels of space between each button, set the horizontal grid to 44 and the vertical grid to 22 (the button’s dimensions plus 2 pixels of empty space). When you drag buttons on the palette, they line up to the grid points, leaving two blank pixels between each button.

### ▶ To change the size of the grid spacing



Grid W: 5 H: 5

1. Type new sizes in the Grid Width and Height boxes.
2. Drag buttons on the palette to make them snap to the new grid points.

**Shortcut** To override the snap grid when dragging buttons or resizing a palette, hold down any modifier key (Command, Option, Shift, or Control) while dragging.

A palette's margin setting determines the amount of space between buttons and the palette's edges. A value of 2 means two pixels of empty space between a button and the edge of a palette. Changing the margin setting affects the upper-left corner where the grid starts.

▶ **To change the size of the palette margin**

1. Type a new size in the Margin box.
2. Click **Fit To Buttons** to resize the palette with the new margin setting.

## Turning a palette's title bar on or off

Some of the palettes that come with OneClick, such as the System Bar and Finder palettes, don't display standard title bars. You can turn a palette's title bar on or off with the **Title Bar** checkbox.

▶ **To turn a palette's title bar on or off**

- Check or uncheck the **Title Bar** checkbox.

▶ **To move a palette that doesn't have a title bar**

- If the OneClick Editor is open, hold down the Option key and drag the palette's background.
- If the OneClick Editor is closed, hold down Option, Command, or Shift and drag the palette's background.

## Keeping a palette from appearing in the OneClick menu

Some "secondary" palettes that you may not use regularly are not listed in the OneClick menu. For example, you can display the System Folders palette only by clicking the System Folders button on the System Bar—the palette doesn't appear in the OneClick menu. You can use the **Display in Menu** checkbox to keep other palettes from appearing in the OneClick menu.

▶ **To hide or show a palette's name in the OneClick menu**

- Check or uncheck the **Display in Menu** checkbox.

**Note** All available palettes appear in the OneClick menu whenever the OneClick Editor is open, allowing you to show or select any palette that doesn't normally appear in the menu.

## Changing the default settings for new palettes

You can change the default colors, size, grid, margin, and other settings for any new palettes you create. When you change the default settings, new palettes you create contain the current settings in the Palette Editor, except for the global setting and the palette's name which remains Untitled.

▶ **To change the default settings for new palettes**

1. Change the settings in the Palette Editor to the settings you want new palettes to have (colors, size, and so on).
2. Click **Make Default**.

Using the Make Default feature is a quick way to copy the characteristics of one palette to a new palette. Just select a palette, click Make Default, then click New Palette.

## Deleting a palette

Deleting a palette permanently removes the palette and all the buttons it contains. Before deleting a palette, make sure you've copied to another palette (or library) any buttons you want to keep.

▶ **To delete a palette**

1. Select the palette you want to delete.

**2** Click **Delete Palette**.

If the palette contains any buttons, OneClick displays a message asking if it's OK to delete the palette.

**Note** Make sure you really want to delete the palette and all its buttons. This step cannot be undone.

**3** Click **Delete**.

The palette is permanently removed.

## Managing palette files

Palettes are stored in palette files in the Button Palettes folder (inside the OneClick Folder in Preferences). Each palette file can contain one or more palettes. All global palettes are stored in a single file named Global Palettes, while application-specific palettes are stored in files named after their associated application, followed by the word Palette, such as "SimpleText Palette" or "Adobe Illustrator® 8.0 Palette."

At system startup, OneClick opens the Global Palettes file and loads the palettes in that file. When you open an application that has an associated palette file in the Button Palettes folder, OneClick loads the palettes from the palette file and adds them to the OneClick menu. Any changes you make to active palettes are automatically saved to the corresponding palette file in the Button Palettes folder.

### Loading palette files by an application's creator code

OneClick associates an application with its palette file by name, so that if you open an application named "SimpleText 1.2," OneClick looks for a palette file named "SimpleText 1.2 Palette" in the Button Palettes folder. If you open another version of the application named "SimpleText 1.3.1," however, OneClick won't load the palettes in the file named "SimpleText 1.2 Palette" because the names don't match.

You can have OneClick locate a palette file using an application's creator code instead of its file name by including the creator code in square brackets in the palette file's name. When a palette file name includes a creator code, OneClick goes by the creator code and ignores the application name when locating the palette file to open. For example, the palettes in file “SimpleText [ttxt] Palette” open when you launch SimpleText, regardless of the actual file name of the SimpleText application. The creator code in square brackets can appear anywhere in the palette file name (if the creator code is present, the file name is ignored).

## Using palette file backups

OneClick automatically backs up palette files to help recover from accidental file corruption. Each time the system or an application starts up, OneClick opens the associated palette file, then stores a copy of the palette file in the Backup folder (inside the Button Palettes folder). If for any reason OneClick cannot open the palette file (because it has become corrupted due to a system crash, for example), OneClick displays an alert message and adds the word “(damaged)” to the end of the palette file’s name. When OneClick encounters a damaged palette file, it automatically restores and opens the previous working copy from the Backup folder. When starting up the computer or when opening an application, if you see a message telling you that a palette file has been damaged, you should open the Button Palettes folder and drag the damaged file to the Trash, then empty the Trash.

## Exporting a palette to a file

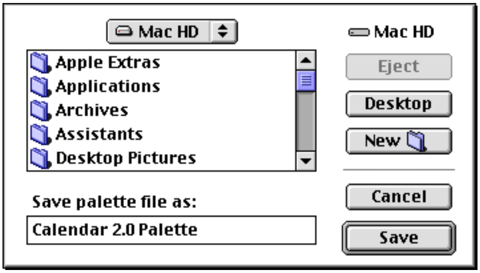
If you want to share with other people any palettes that you’ve created, you can export the palettes to files on a disk. You can also use the export feature to make backup copies of palettes.

▶ **To export a palette to a file**

1. Select the palette you want to export (so that its name appears in the Palette Editor).

   You can export only one palette at a time.

2. Click **Export Palette**.



Save palette file as: Calendar 2.0 Palette

A directory dialog box appears with a default name for the palette file.

3 Choose a location and type a new name (if desired), then click **Save**.

## Importing a palette from a file

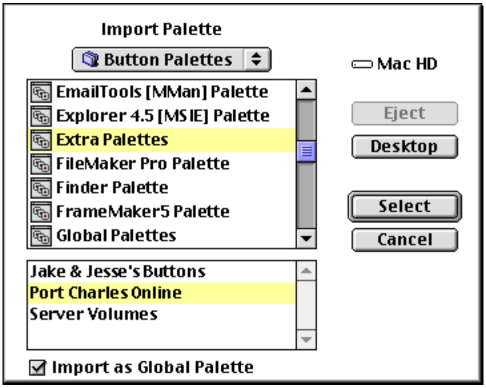
To import a palette from a file, use the Import Palette button. Importing a palette copies the palette(s) you choose into the active application's palette file or into the Global Palettes file. This lets you use palettes given to you by other people or downloaded from online services.

▶ **To import a palette**

1 If you want the palette you're importing to be application-specific, switch to the desired application first.

2 Click **Import Palette**.

3 Select the palette file containing the palette you want to import.



All the palettes contained in the palette file appear in the bottom list box.

4 Select the palette you want to import. Command-click palette names to select multiple palettes.

The **Import as Global Palette** checkbox appears checked if the palette was a global palette when it was exported.

5 If you want the new palette to be global, check the **Import as Global Palette** checkbox. Uncheck the box if you want the imported palette to work only in the active application.

6 Click **Select**.

The imported palette appears on the screen. Close the OneClick Editor to use it.

## Duplicating an active palette

Importing a palette from one of your active palette files is the easiest way to make working copies of any palette.

### ▶ To duplicate a palette

1 If you want the palette you're duplicating to be application-specific, switch to the desired application first.

2 Click **Import Palette**.

3 Go to the **Button Palettes** folder (System Folder:Preferences:OneClick Folder:Button Palettes).

The Button Palettes folder contains all of your active palette files. The Global Palettes file contains all the global palettes; other files named after applications contain application-specific palettes.

4 To duplicate a global palette, select the **Global Palettes** file. To duplicate an application-specific palette, select the application's palette file.

All the palettes contained in the palette file appear in the bottom list box.

5 Select the palette you want to duplicate. Command-click palette names to select multiple palettes.

6 If you want the new palette to be global, check the **Import as Global Palette** checkbox. Uncheck the box if you want the imported palette to work only in the active application.

7 Click **Select**.

The new palette appears in the same place on the screen as the original palette (if the original palette is open). Drag the new palette out of the way to see the original palette. Be sure to give the duplicate palette a new name so you don't get it confused with the original palette you copied.

**Note** Button scripts that refer to a palette by name may not work correctly if two palettes have the same name. Make sure all your palettes have unique names.

Chapter 4

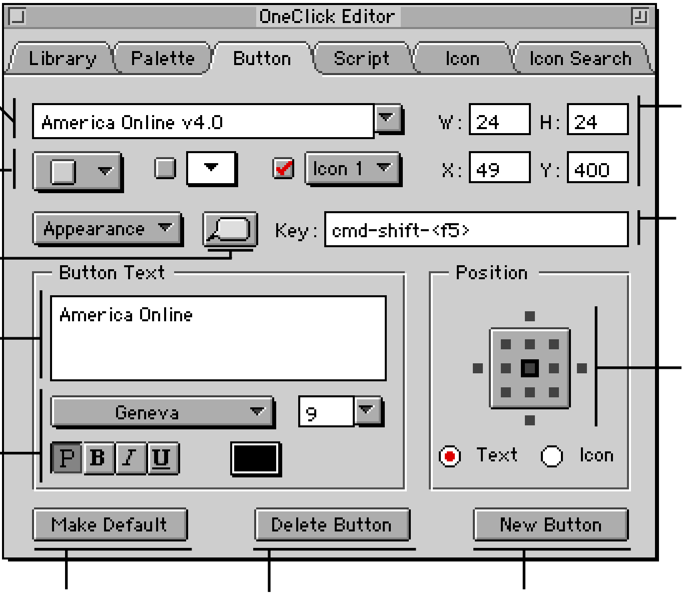
# Using the Button Editor

The Button Editor lets you change a variety of attributes for buttons, such as the button's name, visual appearance, help message, and keyboard shortcut. You can also create, duplicate, and delete buttons.

▶ **To open the Button Editor**

1 Open the OneClick Editor.

2 Click the **Button** tab.



America Online v4.0

W: 24 H: 24

X: 49 Y: 400

Key: cmd-shift-<f5>

Button Text: America Online

Geneva 9

● Text ○ Icon

Type to rename the selected button; choose a button to edit from the pop-up menu

Use pop-up menus to choose a border style, color, and icon

Click to add a Balloon Help message

Text label (appears on button's face)

Choose the font, size, style, and color of the button's text label

Type numbers to change the button's size and position

Click in the box and press a key to assign a keyboard shortcut

Click to position the button's text or icon

Makes current settings the default for new buttons

Deletes selected buttons

Creates a new, blank button

**Shortcut** To quickly open the Button Editor (if the OneClick Editor is closed), hold down the Control key and click the button you want to edit, then choose **Edit Button** from the contextual menu. (This shortcut isn't available for buttons that override OneClick's contextual menu with a custom contextual menu.)
—Or—
If the OneClick Editor is open, but another editor is active, double-click the button you want to edit to switch to the Button Editor.

## Creating a new button

The first step in creating and customizing a button is to add a new button to a palette. After you've added a button, you can change its visual attributes in the Button Editor, add an icon in the Icon Editor, and record or write a script for it in the Script Editor.

▶ **To add a button to a palette**

- Select the palette on which you want to add a button, then click New Button.

  –Or–

- Click the palette on which you want to add a button, then press Command-N.

**Note** Clicking the palette allows the palette to receive keystrokes. If you don't click the palette before pressing Command-N, then the keystroke goes to the active application or the OneClick Editor window (wherever you last clicked).

The new button appears in the first open space on the palette. If the palette is full of buttons and there's no room for a new one, then the palette enlarges itself to hold the new button. If the palette is short and wide, then consecutive buttons are added from left to right; on tall and narrow palettes, buttons are added from top to bottom.

## Selecting buttons to edit

All operations you perform in the Button Editor work on the selected button(s). You must first select a button before choosing options in the Button Editor.

### ▶ To select a button

- Click the button you want to select.

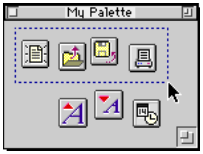
  The selected button's name appears in the Button Editor's Name box. Selection handles appear on the button's corners to indicate it's selected, and a resize box appears in the palette's lower-right corner to indicate the palette is also selected.

### ▶ To select more than one button

- Hold down the Shift key and click additional buttons.

  –Or–

- Click the mouse on the palette's background and drag over the buttons you want to select.



  Any buttons inside (or touching) the selection rectangle become selected.

  –Or–

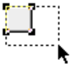
- To select all the buttons on a palette, click the palette's background, then press Command-A.

  After you select multiple buttons, you can change the attributes of the selected buttons all at once.

## Resizing selected buttons

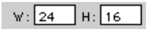
You can resize buttons two different ways.

**▶ To resize selected buttons**



- Drag a resize handle on one of the button's corners.

  –Or–



W: 24 H: 16

- Type numbers in the Width and Height boxes in the Button Editor.

If you have multiple buttons selected, all the selected buttons change size.

You can make a button look like a line by setting its height or width to 1. (Changing the height to 1 draws a horizontal line, changing the width to 1 draws a vertical line.)

## Moving and aligning buttons

When you drag buttons on a palette, the buttons snap to the grid points that are set in the Palette Editor (see "Changing a palette's grid settings and button spacing" on page 36). You can drag a group of buttons by selecting all the buttons in the group, then dragging one of the selected buttons.

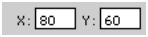
**▶ To nudge a button one pixel in any direction**

- Click the palette's title bar (so the palette can receive keystrokes), then press the arrow keys to nudge the selected button(s).

**▶ To nudge a button by the number of pixels in the grid**

- Click the palette's title bar (so the palette can receive keystrokes), then hold down the Command key and press the arrow keys.

**▶ To move a button to an exact pixel location on the palette**



X: 80 Y: 60

- Type numbers in the X (left) and Y (top) boxes.

  The palette's left and top coordinates start at 0, 0.

## Editing and formatting a button's text label

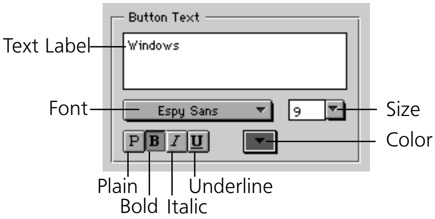
Text you type in the Button Text box appears on the button's face. Use button text to convey the button's purpose if the purpose is difficult to represent with an icon.

### ▶ To change the button's text label

1. Click the Button Text box to make it active (so the Button Editor can receive keystrokes).

2. Type or paste text in the box.

   The length of the button's label is limited only by available memory. If there's too much text to fit on one line in the button, the text wraps automatically.



Button Text

Windows

Espy Sans

9

3. Choose formatting options from the Font, Size, and Color menus, or click the style buttons to apply different styles.

Each button on a palette can have a different text format.

You can change the text formatting for all the selected buttons at once, but you can edit the text for only one button at a time.

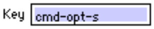
## Adding a keyboard shortcut

Each button on a palette can have its own keyboard shortcut that activates the button. When you press the button's shortcut key, the button's script runs just as if you had clicked the button. A button or its palette do not need to be visible on screen to be triggered by a keyboard shortcut.

**Note** You can override an application's command keys by defining a button with the same shortcut key. If an application and a OneClick button both use the same shortcut key, the OneClick button takes precedence.

### ▶ To assign a shortcut key

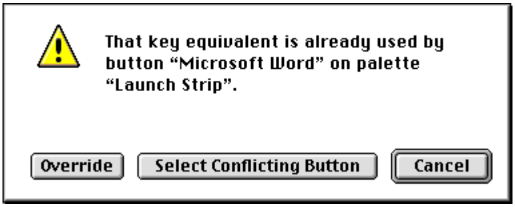
1 Click the Key box to make it active.


Key cmd-opt-s

2 Press a key combination (such as Command-Option-S) or a function key (such as F12).

- Choose a shortcut key that won’t conflict with any command keys you use in other applications.
- No two buttons can have the same shortcut key.

If the key you press is already assigned to another button, a dialog box appears.


That key equivalent is already used by button “Microsoft Word” on palette “Launch Strip”.

3 Do one of the following:

- To go back and type a different shortcut key, click **Cancel**.
- To find the other button that has the same shortcut, click **Select Conflicting Button**. OneClick shows the palette containing the conflicting button and then selects the button so you can change or remove its shortcut key.
- To add the shortcut anyway, click **Override**. OneClick removes the shortcut from the button named in the dialog box and assigns it to the selected button.

To activate the button with the shortcut, close the OneClick Editor, then press the shortcut key.

### ▶ To remove a shortcut key

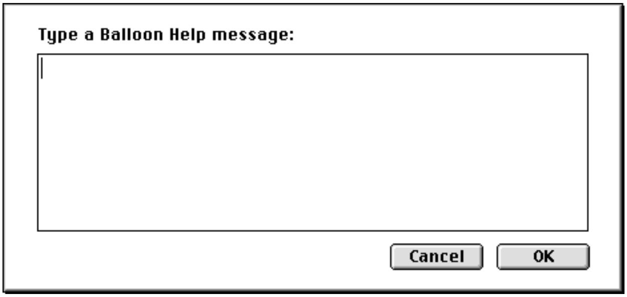
- Click the Key box to select the shortcut, then press Delete.

## Adding a Balloon Help message

Each button can have a help message that appears in Balloon Help and in the Help Status area of the System Bar. Newly-created buttons have no help message.

### ▶ To edit a button's help message

1 Click the balloon button (▭).



Type a Balloon Help message:

2 Type a help message (up to 255 characters) in the dialog box. Press Return to insert blank lines in the message.

3 Click **OK**.

When you turn on Balloon Help (or press Shift-Option) and point to the button, the button's help message appears in a balloon.

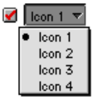
## Choosing which icon appears on a button

Each button can contain up to four different icons, not just one, although a button displays only one icon at a time. You use the Icon pop-up menu to determine which icon (1–4) appears on the button. The default icon is 1.

You can view and edit a button's four different icons in the Icon Editor. A newly-created button contains only blank icons until you edit them using the Icon Editor or Icon Search. For more information on editing and retrieving icons, see "Using the Icon Editor and Icon Search" on page 79.

The most common reason for having different icons in a button are for scripts that switch the button's icon to indicate a different state—on, off, or disabled states, for example. If you don't write scripts, then you'll probably never need to use icons 2, 3, and 4. Leave the setting on Icon 1.

▶ **To choose a different icon**



1. Check the box next to the Icon pop-up menu if it's not already checked.
2. Choose an icon number (1–4) from the pop-up menu.

▶ **To hide a button's icon**

■ Uncheck the box next to the Icon pop-up menu.

OneClick doesn't remove any of the button's stored icons if you uncheck the checkbox. It just doesn't show an icon on the button. You can still view and edit the icons in the Icon Editor.

## Changing a button's name

A button's name doesn't appear anywhere on the button, just in the Button Editor and Script Editor. In a button's script, you can refer to other buttons by their names, so each button on a palette should have a unique name. A button name can be up to 31 characters long.

▶ **To change a button's name**

■ Type a new name in the Name box.

You can rename only one button at a time.

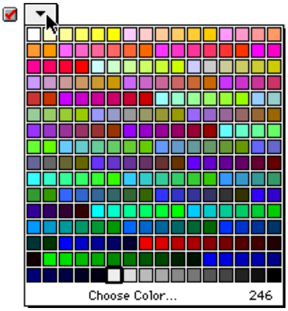
## Changing other visual properties of buttons

For the following properties in the Button Editor, you can apply properties to more than one button at a time. To do so, select multiple buttons and then choose the new property (color, border style, appearance, and so on).

▶ **To change a button's color**

1. Check the checkbox next to the Color pop-up menu if it's not already checked.

**2** Choose a new color from the pop-up menu.



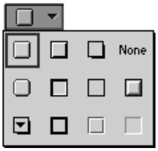
To choose a color not shown in the menu, select **Choose Color** to open the Color Picker dialog box. (See "Choosing a custom color" on page 33.)

### ▶ To make a button transparent (so the palette's background shows through)

- Uncheck the box next to the Color pop-up menu.

### ▶ To change a button's border style

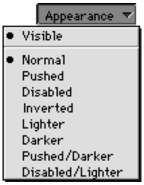
- Choose a new border style from the Border pop-up menu.



There are a dozen border styles to choose from. The "None" style means the button has no border—only the text, color, and icon (if any) appear on the button.

By choosing the pop-up menu (▾) style, you can automatically add a downward-pointing triangle to the button's right side. Use this style for buttons that behave like pop-up menus or tear-off palettes. For more information, see PopupMenu, PopupFiles, and PopupPalette in the *OneClick Authoring Guide*.

▶ **To change a button's visual appearance (highlighting)**



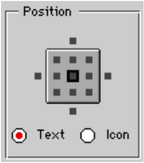
- Choose an option from the bottom part of the Appearance pop-up menu.

  Choosing an option changes the look of the button (its highlighting or shading, or its pushed-in/popped-out appearance). "Lighter" and "Darker" make the whole button (text, color, and icon) look either 50% lighter or 50% darker. "Disabled" means the button won't push in or highlight when you click it. Disabled buttons still work normally when you click them.

- Deselect the Visible option to prevent the button from appearing on the palette.

  Invisible buttons don't appear on the palette when the OneClick Editor is closed; they appear only when you open the OneClick Editor and select the palette. Invisible buttons are used mainly as hidden subroutine buttons for scripts—you can't see or click an invisible button, but you can call its script from another button's script.

▶ **To align a button's text label or icon in the button**



1. In the Position box, click the Text or Icon option to specify what you want to align.
2. Click a point on the Position grid to move the button's text or icon.

   If the text or icon doesn't appear to move when you click different points, try making the button larger. Then click the points again to see the effect.

## Changing the default settings for new buttons

You can change many of the default settings for new buttons you create. Doing so lets you make several new buttons that all have the same look and feel, and you don't have to change each button's settings in the Button Editor after creating them. Default settings you can change include the following:

- border style
- color
- size
- text formatting (font, size, style, and color)

- text position

▶ **To change the default settings for new buttons**

1. Change the settings in the Button Editor to the settings you want new buttons to have (border style, size, and so on).
2. Click **Make Default**.

Using the Make Default feature is also a quick way to copy the characteristics of one button to new buttons you create. Just select a button, click **Make Default**, then click **New Button**.

## Duplicating buttons

You can use either keyboard commands or drag and drop to make copies of buttons. Duplicating a button copies all button properties, including the button's icon, script, and all settings in the Button Editor.

▶ **To duplicate a button**

- Select the button to duplicate, then press Command-D.

  The new button appears offset below and to the right of the original button.

▶ **To copy a button from one palette to another**

- Drag the button to another palette.

▶ **To move a button from one palette to another**

- Hold down the Option key and drag the button to another palette.

## Deleting buttons from a palette

There are several ways you can delete buttons depending on whether the OneClick Editor is open or closed.

### ▶ To delete selected buttons with the OneClick Editor open

- Press the Delete key.

  –Or–

- Click **Delete Button** in the Button Editor.

### ▶ To undo (restore) a deleted button

- Click the palette containing the buttons you deleted, then press Command-Z.

**Note** To undo a deleted button, you must do so before closing the OneClick Editor, selecting a different palette to edit, or deleting additional buttons. Otherwise the deleted button cannot be restored.

### ▶ To delete a button when the OneClick Editor is closed

- Control-click the button to delete, then choose **Delete Button** from the contextual menu. (This shortcut isn't available for buttons that override OneClick's contextual menu with a custom contextual menu.)

# Chapter 5
# Using the Script Editor

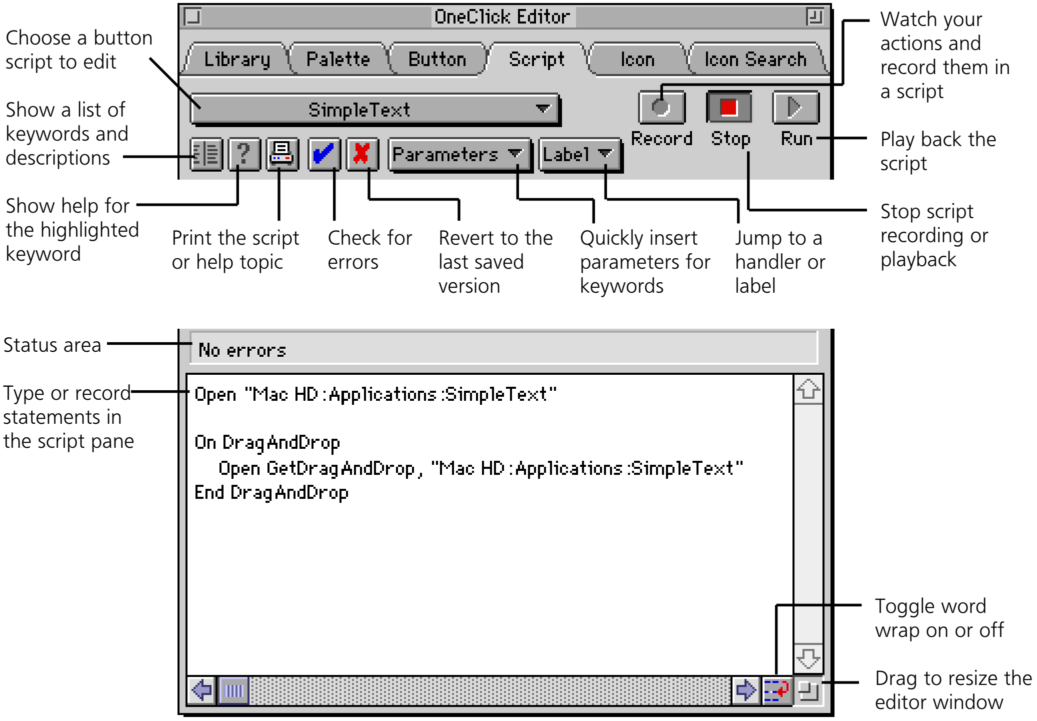
This chapter describes all the features of the Script Editor. The chapter covers the following topics:

- 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.


OneClick Editor
Library
Palette
Button
Script
Icon
Icon Search
SimpleText
Parameters
Label
Record
Stop
Run
Choose a button script to edit
Show a list of keywords and descriptions
Show help for the highlighted keyword
Print the script or help topic
Check for errors
Revert to the last saved version
Quickly insert parameters for keywords
Jump to a handler or label
Watch your actions and record them in a script
Play back the script
Stop script recording or playback
Status area
Type or record statements in the script pane
Toggle word wrap on or off
Drag to resize the editor window

```
No errors
```

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

On DragAndDrop
   Open GetDragAndDrop, "Mac HD:Applications:SimpleText"
End DragAndDrop
```

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.

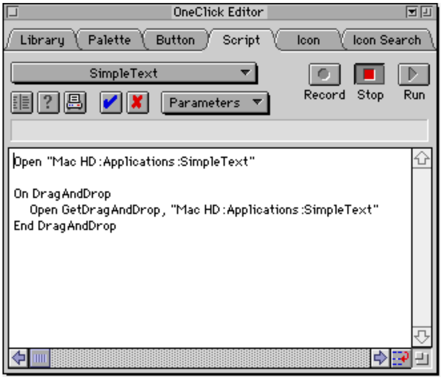
### ▶ To display a button's script

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

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.



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

On DragAndDrop
   Open GetDragAndDrop, "Mac HD:Applications:SimpleText"
End DragAndDrop
```

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 New Button** from the pop-up menu. |
| Edit a button's script when the OneClick Editor window is closed | Command-Option-click the button, then choose **Edit Script** from the pop-up menu. |
| Edit a different button's script while the Script Editor is active | Click the button to edit or choose the button's name from the pop-up menu in the Script Editor. |

# 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.

   When recording clicks on unnamed buttons in a dialog box or window, OneClick records the SelectButton statement with the dialog button's index number instead of a name. The SelectButton statement plays back normally.

   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** While recording is in progress, you can click other buttons to run their scripts, but those actions won't be recorded into the script you are recording.

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). |
| Delete the character to the right of the cursor (forward delete) | Press Shift-Delete. |

## Jumping directly to a line in a script

The Script Editor's Label pop-up menu is useful for navigating long scripts. The menu lists all of the handlers and any labels in the script. Selecting an item from the menu jumps immediately to that handler or label.

To put a label in the script, add a comment with all or part of the comment's text within angle brackets ("< >"). For example, either of the following adds the label "get the list of files" to the Label menu.

```
// <get the list of files>
// This routine will <get the list of files> from the Data folder
```

The comment must be the only statement on the line. The following will not work.

```
Variable A   // This is <Variable A>
```

## 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 76 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**

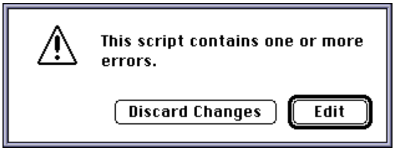
- Press Command-S.

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:



This script contains one or more errors.

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.

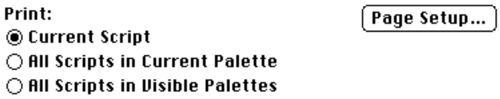
# Printing scripts

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.



Print:
● Current Script
○ All Scripts in Current Palette
○ All Scripts in Visible Palettes

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.

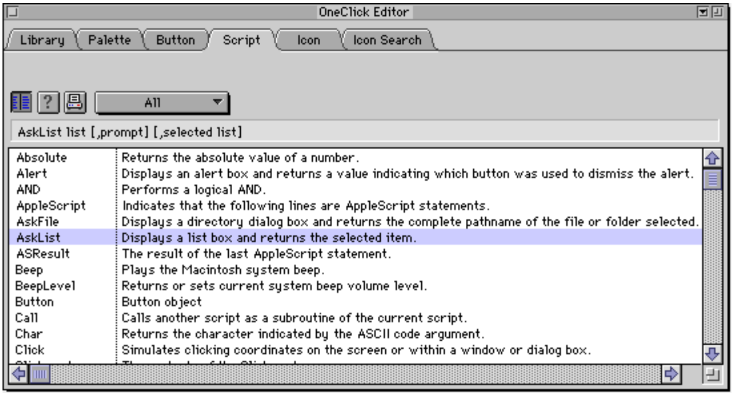
## Using the Keyword List

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.



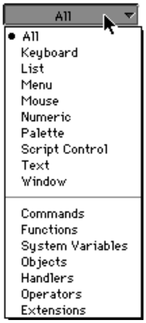
AskList list [,prompt] [,selected list]

| | |
|---|---|
| Absolute | Returns the absolute value of a number. |
| Alert | Displays an alert box and returns a value indicating which button was used to dismiss the alert. |
| AND | Performs a logical AND. |
| AppleScript | Indicates that the following lines are AppleScript statements. |
| AskFile | Displays a directory dialog box and returns the complete pathname of the file or folder selected. |
| AskList | Displays a list box and returns the selected item. |
| ASResult | The result of the last AppleScript statement. |
| Beep | Plays the Macintosh system beep. |
| BeepLevel | Returns or sets current system beep volume level. |
| Button | Button object |
| Call | Calls another script as a subroutine of the current script. |
| Char | Returns the character indicated by the ASCII code argument. |
| Click | Simulates clicking coordinates on the screen or within a window or dialog box. |

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 8, "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.

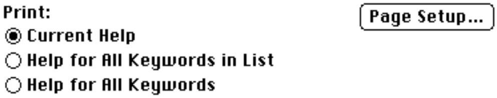
## Printing keyword help

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.



Print:
● Current Help
○ Help for All Keywords in List
○ Help for All Keywords

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.

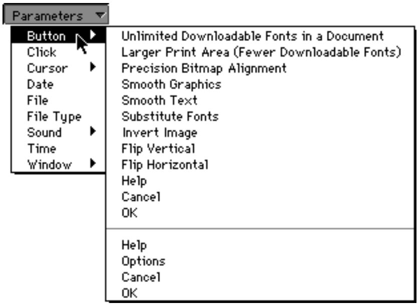
3 If the option you choose displays a dialog box, choose options in the dialog box and click **OK**.

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.



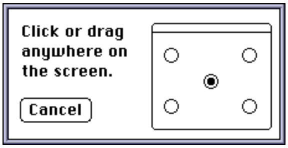
Button submenu when the Page Setup Options dialog box is open in an application.

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

For more information, see "SelectButton command" on page 287 and "DialogButton object" on page 187.

## 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:



Click or drag anywhere on the screen.

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.

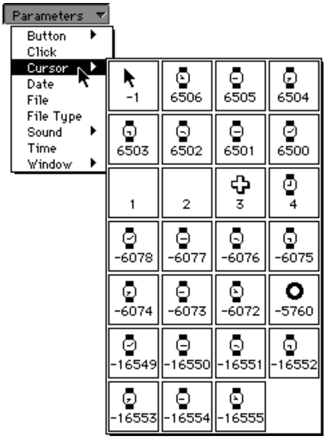
When you click and release the mouse button, OneClick inserts the mouse click's coordinates in the script. If you click within a window, the coordinates are local to the window; if you click outside of a window (such as on the desktop), the keyword Global is also inserted, indicating the coordinates 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 command” on page 172 for more information.

## Cursor

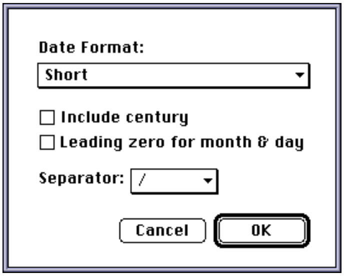
Use the Cursor submenu to insert the ID number of a cursor (for use with the Cursor system variable). The Cursor submenu shows all the cursors available in the System file and the active application, with the ID number of each cursor. Choosing a cursor from the submenu inserts its ID number in the script.



See “Cursor system variable” on page 180 for more information.

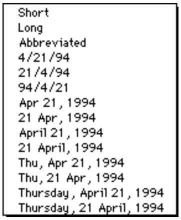
## Date

Use the Date option to insert a date format number for use with the DateTime object’s DateString property. When you choose Date, the Date Format dialog box appears:


Date Format: Short

☐ Include century
☐ Leading zero for month & day

Separator: /



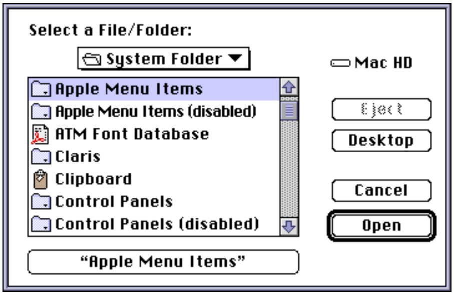
Date Format pop-up menu

Choose options in the dialog box to specify the format you want the DateString property 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 ".DateString" on page 182 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.

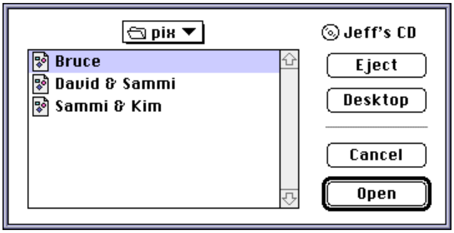
To choose a file, locate the file and then click **Select**, or click the button at the bottom of the dialog box. To choose a folder, locate the folder and then click the button at the bottom of the dialog box.

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.



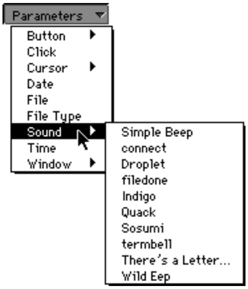
Choose the file whose file type code you want to insert, then click **Open**. OneClick inserts in the script the file type code of the chosen file.

The following keywords can use a file type parameter:

- AskFile function (page 152)
- PopupFiles function (page 264)
- File.Kind (page 202)

## 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.

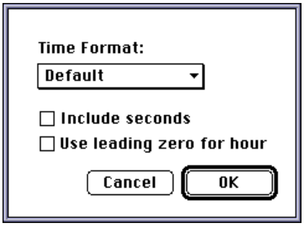


OneClick inserts in the script the name of the sound chosen from the submenu.

See “Sound command” on page 292 for more information.

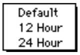
## Time

Use the Time option to insert a time format number for use with the DateTime object’s TimeString property. When you choose Time, the Time Format dialog box appears:


Time Format: Default

□ Include seconds

□ Use leading zero for hour



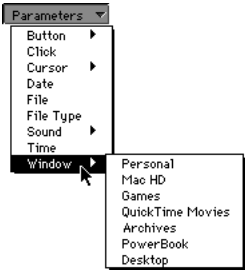
Time Format pop-up menu

Choose options in the dialog box to specify the format you want the TimeString property 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 “.TimeString” on page 185 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 object" on page 305 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.

## Cannot use a function as 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 94.

## 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.

## Expected "End If," "End For," "End *handler-name*,", and so on

An End statement is missing from a block of statements, such as an If statement, a For loop, a Repeat loop, a While loop, or a handler. The cursor is placed on the line nearest where the missing End statement was expected.

## 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.

## Unexpected "End"

The compiler found an incorrect or extra End 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.

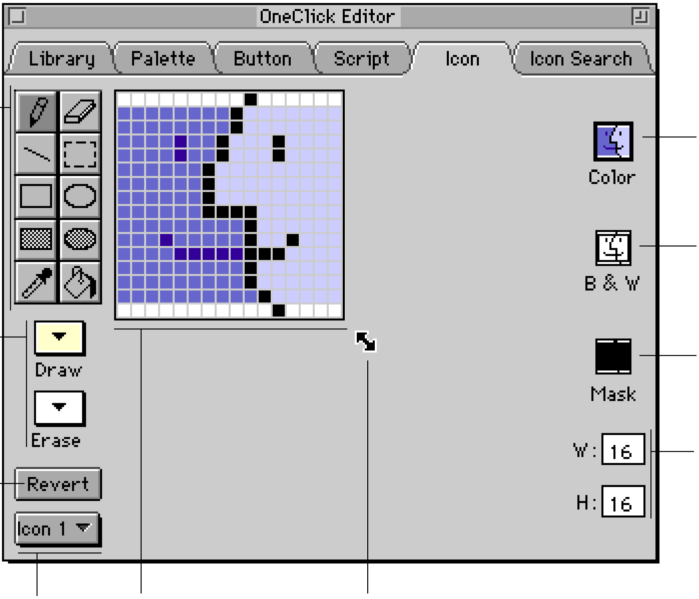
### Unknown version of script

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 6

# Using the Icon Editor and Icon Search

OneClick includes an Icon Editor you can use to create button icons or to edit icons taken from other files. You can create 256-color icons of any size up to 32 by 32 pixels. The Icon Editor lets you create both color and black-and-white versions of the same icon; OneClick displays the appropriate icon on the button depending on whether you're using a color or black-and-white monitor.



W: 16

H: 16

Click a tool, then click or drag in the drawing area to change the icon

Choose drawing and erasing colors from the pop-up menus

Restore the last saved version of the icon

Click to edit the icon's color version

Click to edit the icon's black & white version

Click to edit the icon's mask

Type numbers to change the icon's width or height

Choose a button icon to edit (1–4) from the pop-up menu

Click or drag in the editing area to change the color of pixels

Drag below the editing area's bottom-right corner to change the icon's size

### ▶ To edit a button’s icon

1. Open the OneClick Editor and click the **Icon** tab.
2. Select the button whose icon you want to edit.

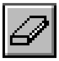
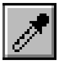
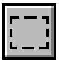
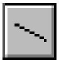
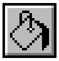
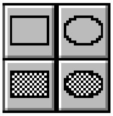
   The button’s icon appears in the drawing area. If the button didn’t previously have an icon, a blank icon appears instead.
3. Choose which icon (1–4) you want to edit by choosing a number from the Icon pop-up menu.

   A button’s default icon is 1, but each button can store and use up to four different icons. You can choose which icon appears on a button by setting the button’s icon in the Button Editor or by setting the button’s Button.Icon property in a script.
4. Use the Icon Editor’s drawing tools to change the icon image.

## Icon Editor tools

The tools let you draw in the drawing area much like a paint program.

| Tool Icon | Name | What it does |
|---|---|---|
| ▾<br>Draw | Draw color | Changes the current draw color used by the pencil, line, fill, and shape tools. Click to choose a color from the pop-up menu.<br>If your monitor displays 256 or more colors, hold down the Option key and click the color pop-up to get a smaller menu of 34 colors recommended by Apple for use in icons. (This also works for other OneClick color pop-ups, not just the icon draw color.)<br>To choose a color not shown in the menu, select **Choose Color** to open the Color Picker dialog box. (See “Choosing a custom color” on page 33.) |
| ▾<br>Erase | Erase color | Changes the current erase color used by the eraser and selection tools. Click to choose a color from the pop-up menu. |
| ✎ | Pencil | Changes the color of individual pixels in the drawing area. Click the pencil tool, then click pixels in the drawing area. |

| Tool Icon | Name | What it does |
|---|---|---|
|  | Eraser | Erases pixels. Click the eraser tool, then click pixels in the drawing area to erase. The eraser uses the current erase color. |
|  | Dropper | Changes the current draw color to a color in the drawing area. Click the dropper tool, then click a color in the drawing area to "suck up" the color and make it the current draw color. To pick up the erase color instead, hold down the Option key. Holding down the Option key with any other tool selected causes the Dropper to appear. |
|  | Selection | Selects rectangular parts of the drawing area. Click the selection tool, then drag to select part of the icon. After selecting, you can drag the selection to move it, Option-drag to create a copy, press Command-C to copy the selection to the clipboard, or press Delete to erase the selection. To select the entire icon, press Command-A. |
|  | Line | Draws line segments. Click the line tool, then drag in the drawing area to draw lines. |
|  | Fill | Fills a colored area of the icon with the current draw color. Click the fill tool, then click a color in the drawing area to fill the area with color. To fill all pixels of the same color (not just adjacent pixels), Command-click a color in the drawing area. |
|  | Shapes | Draws hollow or filled shapes. Click a shape tool, then drag to draw a shape. |

## Resizing the icon

If you edit the icon for a new button that didn't previously have an icon, then the icon's default size is the same size as the button. You can change the icon's size to any size up to 32 by 32 pixels.

### ▶ To resize the icon

■ Type numbers in the Width and Height boxes.

–Or–

1 Move the pointer over the lower-right corner of the drawing area.

2. When the cursor changes to the resize cursor (⇲), drag to resize the icon.

**Tip** Resize an icon so that no unused space appears on the right or bottom edges. Doing so allows the icon to appear centered correctly on the button.

## Pasting an icon or picture from the clipboard

You can copy an icon from a Get Info window in the Finder, or copy a graphic from a graphics program, and then paste the graphic in the Icon Editor.

### ▶ To paste a graphic in the Icon Editor

1. Copy a graphic (up to 32 by 32 pixels large) in a graphics program. (To copy a Finder icon, select the icon and choose **Get Info**, then click the icon in the Info window and choose **Copy**.)
2. Click the Icon Editor's drawing area to make it active.
3. Press Command-V to paste.

**Tip** You can also copy an icon from the Icon Editor and paste it into a Get Info window or another application.

## Designing both color and black-and-white icons

Each icon has both a 256-color version and a black-and-white version. The black-and-white icon appears only on monitors that are set to display fewer than 16 colors (in the Monitors control panel).

### ▶ To edit the black-and-white icon

- Click the **B & W** box.

  The black-and-white icon appears in the drawing area. The Draw and Erase color menus show black and white as the only choices.

If you leave the black-and-white icon empty (all white pixels), then OneClick approximates a black-and-white version of the color icon for display on the button. (Light colors change to white, dark colors change to black.) You don't need to create a black-and-white icon unless the approximation looks poor.

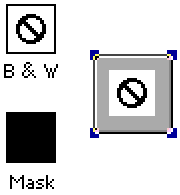
▶ **To copy the color icon to the black-and-white icon and touch it up**

1. Click the **Color** box to switch to the color version.
2. Drag the sample icon from the **Color** box to the **B & W** box.
3. Click the **B & W** box to switch to the black-and-white version.
4. Use the drawing tools to turn pixels on and off in the black-and-white icon.

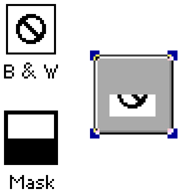
## Making parts of the icon transparent

An icon's mask lets you determine which parts of an icon's image appears on a button and which parts are transparent, allowing the button's background color to show through. Careful mask editing lets you create irregularly-shaped icons and icons that appear to have holes in them.

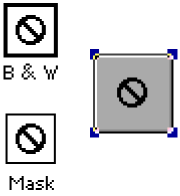
When a mask pixel is black, the corresponding pixel in the icon appears on the button. If a mask pixel is white, then the corresponding icon pixel doesn't appear. Following are three examples of how black pixels in different masks affect the same icon on a medium-gray button.



Full mask (the default)



Only half the icon is masked



Custom mask (after draggin
the icon to the mask)

▶ **To copy the color or black-and-white icon to the mask and touch it up**

1. Click the **Color** or **B & W** sample box to select it.
2. Drag the icon from either the **Color** or **B & W** box to the **Mask** box.

Dark colors in the icon change to black in the mask, light colors change to white.

3 Click the **Mask** box to select it.

4 Use the drawing tools to turn pixels on and off in the mask.

## Saving changes to an icon

OneClick automatically saves any changes to a button's icon when you switch to another button or close the Icon Editor, so you don't need to manually save it. However, the changed icon doesn't appear on the button until you save it.

▶ **To save the icon and update the button (without closing the editor)**

■ Press Command-S.

The new icon appears on the selected button.

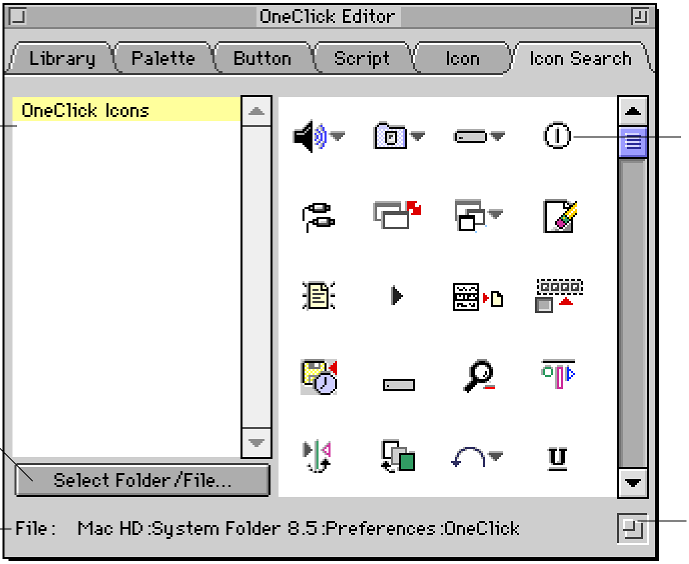
▶ **To discard changes and restore the original icon**

■ Click **Revert**.

The last saved icon appears in the drawing area.

# Using Icon Search

The Icon Search feature lets you use icons from any file that contains icon, cursor, or picture resources. Icon Search scavenges through folders on your hard disk and displays a list of icons which you can then drag to palettes and buttons.



OneClick Icons

File: Mac HD:System Folder 8.5:Preferences:OneClick

Select a file to display its icons in the icon list

Click to search a folder for files containing icons

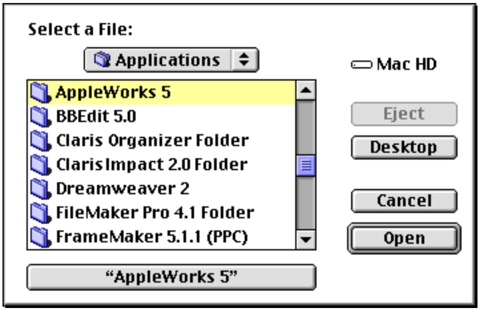
The path to the selected file

Drag an icon to a button to replace the button's icon, or drag to a palette to create a new button

Drag to resize the Icon Search window

### ▶ To search for icons

1. In the OneClick Editor window, click the **Icon Search** tab.
2. Click **Select Folder/File**.
3. Use the directory dialog box to locate the folder or file you want to search.



AppleWorks 5
BBEdit 5.0
Claris Organizer Folder
ClarisImpact 2.0 Folder
Dreamweaver 2
FileMaker Pro 4.1 Folder
FrameMaker 5.1.1 (PPC)

The selected item’s name appears in the button below the list box.

4. Click the button below the list box to begin the search.

If you selected a folder, Icon Search recursively searches through all the files and folders within the folder you select. Searching a folder that contains a lot of files and folders (such as your System Folder) may take a few moments.

A file list appears on the left when the search is complete.

### ▶ To copy an icon from a file to a button

1. Click a file in the list to display any icons, cursors, and pictures found in the file.

   In some files you may notice that icons appear more than once. These are usually 16- and 256-color versions of the same icon where similar colors are used in each version.

2. Drag an icon from the Icon Search list to a button or a palette:

   - To create a new button with an icon, drag an icon from the list to an empty space on a palette.
   - To change the icon of an existing button, drag an icon to a button.

   When you use Icon Search to create a new button, OneClick makes the new button the same size as the icon.

**Tip** The OneClick Icons file (inside the OneClick Folder in Preferences) contains a large selection of professionally-designed icons for use on OneClick buttons. Online services such as America Online are also a good source for icon files.

Chapter 7

# 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:

- About scripting
- Parts of the EasyScript language
- Common scripting techniques
- Testing and debugging a script
- Specifications and limits

## 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 8, "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
- Functions
- Comments
- Variables
- Expressions and operators
- Control statements (branching and looping)
- 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 (<reutrn>). Examples of lists include the following:

```
"Apple<return>Banana<return>Navel Orange<return>Strawberry<return>Peach"
"9<return>26<return>66<return>7<return>13<return>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.

You don't need to insert the <return> tag before the first item or after the last item in the list, just between each item.

**Tip** To quickly insert the <return> tag in the Script Editor, press Option-Return.

For more information on how to use lists in your scripts, see "Manipulating lists" on page 120.

## 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<return>Oranges<return>Bananas<return>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<return>Dodgers<return>Giants", "Pick your favorite."
```

```
// 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 11/9/99" |
| | | 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<reutrn>Viki<return>Erica"

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

// Type the salutation
Type “Dear ", MyChoice, ", <return><return>"
```

If statements can be nested to create even more complex conditions. For more information, see “If, Else, Else If, End If commands” on page 228.

## 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 120.*)

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 commands" on page 219.

## 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.Name <> "")
    SelectMenu "File", "Save"
    CloseWindow
End While
Message "All done."
```

The script works by repeatedly checking the Window.Name 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.Name <> “”) is tested. If it is true (Window.Name 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.Name 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 commands” on page 304.

## 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.Name = "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 command” on page 303.

### 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.Name <> "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 | DateTime | DialogButton | File | Menu | Palette | Process | Screen | Volume | Window |
|---|---|---|---|---|---|---|---|---|---|---|
| .Append | | | | m | | | | | | |
| .Border | r/w | | | | | | | | | |
| .Busy | | | | r/o | | | | | | |
| .Checked | | | r/o | | r/o | | | | | |
| .Collapsed | | | | | | | | | | r/w |
| .Color | r/w | | | | | r/w | | r/w | | |
| .Count | r/o | | r/o | r/o | m | r/o | r/o | r/o | r/o | r/o |
| .CreationDate | | | | r/o | | | | | | |
| .Creator | | | | r/w | | | r/o | | | |
| .CursorScreen | | | | | | | | r/o | | |
| .CursorX | | | | | | | | r/w | | |
| .CursorY | | | | | | | | r/w | | |
| .Data | r/w | | | | | | | | | |
| .DateSerial | | r/o | | | | | | | | |
| .DateString | | r/o | | | | | | | | |
| .Day | | r/o | | | | | | | | |
| .Delete | m | | | m | | m | | | | |
| .Depth | | | | | | | | r/w | | |
| .Drag | m | | | | | m | | | | |
| .Eject | | | | | | | | | m | |
| .Enabled | | | r/o | | r/o | | | | | |
| .Exists | r/o | | r/o | r/o | r/o | r/o | r/o | r/o | r/o | r/o |
| .FileVersion | | | | r/o | | | | | | |
| .Folder | | | | | | | r/o | | | |
| .Free | | | | | | | r/o | | r/o | |

Key: r/w=Read/write r/o=Read-only w/o=Write-only m=Message

| **Properties & Messages** / **Objects** | Button | DateTime | DialogButton | File | Menu | Palette | Process | Screen | Volume | Window |
|---|---|---|---|---|---|---|---|---|---|---|
| .Front | | | | | | m | m | | | m |
| .Grow | | | | | | m | | | | |
| .Height | r/w | | | | r/o | r/w | | r/o | | r/w |
| .Help | r/w | | | | | | | | | |
| .Hour | | r/o | | | | | | | | |
| .Icon | r/w | | | | | | | | | |
| .IconAlign | r/w | | | | | | | | | |
| .Index | r/w | | r/o | | r/o | r/o | r/o | | r/o | r/o |
| .InMenu | | | | | | r/w | | | | |
| .IsGlobal | | | | | | r/w | | | | |
| .KeyShortCut | r/w | | | | | | | | | |
| .Kind | | | | r/w | | | r/o | | | r/o |
| .KindString | | | | r/o | | | | | | |
| .Left | r/w | | | | | r/w | | r/o | | r/w |
| .List | r/o | | r/o | r/o | r/o | r/o | r/o | | r/o | r/o |
| .Location | w/o | | | | | w/o | | | | w/o |
| .Locked | | | | r/w | | | | | | |
| .MainScreen | | | | | | r/o | | | | |
| .Maximum | | | | | | | | r/o | | |
| .Minute | | r/o | | | | | | | | |
| .Mode | r/w | | | | | | | | | |
| .ModificationDate | | | | r/o | | | | | | |
| .Month | | r/o | | | | | | | | |
| .Name | r/w | | r/o | r/w | r/o | r/w | r/o | | r/o | r/o |
| .New | m | | | | | m | | | | |
| .NewFolder | | | | m | | | | | | |
| .Original | | | | r/o | | | | | | |
| .PICT | | | | | | w/o | | | | |
| .Quit | | | | | | | m | | | |

Key: r/w=Read/write r/o=Read-only w/o=Write-only m=Message

| **Properties & Messages** / **Objects** | Button | DateTime | DialogButton | File | Menu | Palette | Process | Screen | Volume | Window |
|---|---|---|---|---|---|---|---|---|---|---|
| .Record | r/w | | | | | | | | | |
| .Script | r/w | | | | | | | | | |
| .Second | | r/o | | | | | | | | |
| .Selection | | | | | | | r/w | | | |
| .SendAE | | | | | | | m | | | |
| .Size | w/o | | | r/o | | w/o | r/o | | r/o | w/o |
| .Text | r/w | | | r/w | | | | | | |
| .TextAlign | r/w | | | | | | | | | |
| .TextColor | r/w | | | | | | | | | |
| .TextFont | r/w | | | | | | | | | |
| .TextSize | r/w | | | | | | | | | |
| .TextStyle | r/w | | | | | | | | | |
| .TimeSerial | | r/o | | | | | | | | |
| .TimeString | | r/o | | | | | | | | |
| .TitleBar | | | | | | r/w | | | | r/o |
| .Top | r/w | | | | | r/w | | r/o | | r/w |
| .Unmount | | | | | | | | | m | |
| .Update | m | | | | m | m | | m | | m |
| .Visible | r/w | | | r/w | | r/w | r/w | | | r/w |
| .Weekday | | r/o | | | | | | | | |
| .Width | r/w | | | | | r/w | | r/o | | r/w |
| .Window | | | | | | | r/o | | | |
| .Year | | r/o | | | | | | | | |
| .Zoom | | | | | | | | | | r/w |

Key: r/w=Read/write r/o=Read-only w/o=Write-only m=Message

For more detailed information about objects and their associated properties and messages, refer to the descriptions of each object in Chapter 8, "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:<br>• the application starts up (for global palettes, Startup handlers execute after the computer starts up)<br>• a button assigns a script containing a Startup handler to another button<br>• you edit a script containing a Startup handler and then close the OneClick Editor window<br>• you import a palette that contains a Startup handler in one of its scripts<br>• 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 8, "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 8, "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 136.*) 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<return>Banana<return>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<return>Banana<return>Orange<return>Strawberry<return>Peach"
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 <return> (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<return>Renstrom, R./555-5721/25<return>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 8, "EasyScript Reference."

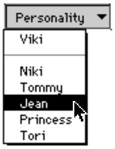
| Keyword | Description |
|---|---|
| AskList | Displays a list in a list box and returns a list of the selected item(s). |
| GetDragAndDrop | Returns a list of paths when multiple Finder items are dropped on a button. |
| GetResources | Returns a list of resources of the specified resource type (sound, 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. |

| Keyword | Description |
|---|---|
| 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<return>–<return>Niki<return>Jean<return>Tori"
Message "You chose " & theChoice
```

**update screenshot to reflect script change**

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 8, "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 ListSort function sorts alphabetically. The result of the ListSort 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"
```

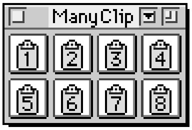
## Storing Clipboard data in static variables

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 system variable" on page 174 and "ConvertClip command" on page 179.

## 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 8, “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
```

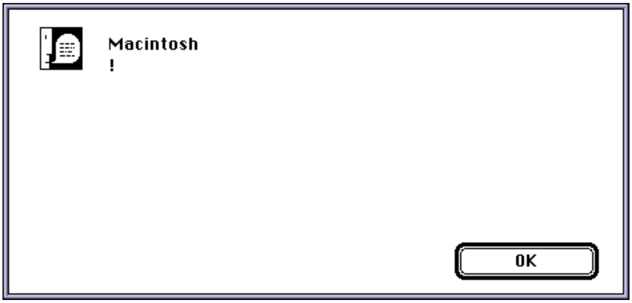
When you run the script, a message box appears showing the value of fileCount when the script reaches the Message statement. When you have the script working the way you expect and no longer need the message box, simply delete the Message statement.

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:

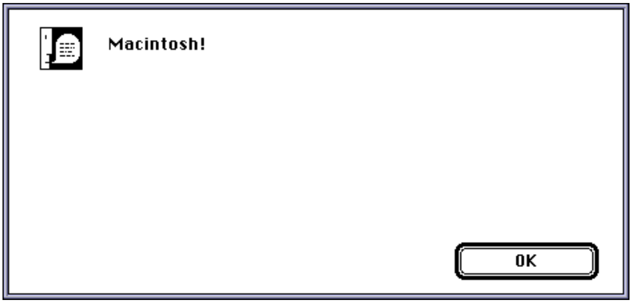
```
Message myStringVar & "!"
```

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



```
Macintosh
!
```

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:



Macintosh!

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

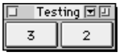
If you have more than a couple of variables you want to monitor continuously, or you don't want message boxes interrupting your script (using the previous technique), you can use buttons as text displays to show the current values of variables. The Button.Text property changes a button's text label; 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 196 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 (2 GB 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 | 32767 characters |
| 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 8

# EasySript Reference

## Using the EasyScript Reference

This chapter contains detailed information about each keyword in the EasyScript language and covers all commands, functions, system variables, objects, and handlers. Keywords are listed alphabetically.

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

For information on values, expressions, operators, iteration, and other EasyScript language concepts, see Chapter 7, "Using EasyScript."

### What's new in OneClick 2.0

If you previously wrote your own scripts with OneClick 1.0.3 or earlier, review this section first to find out what's new and changed in EasyScript for OneClick 2.0. If you are new to scripting with EasyScript, you can skip this section and jump ahead to the keyword reference beginning on page 149.

In most cases, existing scripts for OneClick 1.x will run without changes in OneClick 2.0. Because of new keywords added in EasyScript, however, you may need to edit some of the scripts you've written to avoid name conflicts between your variable names and the names of new keywords. For example, the words Data, Editor, and True are no longer valid variable names because they are now EasyScript keywords.

You shouldn't need to edit your scripts right away—a script with a conflicting name will run fine until you attemp to edit and save it. When you try to save a script with a variable name that conflicts with a new EasyScript keyword, OneClick displays "Invalid variable name" and highlights the variable name you'll need to change.

Following is a list of all the new and changed keywords in EasyScript.

| | | |
|---|---|---|
| Alias function | New | Returns an alias for the specified file as a string. |
| AskButton function | Changed | Now supports up to eight buttons. |
| AskFile function | Changed | You can now supply a custom prompt for the dialog box. |
| AskKey function | New | Prompts you to press a key and returns the keystroke in text format. |
| AskNewFile function | New | Displays a Save As dialog box and returns a file path. |
| AskText function | Changed | Now allows you to supply a custom value that is returned when the user clicks Cancel. |
| Button.Data property | New | Unlimited-length persistent storage for buttons. |
| CallResult system variable | New | A global variable you can use to save or pass information between calls to different handlers or scripts. |
| CloseResFile command | New | Closes a file's resource fork and removes it from the current resource chain. |
| ColorPicker function | New | Displays the color picker dialog box and returns the RGB values of the chosen color. |
| ContextualMenu handler | New | Indicates statements to execute when you Control-click the button, overriding the default OneClick contextual menu and MouseDown handler. |
| Cursor system variable | Changed | Can now set the active cursor in addition to getting its ID number. |
| Editor command | New | Opens the OneClick Editor window to the specified tab, automatically selecting the specified button or palette. |
| EditorFont system variable | New | Gets or sets the font displayed in the Script Editor. |
| EditorSize system variable | New | Gets or sets the font size displayed in the Script Editor. |
| False system variable | New | Returns the number 0 (zero). |
| File.Busy property | New | Returns True (1) if the specified file is open for reading or writing. |
| File.CreationDate property | New | Returns a file's creation date and time. |
| File.FileVersion property | New | Returns a file's version number. |
| File.KindString property | New | Returns a file's type as it appears in the Kind column of Finder windows |
| File.ModificationDate property | New | Returns a file's modification date and time. |
| File.Name property | Changed | Can now return the file name portion of a full path. |

| | | |
|---|---|---|
| File.Visible property | Changed | Now works on folders as well as files. |
| FinderAlias command | New | Makes aliases of the specified files. |
| FileClose command | New | Closes a file opened with FileOpen. |
| FileGetEOF function | New | Gets the end-of-file (length) of a file opened with FileOpen. |
| FileGetPos function | New | Gets the file mark (position for reads/writes) of the specified file opened with FileOpen. |
| FileOpen function | New | Opens a file for reading/writing and returns a reference number for use with other FileIO commands. |
| FileRead function | New | Reads the specified amount of data from a file into a variable. |
| FileSetEOF command | New | Sets the end-of-file (length) of a file opened with FileOpen. |
| FileSetPos command | New | Sets the current mark (position) from which future reading/writing will occur in a file opened with FileOpen. |
| FileWrite command | New | Writes data to a file opened with FileOpen, starting at the current file mark. |
| FinderCopy command,<br>FinderMove command | Changed | Now allows script to wait until Finder operation completes before continuing execution. |
| FKey command | New | Executes the specified FKey (0-9). |
| FontMenu function | New | Pops up a WYSIWYG font menu and returns the name of the selected font. |
| For, Repeat, While commands | Changed | Switching applications within a loop no longer causes the loop to stop executing. |
| GetICHelpers function | New | Returns the list of helper applications from Internet Preferences. |
| GetICPref function | New | Returns the value of the specified preference from Internet Preferences. |
| GetPalettes function | New | Returns a list of names of all palettes in a palette file. |
| GetResIDList function | New | Returns a list of resource IDs of the specified type. |
| GetResNameList function | New | Returns a list of resource names of the specified type. |
| GetResource function | New | Retrieves the data from the specified resource in the specified file. |
| GetResTypeList function | New | Returns a list of resource types in the specified file. |
| GetScrap function | New | Returns the data of the specified resource type from the Clipboard. |
| GetStringList function | New | Retrieves the strings in the specified STR# resource. |

| | | |
|---|---|---|
| GetWindowText command | New | Captures text from the specified window and puts it in a variable. |
| IgnoreClicks system variable | New | Causes the system to ignore all mouse activity except clicks on OneClick palettes. |
| LaunchURL command | New | Launches a URL using a helper application set in Internet Config. |
| ListCount function | Changed | Now counts the null item at the end of a list if the list ends with the list delimiter. |
| ListDelete function | New | Deletes items from a list and returns the new list. |
| ListFind function | New | Returns the position (index) of an item in a list. |
| ListInsert function | New | Inserts new items into a list and returns the new list. |
| ListItems function | Changed | Now returns the null item at the end of a list if the list ends with the list delimiter. |
| LoadExtensions command | New | Loads or reloads OneClick extensions from the specified file. |
| KeyPress command | New | Simulates a key press of the specified key. |
| MenuNumber function | New | Returns the item number of the last item selected with the PopupMenu function. |
| MountVolume command | New | Mounts an AppleShare server volume over AppleTalk. |
| MountVolumeIP command | New | Mounts an AppleShare server volume over TCP/IP. |
| Notify command | New | Displays a message in a notification dialog box or floating window. |
| OldListCount function | Obsolete | The OneClick 1.0 version of ListCount. Use the new ListCount instead. |
| OldListItems function | Obsolete | The OneClick 1.0 version of ListItems. Use the new ListItems instead. |
| OnlineHelp handler | New | Used by the Online Help and Online Help Editor palettes for displaying customized palette help. |
| OpenFileList function | New | Returns a list containing the paths of all open files. |
| OpenResFile function | New | Opens a file's resource fork into the current resource chain. |
| Palette.IsGlobal property | New | True if the specified palette is global, false if application-specific. Can now toggle palettes between global and app-specific under script control. |
| Palette.MainScreen property | New | Returns the number of the screen where the palette appears on multiple-monitor systems. |
| PopupFiles function | Changed | Now allows you to suppress the display of parent folders in the pop-up menu. |

| | | |
|---|---|---|
| PopupMenu function | Changed | Can now supply checked menu items either individually or in a list. |
| PopupMenuFont system variable | New | Gets or sets the font used in all OneClick pop-up menus. |
| PopupMenuSize system variable | New | Gets or sets the font size used in all OneClick pop-up menus. |
| PopupPalette command | Changed | Now optionally prevents the tearing off of pop-up palettes, forcing the palette to behave like a pop-up menu. |
| PrintText command | New | Prints text to the current printer using the specified format options. |
| Process.Window property | New | Can now access windows in applications other than the active application. |
| QuoteText function | New | Word-wraps text and puts “>” at the beginning of each line. |
| SetICPref command | New | Assigns a value to the specified preference in Internet Preferences. |
| SetResource command | New | Stores data in a resource of the specified type and ID in the specified file. |
| SetScrap command | New | Puts data on the Clipboard as the specified resource type. |
| SetStringList command | New | Stores the specified <return>-delimited list in a STR# resource in the specified file. |
| SysVersion system variable | New | Returns the Mac OS version as a whole number (750, 851, and so on.) |
| TextWidth function | New | Returns the pixel width of a line of text. |
| True system variable | New | Returns the number 1. |
| TruncText function | New | Truncates a line of text so that it fits within the specified pixel width. |
| UserHandler1 … UserHandler5 handlers | New | User-defined handlers you can use as subroutines in scripts. |
| Window.Collapsed property | New | Gets or sets the windowshade (collapsed) setting of a window. |
| Window.Zoom property | Changed | Now works with Appearance Manager, Kaleidoscope, and other software that relocates zoom boxes. |

# Absolute function

**Syntax** Absolute *number*

**Description** Returns the absolute value of *number*.

**Examples**

```
// Each statement types the value 25
Type Absolute –25
Type Absolute 25
```

## Alias function

**Syntax** Alias *path*

**Description** Returns an alias for the specified file or folder, formatted as a string. The alias string can be stored for later use and can locate the file it represents even if the file is moved or renamed. The alias string can be passed to any command, function, or object that expects a file path as a parameter, such as Open and the File object.

You can also use an alias string to send an alias in an Apple event to applications that require alias parameters. The function formats the alias string in the same format required by the Process.SendAE message.

**Example**

```
On DragAndDrop
    // Store an alias to the dropped file in a static variable
    Variable Static theAlias
    theAlias = Alias GetDragAndDrop 1
End DragAndDrop
On MouseUp
    Variable Static theAlias
    If OptionKey
        // Display the path to the alias' parent
        Message File(theAlias).Original
    Else If CommandKey
        // Display the alias string, just for fun
        Message theAlias
    Else
        // Open the file pointed to by the alias
        // Still works even if the original file is moved or renamed
        Open theAlias
    End If
End MouseUp
```

**See Also** Open command (page 253), File object (page 199)

## AppleScript command

**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.

AppleScript scripts that change any properties within the script get updated with the new properties only if the script is in a compiled script file. AppleScripts embedded within EasyScript scripts do not get their properties updated.

You must have the AppleScript software installed to use this command. AppleScript is included in Mac OS 7.5 and later.

For more information on integrating AppleScript with EasyScript, including how to share data between environments and how to embed EasyScript within AppleScript applets or compiled scripts, see Appendix B, "Integration with AppleScript".

**Examples**

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

// Open the Finder's "About This Macintosh" window.
AppleScript
    tell application "Finder"
        activate
        open about this computer
    end tell
End AppleScript
```

**See Also** Appendix B, "Integration with AppleScript"

# AskButton function

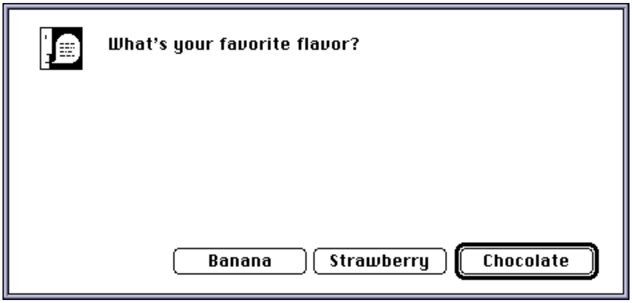
**Syntax** AskButton [*prompt*] [, *button-list*] | [, *button1* [, *button2*...]]

**Description** Displays a dialog box with the specified *prompt* message and lets you click one of up to eight buttons. If you don't specify any buttons, the dialog box has a single OK button. The AskButton function returns the number (1–8) of the button clicked.

Button names can either be passed in one parameter as a list, or as separate parameters, one button name per parameter.

**Examples**

```
Variable theAnswer
theAnswer = AskButton "Favorite flavor?", "Chocolate<return>Strawberry<return>Banana"
// Types 1, 2, or 3 depending on the button clicked (1=Chocolate, 2= Strawberry, 3=Banana)
Type theAnswer
```



What's your favorite flavor?

**See Also** Message command (page 244), Notify command (page 250), AskList function (page 154)

# AskFile function

**Syntax** AskFile [*type-list*] [, *prompt*]

**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.

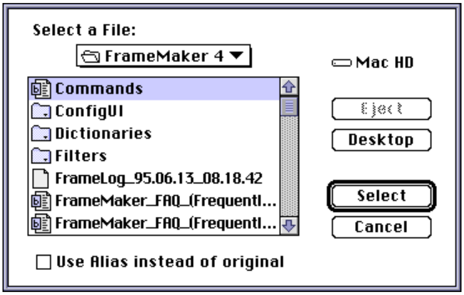
*Prompt* is an optional message to display in the dialog box. If *prompt* is omitted, the default message “Select a File/Folder:” appears.

To set the default directory that appears in the dialog box, set the Directory system variable to the desired path before calling AskFile.

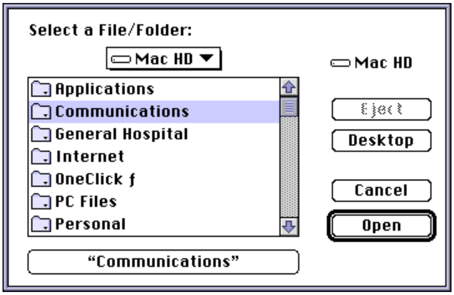
In the AskFile dialog box, check the “Use Alias instead of original” checkbox to return the path to an alias instead of the path to the file or folder the alias refers to.

**Examples**

```
Variable theFile, theFolder
theFolder = AskFile "fold"
theFile = AskFile "TEXT"
```


Select a File:
FrameMaker 4
Mac HD
Commands
ConfigUI
Dictionaries
Filters
FrameLog_95.06.13_08.18.42
FrameMaker_FAQ_(Frequentl...
FrameMaker_FAQ_(Frequentl...
Eject
Desktop
Select
Cancel
□ Use Alias instead of original

Select File dialog box (AskFile "TEXT")



Select Folder dialog box (AskFile "fold")

**See Also** AskNewFile function (page 155), Directory system variable (page 191)

## AskKey function

**Syntax** AskKey [*prompt*] [, *default-shortcut*] [, *cancel-value*]

**Description** Prompts you to press a key and returns the keystroke in text format. The format is the same as that expected by the Button.KeyShortCut property (see page 163). You can optionally provide a default shortcut, a brief prompt message, and a special value to return if the user clicks Cancel.

AskKey returns the null string ("") if you click None in the shortcut dialog box.

If you supply a value in *cancel-value*, a Cancel button appears in addition to the OK and None buttons and AskKey returns the *cancel-value* if you click Cancel.

**Example**

```
// Prompt the user to change the button’s shortcut, then display the new shortcut.
// Use the button's existing shortcut as the default shortcut.
Variable theKey
theKey = AskKey "Press a key combination:", Button.KeyShortCut, Char 0
If theKey = ""
    Message "You clicked None."
    Button.KeyShortCut = ""   // remove the shortcut, if any
Else If theKey = Char 0
    Message "You clicked Cancel."        // don’t change or remove the shortcut
Else
    Button.KeyShortCut = theKey  // change the shortcut to the key typed by the user
    Message "This button's shortcut is now " & theKey
End If
```

## AskList function

**Syntax** AskList *item-list* [, *prompt*] [, *selected-list*]

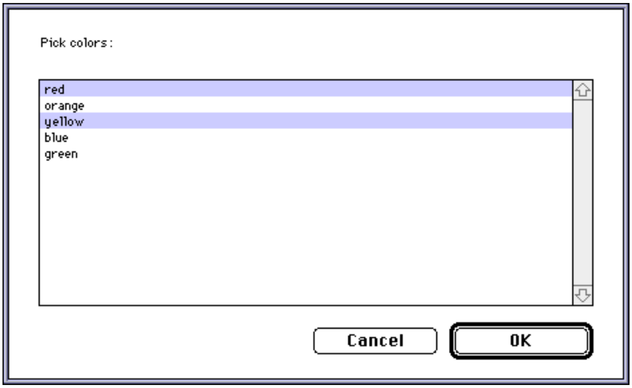
**Description** Displays a message (*prompt*) and a list of items (*item-list*) in a dialog box and returns a list containing the selected items. If you want one or more items in the list to appear selected by default, include the list of selected items in *selected-list*.

When the dialog box appears, select an item by clicking it, by using the arrow keys, or by typing its name (if the list is sorted). To select a single item and close the dialog box, double-click the item.

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 colors with red and yellow already selected
theResponse = AskList "red<return>orange<return>yellow", "Pick colors:", "red<return>yellow"
Type "You picked:" & Return & theResponse
```



Pick colors:

red
orange
yellow
blue
green

Sample AskList dialog box

**See Also** AskText function (page 156), AskButton function (page 152)

## AskNewFile function

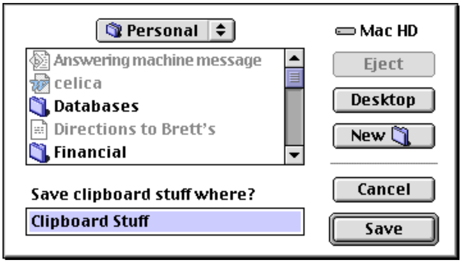
**Syntax** AskNewFile [*prompt*] [, *default-name*]

**Description** Displays a standard Save As dialog box for you to locate and name a file and returns the full path to the file.

AskNewFile does not actually create the file. Its purpose is to return a full path to a file that your script may create later.

**Examples**

```
// Display the Save As dialog box
Variable userPath
userPath = AskNewFile

// Display the Save As dialog box with a prompt and default name of "Clipboard Stuff"
Variable userPath
userPath = AskNewFile "Save clipboard stuff where?", "Clipboard Stuff"
// Add the Clipboard contents to the end of the file specified by the user
ConvertClip
File(userPath).Append = Clipboard & Return & Return
// Display the full path to the named file
Message "Your Clipboard data was added to " & Return & userPath
```



Save clipboard stuff where?

Clipboard Stuff

Sample AskNewFile dialog box

**See Also** [AskFile function (page 152)](), [Directory system variable (page 191)]()

## AskShortcut function

**Syntax** AskShortcut

**Description** Reserved for WestCode use. AskShortcut displays a dialog promting the user for a new shortcut's name, keystroke, visibility, and global status and returns the results in a list.

## AskText function

**Syntax** AskText [*prompt*] [, *default-value*] [, *cancel-value*]

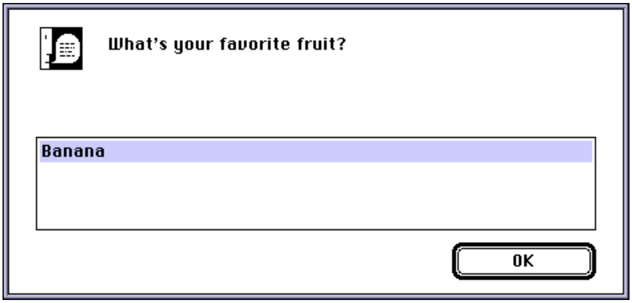
**Description** Displays a dialog box with an edit box and prompts you to type a line of text. When you click OK, the function returns the typed text. The optional *prompt* is the prompt in the dialog box. The second parameter, *default-value*, is an optional default string that appears pre-entered in the edit box.

If you supply a value in *cancel-value*, a Cancel button appears in addition to the OK button and AskText returns the *cancel-value* if you click Cancel.

**Examples**

```
// Assigns whatever is typed to Result. The dialog box says: Type something
Variable Result, favoriteFruit, theResponse
Result = AskText "Type something"

// The default response is Banana
favoriteFruit = AskText "What's your favorite fruit?", "Banana", Char 0
theResponse = "You typed " & Result & " and "
If favoriteFruit = Char 0
	Message theResponse & "you clicked Cancel."
```

```
Else
	Message theResponse & "your favorite fruit is " & favoriteFruit
End If
```


What's your favorite fruit?

Banana

Sample AskText dialog box

**See Also** Message command (page 244), AskButton function (page 152), AskList function (page 154), AskFile function (page 152)

## ASResult system variable

**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 322), AppleScript command (page 150)

## Beep command

**Syntax** Beep

**Description** Plays the system alert sound.

**Examples**

```
Beep
```

**See Also** BeepLevel system variable (page 158), Sound command (page 292), SoundLevel system variable (page 293)

## BeepLevel system variable

**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 System Alert Volume on the Alerts panel of the Monitors & Sound control panel.

The alert sound level can be zero (no sound) to 7 (highest sound level).

**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 system variable (page 293), Beep command (page 157), Sound command (page 292)

## Button object

**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.

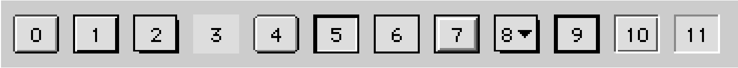
You can specify a button either by name or by number. Specifying by name lets you perform an operation on a specific button that you already know the name of. Specifying by number lets you loop through all the buttons on a palette and perform operations on each of them in sequence. Buttons are numbered 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:

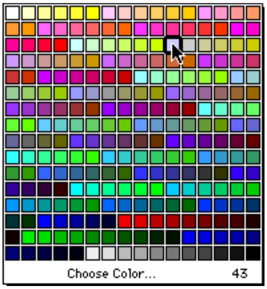
```
Palette(palette-specifier).Button(button-specifier).Property
```

**.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, or to create a transparent hotspot on a palette), 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 X
For X = 1 to Button.Count
      Button(X).Color = 43
End For
```

### .Data

Gets or sets the text stored in the button’s data property. The .Data property is similar to a static variable—OneClick stores the data in the palette file so it doesn’t get lost when you restart or shut down. Unlike statics, however, any button can get or set another button’s .Data property.

Like the .Text and .Help properties, .Data always returns its value as a text value, regardless of whether you stored a text or number value.

```
// Ask the user to choose a text file, then store the path to that file in the button’s .Data
If NOT Button.Data        // nothing stored yet
      Button.Data = AskFile "TEXT", "Choose a text file:"
Else        // Open the file
      Open Button.Data
End If

// Clear all stored .Data in every button on the palette
Variable X
For X = 1 to Button.Count
      Button(X).Data = ""
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 X, numButtons
numButtons = Palette("Cool Buttons").Button.Count
For X = 1 to numButtons
    Palette("Cool Buttons").Button(X).Delete
End For
```

**.Drag** Lets you drag a button anywhere on the palette to reposition the button without opening the OneClick Editor. The dragged button snaps to the palette's grid settings. Dragging to another palette copies the button to that palette.

The .Drag message works only in a MouseDown handler. When used as a property, .Drag returns the name of the palette where the button was dropped.

```
// Allow a button to be dragged to a new location if the Command key is held down
On MouseDown
    Variable newPalette
    If CommandKey
        newPalette = Button.Drag
    Else
        Beep
    End If
    Message "Button was dragged to palette " & newPalette
End MouseDown
```

**.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:

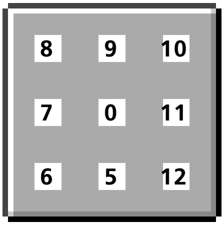
Button.Icon = *icon-number, path* [, *icon-size*]

*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.)



| | | |
|---|---|---|
| 8 | 9 | 10 |
| 7 | 0 | 11 |
| 6 | 5 | 12 |

### .Index

Returns the corresponding index number for the button when the button is specified by name (1 for the first button, 2 for the second, and so on). You can assign a numeric value to .Index to change the button order.

Changing Button.Index visibly changes the order in which overlapped buttons are drawn. A button with a higher index number draws on top of a button with a lower index number.

```
// Make this button the first button drawn on the palette
// Any overlapping buttons are drawn on top of this button
Button.Index = 1

// Make this button the last button drawn on the palette
// Any overlapping buttons are drawn behind this button
Button.Index = Button.Count
```

### .KeyShortCut

Gets or sets the button's keyboard shortcut. In the string that describes the keyboard shortcut, use the following keywords to describe modifier keys.

| Key | Keyword in shortcut string |
|---|---|
| Command | cmd- |
| Shift | shift- |
| Option | opt- |
| Control | ctl- |

| Key | Keyword in shortcut string |
|---|---|
| Numeric keypad | num- |

For example, the string “cmd-opt-1” assigns the shortcut Command-Option-1, and the string “num-7” assigns the keyboard shortcut to the 7 key on the numeric keypad. Set the shortcut to the empty string ("") to remove the shortcut.

```
// Set button shortcut to Command-Shift-Option-Control-C
Button.KeyShortCut = "cmd-shift-opt-ctl-c"

// Remove the shortcut from the button named “Olivia”
Button("Olivia").KeyShortCut = ""
```

**.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 X
For X = 1 to Button.Count
      Type Button(X).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 X
For X = 1 to Button.Count
      Button(X).Name = "Button" & MakeText X
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.

OneClick positions the new button in an empty space on the palette (the same as if you had clicked New Button in the Button Editor). However, the palette is not automatically resized to make room for the button.

You can create a new button with a width and height different from the default by including the optional width and height parameters.

Button(*button-name*).New *width*, *height*

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 for the original button 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")
```

**.Record** Gets or sets the current record mode for the specified button, allowing you to record a new script into a button. To stop, start, or pause recording, set Button.Record to 0, 1, or –1 according to the values in the following table.

| | Set values | Return values |
|---|---|---|
| 0 | Stop recording | Recording off |
| 1 | Start or resume recording | Recording on |
| –1 | Pause recording | Recording paused |

Any button specifiers are ignored except when starting recording.

```
// Start recording a new script into the button “StepSaver” on the current palette
Button("StepSaver").Record = 1

// Stop button recording
Button.Record = 0

// Beep if recording is on
If Button.Record = 1
	Beep
End If
```

**.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> tag 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>** |

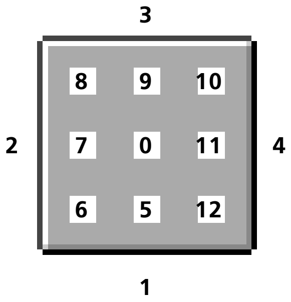
```
// Create three new buttons and assign scripts to them. The scripts assigned to each button
// are actually identical; only the method used to represent the Return character is different.
Button("New Button 1").New
Button("New Button 2").New
Button("New Button 3").New
Button("New Button 1").Script = "Message 'Hello there.'<return>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.

Setting .TextFont to 0 (zero) uses the active System font (the font known as “Large System Font” in the Appearance control panel, usually Charcoal or Chicago). Setting .TextFont to 1 uses the active Application font (usually Geneva).

```
// 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

// Set the button’s font to whatever the active System font is
Button.TextFont = 1
```

**.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: Condense | 64: Extend | |

```
// 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
```

## Call command

**Syntax**

```
Call [handler-name,] button-name [, palette-name]
Call handler-name
```

**Description** Runs another button's script (or handler within a 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*.

Specify a handler name as an argument to call a specific handler in any button (including the same button).

**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"

Call Startup             // Call Startup handler in current script
Call UserHandler3, "Select Font", "Text Buttons"// Call UserHandler3 handler in specified button
```

**See Also** Calling scripts as subroutines (page 128), Calling scripts as functions (page 128), UserHandler1 … UserHandler5 handlers (page 299)

## CallResult system variable

**Description** A variable you can use to save data between calls to different handlers or scripts. You can think of CallResult as similar to a global variable that's always available and that doesn't need to be declared. Because CallResult is actually a system variable (not a global variable), its value cannot be passed to an AppleScript script.

OneClick does not change the value of CallResult; setting or clearing the value is your script's responsibility.

**Examples**

```
// Button “Startup” on palette “Utilities”
Call "CheckVersion", "Extras"
If NOT CallResult
    Message "You need Mac OS 8.1 or later."
    Exit
End If
```

```
// Button "CheckVersion" on palette"Extras"
If (SysVersion < 810)
      CallResult = False
Else
      CallResult = True
End If
```

## Char function

**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 function (page 176)

## Click command

**Syntax** Click [Global | Local] [Command] [Option] [Control] [Shift] *X*, *Y* [, *toX*, *toY* [, Delay *delay-time*] ]

**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.

## Clicking in application tool bars

The Click command can click in application tool bars, such as those built in to Photoshop and ClarisWorks. (The term "tool bar" refers to any palette or other floating window built into the application—not a OneClick palette.) Use the following syntax to click in an application tool bar.

Click Local *tool-bar-name*, *X*, *Y* [, *toX*, *toY*]

You must specify the keyword Local and the name of the tool bar window followed by the click locations (you can also click and drag on a tool bar). If you don't know the name of the tool bar window, use the Window option in the Parameters pop-up menu in the Script Editor. If the tool bars don't have names, you can specify them by number (the same way you specify windows by number), but this method won't be reliable if the application allows the tool bars to change their front-to-back order.

Recording a script creates the appropriate Click command when you click on a tool bar.

## Specifying a delay time

The delay-time parameter allows you to specify in tenths of a second how long the mouse button is held down. In Canvas, for example, if you click on a tool which pops up a menu, it takes a while for the pop-up menu to draw. If the Click commands lets go of the mouse button too soon, no item is selected from the menu.

```
Click 10, 100, 40, 150, Delay 10  // Holds the mouse button down for one second
```

When recording a click and drag, a default delay of 10 (one second) is recorded in the script. If the script doesn't need the delay, you can edit the script and take out the delay.

**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

// Click and hold the mouse down for one second at 10, 100 then drag to 40, 150
Click 10, 100, 40, 150, Delay 10

// Click at 10, 60 in the tool bar or window named Styles
Click Local "Styles", 10, 60
```

**See Also** SelectButton command (page 287), SelectMenu command (page 288), SelectPopUp command (page 290)

# Clipboard system variable

**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, or if you assign new data to the Clipboard and find that pasting the Clipboard doesn't paste the correct data, you may need to use the ConvertClip command (see page 179).

**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 125), ConvertClip command (page 179)

## CloseResFile command

**Syntax** CloseResFile *refNum*

**Description** Closes a file’s resource fork and removes it from the current resource chain.

**Note** This is a technical Mac OS function and should be used at your own risk.

**Examples**

```
Variable refNum
// Open the resource fork of a file containing sound resources
// and put the file’s reference number in refNum.
refNum = OpenResFile "Mac HD:YoYoLand™:Sound Library"
// Play a sound contained in the resource file.
Sound "Mystery"
// Close the file referred to by refNum.
CloseResFile refNum
```

**Author Info** CloseResFile, part of Resource Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## CloseWindow command

**Syntax**

```
CloseWindow
```

**Description**

Closes the active (front) window.

**Examples**

```
CloseWindow
```

**See Also**

Window object (page 305)

## Code function

**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 function (page 172)

## ColorPicker function

**Syntax**

ColorPicker [*prompt*] [, *defaultColor*]

**Description**

Displays the Color Picker dialog box and returns a return-delimited list of the three RGB (red, green, blue) values of the selected color. RGB values are in the range 0–65535.

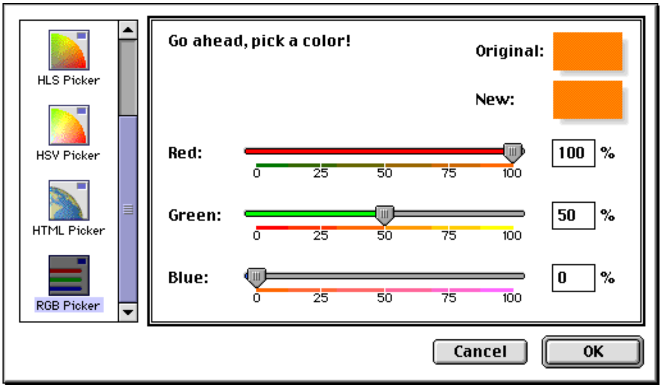
*Prompt* is an optional message to display in the dialog box. *DefaultColor* is a return-delimited list of RGB values to display as the original color.

Clicking Cancel in the dialog box returns the empty string (““).

**Examples**

```
// Show a custom prompt and start with orange (full red, half green, no blue)
Variable theColor
theColor = ColorPicker "Go ahead, pick a color!", "65535<return>32767<return>0"
```

```
// Display the color values chosen
Message theColor
```


Go ahead, pick a color!

Red: 100 %

Green: 50 %

Blue: 0 %

Sample Color Picker dialog box

**Author Info** ColorPicker Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## CommandKey system variable

**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 system variable (page 179), OptionKey system variable (page 255), ShiftKey system variable (page 292)

# ContextualMenu handler

**Description** A script's ContextualMenu handler executes when you Control-click a button, overriding OneClick's built-in contextual menu. Like the MouseDown handler, a ContextualMenu handler executes immediately when you click, before you release the mouse button.

If a button named PaletteContextualMenu exists on the palette, OneClick runs the ContextualMenu handler in the PaletteContextualMenu button whenever you Control-click an empty area of the palette's background, overriding the built-in palette contextual menu. If the PaletteContextualMenu button exists but does not contain a ContextualMenu handler, then nothing happens (and no contextual menu appears) when you Control-click the palette's background.

**Note** If the ContextualMenu handler in the PaletteContextualMenu button displays a pop-up menu, the menu appears on the button, not where you Control-clicked the palette. Because of this behavior, it's best to use a PaletteContextualMenu button only when you want to disable OneClick's contextual menu for your palette, rather than provide an alternate menu.

**Examples**

```
// When you Control-click the button, you see a pop-up menu of available sounds.
// To play the sound chosen from the menu, click the button without the Control key down.
On ContextualMenu
    Variable Static chosenSound // Static so the name won't be lost when the script ends
    Variable theChoice
    // Pop up a menu of sounds
    theChoice = PopupMenu (GetResources "snd ")
    If theChoice
        chosenSound = theChoice
    End If
End ContextualMenu
On MouseDown
    Variable Static chosenSound
    Sound chosenSound
End MouseDown

// When you Control-click the palette's background, this handler in a button named
// PaletteContextualMenu runs, overriding OneClick's contextual menu. The handler behaves
// as if you had Control-clicked the PaletteContextualMenu button itself.
On ContextualMenu
```

```
        Beep
        Beep
End ContextualMenu
```

**See Also** MouseUp handler (page 249), IsMouseDown system variable (page 230)

## ControlKey system variable

**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 system variable (page 177), OptionKey system variable (page 255), ShiftKey system variable (page 292)

## ConvertClip command

**Syntax** ConvertClip [*copy-to-application*]

**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.

If the *copy-to-application* parameter is 1 (True), OneClick tells the application that your script has put new data on the Clipboard and that the application needs to recognize the new data. If the parameter is 0 (False) or missing, it indicates that you want to get the data from the Clipboard and the application needs to make its Clipboard data publicly available.

**Examples**

```
// Get the contents of the Clipboard
ConvertClip
a = Clipboard

// Put data on the Clipboard
Clipboard = a
ConvertClip 1
```

**See Also** Clipboard system variable (page 174)

## Cursor system variable

**Description** Returns the ID number of the cursor (mouse pointer) or changes the cursor.

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.

If you change the cursor by setting the Cursor variable, the cursor reverts to the previous cursor when the script ends or when you set Cursor to 0 (zero). To see available cursors and insert a cursor ID number in a script, choose Cursor from the

Parameters pop-up menu in the Script Editor, then choose one of the cursors displayed.

**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"

// In Finder, change the cursor to a crossbar, then display a message
Cursor = 3
Message "The cursor should be a crossbar"
// The cursor changes back when the script ends
```

## DateTime object

**Description**

The DateTime object lets you work with dates and times. Using the properties of the DateTime object, you can get the current date and time, format the date or time in a variety of string formats, and perform date math.

The specifier for the DateTime object is a *serial number*, which is the number of seconds since January 1, 1904, 00:00:00 (midnight). You use .DateSerial to convert a year, month, and day to a serial number; you use .TimeSerial to convert hours, minutes, and seconds (since midnight of the current date) to a serial number. To get a serial number that represents both a date and time, add the .DateSerial and .TimeSerial values.

You can perform date math (such as adding a number of days to a date) by first converting a date to a serial number, then adding or subtracting the appropriate number of seconds (86400 seconds per day). You can then use .DateString or other date-related properties to convert the new serial number to a date.

The .DateString and .TimeString properties return the specified date or time formatted as a string. An optional parameter lets you adjust how the date or time should be formatted.

The .Year, .Month, .Day, .Weekday, .Hour, .Minute, and .Second properties return their respective part of the specified serial number. If no serial number is specified, the properties assume the current date and time.

The DateTime object supports dates and times from January 1, 1904 00:00:00 AM through February 6, 2040 6:28:15 AM. Because dates are expressed as a number of seconds since 1904, dates larger than EasyScript's maximum number (2,147,483,647—January 19, 1972 3:14:07 AM) "wrap around" into the negative number range and continue counting up from –2,147,483,648.

All DateTime properties are read-only.

### .DateSerial

Returns the current date as a serial number. The number of seconds since 00:00:00 (midnight) is not included in the serial number.

To get a serial number for a different date, include the year, month, and day parameters following .DateSerial. The year must be four digits; 99 is not interpreted as 1999.

```
// Determine the number of days between today and July 13, 1963
Variable currentDate, birthDate, numSeconds, numDays
// Get the serial number for today's date
currentDate = DateTime.DateSerial
// Get the serial number for July 13, 1963
birthDate = DateTime.DateSerial 1963, 7, 13  // year, month, day
numSeconds = currentDate - birthDate
numDays = numSeconds / 86400    // 86400 seconds per day (24 * 60 * 60)
Message "I was born " & numDays & " days ago."
```

### .DateString

Returns the specified date as a string, or the current date if no serial number is specified. The optional format parameter specifies which of several date formats to use. If you don't specify a format, DateString 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 |

| Format | Type | Example |
|---|---|---|
| 4 | Short DMY | 22/4/94 |
| 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 |

```
Variable theDate
theDate = DateTime.DateSerial 1995, 9, 26
```

| | |
|---|---|
| `Type DateTime(theDate).DateString` | Types: 9/26/95 |
| `Type DateTime(theDate).DateString 0` | Same as above |
| `Type DateTime(theDate).DateString 1` | Types: September 26, 1995 |
| `Type DateTime(theDate).DateString 3` | Types: 9/26/95 |
| `Type DateTime(theDate).DateString 19` | Types: 09/26/95 |
| `Type DateTime(theDate).DateString 179` | Types: 09.26.1995 |
| `Type DateTime(theDate).DateString 117` | Types: 1995-09-26 |
| `Type DateTime(theDate).DateString 8` | Types: September 2, 1995 |

```
Type DateTime(theDate).DateString 26              Types: Tue, Sep 26, 1995
```

**.Day** Returns the day of the month (1–31) for the specified serial number, or the current day if no serial number is specified.

**.Hour** Returns the number of hours (0–23) since midnight for the specified serial number, or the current hour if no serial number is specified.

**.Minute** Returns the number of minutes (0–59) past the hour for the specified serial number, or the current minute if no serial number is specified.

**.Month** Returns the month number (1–12) for the specified serial number, or the current month if no serial number is specified.

```
// Get the current month number and display it as a string
Variable theMonth monthList
theMonth = DateTime.Month
monthList = "January,February,March,April,May,June,July,August,"
monthList = monthList & "September,October,November,December"
ListDelimiter = ","
Message "It’s " & ListItems monthList, theMonth
```

**.Second** Returns the number of seconds (0–59) past the minute for the specified serial number, or the current second if no serial number is specified.

**.TimeSerial** Returns the current time as a serial number. The serial number includes only seconds since 00:00:00 (midnight) of the current date; the number of seconds since January 1, 1904 is not included in the serial number.

To get a serial number for a different time, include the hour (0–23), minute (0–59), and second (0–59) parameters following .TimeSerial.

```
// Determine the time two and a half hours (150 minutes) from now
Variable currentTime, newTime
// Get the serial number for the current time
currentTime = DateTime.TimeSerial
newTime = currentTime + (150 * 60)      // 150 minutes times 60 seconds per minute
Message "2.5 hours from now, it will be " & DateTime(newTime).TimeString

// Determine the amount of time it takes a script to run in hours, minutes, and seconds.
// (This calculation works only if the script takes less than 24 hours to run.)
Variable startTime elapsedTime
// Get the start time in seconds since 1-Jan-1904 00:00:00.
// Important: Because .DateSerial accepts parameters, you need to enclose it in parentheses
// so that the expression following it is not interpreted as a parameter.
startTime = (DateTime.DateSerial) + DateTime.TimeSerial
// Do a lengthy operation here. For this example, the “lengthy operation” simply waits
// for you to press a key. After running the script, wait a few seconds or minutes
// and then press a key to see the time elapsed.
Wait IsKeyDown
// Subtract the start time from the end time to get the duration in seconds.
elapsedTime = (DateTime.DateSerial) + (DateTime.TimeSerial) - startTime
// Display the elapsed time in HH:MM:SS format
Message "Elapsed time: " & DateTime(elapsedTime).TimeString 21
```

### .TimeString

Returns the specified time as a string, or the current time if no serial number is specified. The optional format parameter specifies which of several time formats to use. If you don’t specify a format, .TimeString 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 |
| 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 |

```
Variable theTime
theTime = DateTime.TimeSerial 16, 9, 13        // 4:09:13 PM
Type DateTime(theTime).TimeString               Types: 4:09 PM (format may vary)
Type DateTime(theTime).TimeString 19            Types: 04:09:13 PM
Type DateTime(theTime).TimeString 5             Types: 16:09:13
```

**.Weekday** Returns a number for the day of the week (1=Sunday, 2=Monday, … 7=Saturday) for the specified serial number, or the current day of week if no serial number is specified.

```
Variable theDate newDate dayList msg
dayList = "Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday"
theDate = DateTime.DateSerial
newDate = theDate + (86400 * 4)  // add four days
ListDelimiter = ","
msg = "Today is " & (ListItems dayList, DateTime(theDate).Weekday) & Return
msg = msg & "Four days from now it will be " & (ListItems dayList, DateTime(newDate).Weekday)
Message msg
```

**.Year** Returns the year (1904–2040) for the specified serial number, or the current year if no serial number is specified.

## Dial command

**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
```

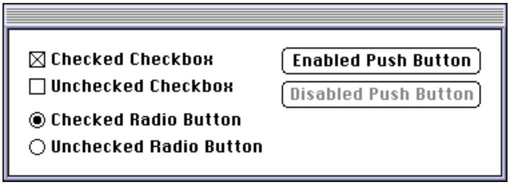
```
Dial "555-7427"

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

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

## DialogButton object

**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.

You can specify dialog box buttons by number. Specifying by number is necessary when buttons have no names or duplicate names. To determine the number of a dialog box button, use the Button item in the Parameters pop-up menu in the Script Editor. The first button listed is 1, the second is 2, and so on.



☒ Checked Checkbox
□ Unchecked Checkbox
● Checked Radio Button
○ Unchecked Radio Button

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.

**.Count** Returns the number of buttons and checkboxes in the active dialog box.

**.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
```

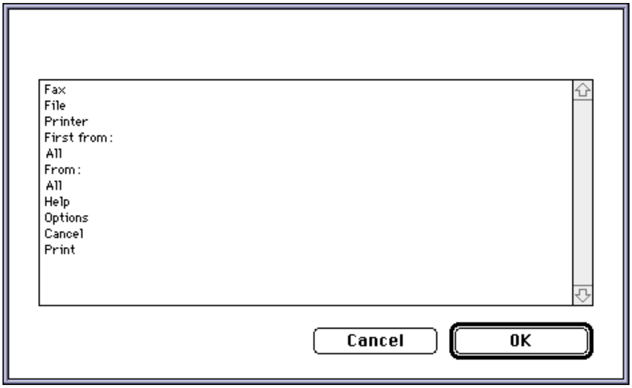
### .Index

Returns the corresponding index number for the dialog box button when the dialog box button is specified by name (1 for the first dialog box button, 2 for the second, and so on). For the DialogButton object, this property is read-only.

### .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
```



Fax
File
Printer
First from:
All
From:
All
Help
Options
Cancel
Print

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<return>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
```

### .Name

Returns the name of a dialog box button specified by index number.

```
Message DialogButton(1).Name      // Displays the first button's name
Message DialogButton(DialogButton.Count).Name // Displays the last button's name
```

## Dialogs system variable

**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
```

## DialogText system variable

**Description** The DialogText system variable lets you get or set the text of the active text box in a dialog box. (The active text box is the box containing the insertion point or highlight.) Setting the DialogText variable is faster than using the Type command to put text in a text box.

In dialog boxes that contain more than one text box, type a Tab character to move between boxes.

**Examples**

```
// Type “Cesar” in the current text box
DialogText = "Cesar"
// Move to the next text box, then type “Faison”
Type Tab
DialogText = "Faison"

// Add the suffix .PICT to the file name in the Save As dialog box
SelectMenu "File", "Save As*"
DialogText = DialogText & ".PICT"

// Copy the text in the active text box to the third text box
Variable theText
theText = DialogText
Type Tab Tab
DialogText = theText
```

## Directory system variable

**Description**

Returns the directory where file open and save operations start from, or sets the current directory for file open and save operations to the full path specified.

Set the Directory system variable to the path of the folder you want to appear in Open and Save As dialog boxes. If you set the Directory variable while an Open or Save dialog box is on the screen, the dialog box switches to the specified folder.

**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

// Change the folder in the Open or Save dialog box to the folder chosen from a pop-up menu
Directory = PopupFiles 0, "fold" // start at the desktop level, list only folders in menu
```

## DragAndDrop handler

**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 function (page 221), DragButton command (page 192), Using Drag and Drop (page 131)

## DragButton command

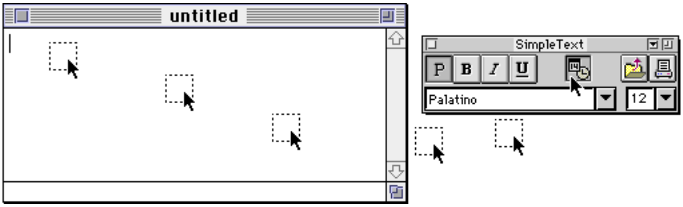
**Syntax** DragButton *text*

**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.

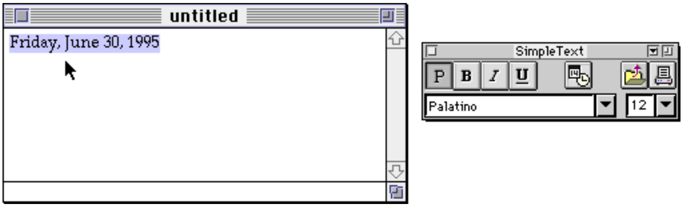
**Examples**

```
// 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
```

# DrawButton handler

**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 command (page 194), Button.Update (page 170), Palette.Update (page 262)

## DrawIndicator command

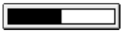
**Syntax** DrawIndicator *progress* [, *color*]

**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.

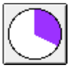
The indicator is proportional in size to the button and fills almost the entire button. To make an indicator appear directly on the palette (not inside a button), set the button's border style to None, set its appearance to Disabled, and uncheck its color checkbox.

**Examples**

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



```
// Draw a purple pie graph that's 1/3 full
DrawIndicator 33, –97
```



**See Also** DrawButton handler (page 193)

## Editor command

**Syntax** Editor *tab-name* [, *palette-name*] [, *button-name*]

**Description** Opens the OneClick Editor and displays the specified tab. If you specify an optional *palette-name* and *button-name*, the editor selects the palette or button.

**Examples**

```
// Open the Button Library
Editor "Library"

// Open the Script Editor and select the “System Startup” button on the palette named “Stuff”
Editor "Script", "Stuff", "System Startup"

// Open the editor chosen from a pop-up menu
Editor PopupMenu "Script<return>Button<return>Palette<return>Icon<return>Icon Search"
```

## EditorFont system variable

**Description** Gets or sets the font used in the OneClick Editor’s script pane. You can specify the font by its name or ID number. The font resets to the default font (Geneva) after the computer starts up.

**Example**

```
// Set the font used by the Script Editor to Monaco 12-point
EditorFont = "Monaco"
EditorSize = 12
```

**See Also** EditorSize system variable (page 195), PopupMenuFont system variable (page 267), PopupMenuSize system variable (page 267)

## EditorSize system variable

**Description** Gets or sets the font size (in points) used in the OneClick Editor’s script pane. The size resets to the default size (9-point) after the computer starts up.

**Example**

```
// Set the font used by the Script Editor to Monaco 12-point
EditorFont = "Monaco"
EditorSize = 12
```

**See Also** EditorFont system variable (page 195), PopupMenuFont system variable (page 267), PopupMenuSize system variable (page 267)

# Error system variable

**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 |

### Error 2: Not Found error

| | |
|---|---|
| Palette(*palette-name*) | Palette not found |
| Process(*process-name*) | Application not open |
| PopupPalette | Palette not found |
| Scroll | Window not found |
| SelectButton | Button not found |
| 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 |

**Error 3: Parameter error (generally means missing parameter)**

| | |
|---|---|
| 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 139), Checking for run-time errors (page 142)

# Exit command

**Syntax** `Exit [For | Repeat | While]`

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

When used with For, Repeat or While, Exit exits the loop and jumps to the statement immediately following the End For, End Repeat, or End While statement.

**Examples**

```
// Display the numbers 1 to 3, then display a message indicating the loop has finished
Variable X
For X = 1 To 50
```

```
        Message X
        If X = 3
                Exit For    // prematurely exit the loop after 3 iterations instead of 50
        End If
End For
Message "Loop is done!"
```

**See Also** Call command (page 171)

## False system variable

**Description** Returns the number 0 (zero).

**See Also** True system variable (page 297)

## File object

**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.

**.Append** Appends text to the end of an existing text file. See also File.Text (page 205).

```
// Append the contents of Text File 1 to the end of Text File 2
Variable fileText
fileText = File("Mac HD:Text File 1").Text
File("Mac HD:TextFile 2").Append = fileText
```

**.Busy** Returns a True (1) or False (0) value indicating whether or not the specified file is in use. A file is considered in use when it is open for reading or writing. For example, when Finder duplicates a file, the .Busy property of both the original file and the copy returns True (1) until the copy is completed.

Use .Busy when you need to wait for an application to finish writing to a file and close the file before your script does any processing with the file.

```
// Tell the application “Server App” to quit
```

```
Process("Server App").Quit
// Server App takes a while to quit. Our script will continue running before the app has actually
// finished closing its log file and quitting. We need to wait until Server App has has finished
// writing to its log file and has closed it before we do anything with it.
Wait NOT File("Mac HD:Server App:Server Log").Busy
// Server App is no longer using its log, so it's now safe to open it and get the text of the log.
Variable theText
theText = File("Mac HD:Server App:Server Log").Text
```

**.Count** Returns the number of items in the specified folder. Both visible and invisible items are counted.

```
// Display the number of files in the Control Panels folder
Message File(FindFolder "ctrl").Count
```

**.CreationDate** Returns the creation date and time of the specified file as a serial number (the number of seconds elapsed since midnight, January 1, 1904). Use .CreationDate with DateTime.DateString or DateTime.TimeString to convert the number of seconds to a human-readable format.

To get the date and time a file was last modified, use the .ModificationDate property.

```
// Display the the date and time that "My Document" was created.
Variable crDate
crDate = File("Mac HD:My Document").CreationDate
Message "Created on " & (DateTime(crDate).DateString) & " " & (DateTime(crDate).TimeString)
```

**.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.

```
// Change the creator code of all TEXT and PICT files in Mac HD:Data to "ttxt" (for SimpleText).
Variable X, theFile, theFolder, fileList
theFolder = "Mac HD:Data:"
fileList = File(theFolder, "TEXT<return>PICT").List// get list of TEXT and PICT files in Mac HD:Data
For X = 1 to ListCount fileList
    theFile = theFolder & (ListItems fileList, X) // theFile should be a full path
    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 X, theFile, fileList
    fileList = GetDragAndDrop
    For X = 1 to ListCount fileList
        theFile = ListItems fileList, X
        If File(theFile).Kind = "TEXT"
            File(theFile).Creator = "R*ch"
        End If
    End For
End DragAndDrop
```

### .Delete

The Delete message deletes the specified file or folder. Folders can be deleted only if they are empty.

**Caution** The file is deleted immediately without going to the Trash first. Use carefully.

### .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
```

### .FileVersion

Returns the version number of the specified file. If the file doesn’t contain a version resource, .FileVersion returns the empty string ("").

```
// Display the version of QuickTime™ in use.
Variable thePath
thePath = (FindFolder "extn") & "QuickTime™"        // QuickTime in the Extensions folder
Message "You’re using QuickTime™ version " & File(thePath).FileVersion
```

**.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, theFolder, theFileList, X
theFolder = "Mac HD:Downloads:"
theFileList = File(theFolder).List
For X = 1 to ListCount theFileList
    theFile = theFolder & (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.

**.KindString** Returns a file's type as it appears in the Kind column of Finder windows.

```
// Displays "SimpleText text document"
Variable theFile
theFile = "Mac HD:Desktop Folder:Test File"
File(theFile).Text = "Hello, world!"
Message File(theFile).KindString

// Unmounts all shared disks (file server volumes)
Variable X volList theVol
volList = Volume.List
For X = 1 To ListCount volList
    theVol = ListItems volList, X
    If File(theVol).KindString = "shared disk"
        Volume(theVol).Unmount
    End If
End For
```

**.List** Returns a list of files in the specified folder. A second file type specifier is optional: specify one or more 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.

```
// Display a list of all the files in the folder Mac HD:Data.
Message File("Mac HD:Data:").List

// Display a list of the TEXT, PICT, and JPEG files in the folder Mac HD:Data.
Message File("Mac HD:Data:", "TEXTPICTJPEG").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
```

**.Locked** Gets or sets the specified file's locked status (the same as the Locked checkbox in the file's Get Info window). Set to 1 to lock or zero to unlock.

```
// Lock a file
File("Mac HD:File A").Locked = 1

// Unlock a file
File("Mac HD:File A").Locked = 0
```

**.ModificationDate**

Returns the modification date and time of the specified file as a serial number (the number of seconds elapsed since midnight, January 1, 1904). Use .CreationDate with DateTime.DateString or DateTime.TimeString to convert the number of seconds to a human-readable format.

To get the date and time a file was created, use the .CreationDate property.

```
// Display the the date and time that "My Document" was last modified
```

```
Variable mdDate
mdDate = File("Mac HD:My Document").ModificationDate
Message "Modified " & (DateTime(mdDate).DateString) & " " & (DateTime(mdDate).TimeString)

// Compare the modification times of two dropped files and display the difference in seconds.
On DragAndDrop
    Variable file1, file2, md1, md2
    file1 = GetDragAndDrop 1
    file2 = GetDragAndDrop 2
    If NOT (file1 AND file2)
        Message "Drop two files on this button to compare their modification dates."
        Exit
    End If
    md1 = File(file1).ModificationDate
    md2 = File(file2).ModificationDate
    If md1 > md2
        Message file2 & " is " & (md1 - md2) & " seconds older than " & file1
    Else
        Message file1 & " is " & (md2 - md1) & " seconds older than " & file2
    End If
End DragAndDrop
```

### .Name

Sets the name of (renames) the specified file, or gets the name of the file at the specified path.

```
// Change the name of file “Test File 1” to “Backup File”
File("Mac HD:Data:Test File 1").Name = "Backup File"

// Display just the name (not the full path) of the dropped file or folder
On DragAndDrop
    Message File(GetDragAndDrop).Name
End DragAndDrop

// Add the extension ".jpg" to all the JPEG files in the dropped folder
Variable theDroppedFolder, fileList, theFile, X
// Get the full path of the dropped folder
theDroppedFolder = GetDragAndDrop 1
// Get a list of JPEG files in the folder, ignoring other file types
fileList = File(theDroppedFolder, "JPEG").List
// Loop through all JPEG files in the list
For X = 1 To ListCount fileList
    theFile = ListItems fileList, X // Get a single filename from the list
    File(theDroppedFolder & theFile).Name = theFile & ".jpg" // Rename the file
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
```

**.Original** Returns the path to the original file of an alias. If you specify a file that isn't an alias, the path of the specifier is returned. If the alias is broken (cannot be resolved), .Original returns the empty string ("").

```
// When an alias is dropped on the button, display the path to the original file
On DragAndDrop
	Message File(GetDragAndDrop).Original
End DragAndDrop
```

**.Size** Returns the size in bytes of the specified file.

```
// When a file is dropped on the button, display the size of the file
On DragAndDrop
	Message File(GetDragAndDrop).Size
End DragAndDrop
```

**.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
Variable theCurrentFile, X
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 X = 1 to ListCount theTotalFileList
    theCurrentFile = theDirectory & (ListItems theTotalFileList, X)
    If Find theSearchString, File(theCurrentFile).Text
        theFoundFileList = theFoundFileList & theCurrentFile & "<return>"
    End If
End For
theResponse = AskList theFoundFileList, "Search results:"
```

**.Visible** Gets or sets the specified file or folder's visibility. Set to 1 to make the specified file or folder visible or set to zero to make it invisible. Invisible files and folders do not appear in Finder windows or in most Open and Save dialog boxes.

**Caution** Once you make an item invisible, you won't be able to access it except in applications that let you see invisible items. Don't make the System folder or any of its contents invisible or unpredictable results may occur.

```
// Make the file “File A” invisible
File("Mac HD:File A").Visible = 0

// Make an invisible file visible
File("Mac HD:File A").Visible = 1
```

## FileClose command

**Syntax** FileClose *refNum*

**Description** Closes a file opened with FileOpen.

**Examples**

```
Variable theFile, theData, refNum
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "This is some text in the FileIO Demo file."
refNum = FileOpen theFile
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

**Author Info** FileClose, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileGetEOF function

**Syntax** FileGetEOF *refNum*

**Description** Returns the end-of-file (length) of a file opened with FileOpen.

**Examples**

```
Variable theFile, refNum, theLength
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "This is some text in the FileIO Demo file."
refNum = FileOpen theFile
theLength = FileGetEOF refNum
Message "The file contains " & theLength & " bytes of data."
FileClose refNum
```

**Author Info** FileGetEOF, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileGetPos function

### Syntax

FileSetPos *refNum*, *mark*

### Description

Gets the mark (current position) in a file opened with FileOpen. Future FileRead and FileWrite operations always begin at the file's current mark. When FileOpen opens a file, the file's mark is set to 0 (zero). After reading or writing data in the file, the mark is set following the last character read or written.

### Examples

```
Variable theFile, refNum, theData, mark
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "First line" & Return & "Second line"
refNum = FileOpen theFile
theData = FileRead refNum, "until", Return      // Reads data until the first carriage return
mark = FileGetPos refNum      // Gets the current position in the file
FileClose refNum
Message "The first carriage return occurs at byte " & mark
```

### Author Info

FileGetPos, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileOpen function

### Syntax

FileOpen *path-to-file*

### Description

Opens a file's data fork for future reading or writing and returns a reference number for the opened file. All other FileIO commands and functions use the file's reference number to access the open file. The file remains open until it is closed using FileClose.

### Examples

```
Variable theFile, theData, refNum
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "This is some text in the FileIO Demo file."
refNum = FileOpen theFile
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

### Author Info

FileOpen, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

# FileRead function

**Syntax**

```
FileRead refNum, "all"
FileRead refNum, "line"
FileRead refNum, "until", character
FileRead refNum, "bytes", numberOfBytes
```

**Description** Reads data from the current mark (position) in a file opened with FileOpen and returns the data read from the file. After reading data, the file's mark is set to the byte following the last byte read.

FileRead can read data in four different modes. The first parameter is a string that indicates which mode to use.

| Mode string | Meaning |
|---|---|
| all | Reads all data from the current mark to the end of the file. After reading, the mark is set to one byte past the end of the file. |
| line | Reads data from the current mark up to (and including) the next carriage return character. After reading, the mark is set to the beginning of the next line. |
| until | Reads data from the current mark up to (and including) the specified character. After reading, the mark is set to the byte following the last byte read. |
| bytes | Reads the specified number of bytes beginning at the current mark. After reading, the mark is set to the byte following the last byte read. |

**Examples**

```
Variable theFile, theData, refNum
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "First line" & Return & "Second line"

// Read the file until the first "r"
refNum = FileOpen theFile
theData = FileRead refNum, "until", "r"
FileClose refNum
Message theData

// Read the whole file
refNum = FileOpen theFile
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

```
// Read the whole file from the current mark (4)
refNum = FileOpen theFile
FileSetPos refNum, 4
theData = FileRead refNum, "all"
FileClose refNum
Message theData

// Read a single line
refNum = FileOpen theFile
theData = FileRead refNum "line"
Message theData

// Read 5 bytes. The file is still open, so the reading begins following the first line.
theData = FileRead refNum, "bytes", 5
FileClose refNum
Message theData
```

**Author Info** FileRead, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileSetEOF command

**Syntax** FileSetEOF *refNum*

**Description** Sets the end-of-file (length) of a file opened with FileOpen.

The SetEOF function sets the end-of-file (length in bytes) of the specified file.

If you set the new end-of-file to a number less than the current end-of-file, the file is truncated. Setting the end-of-file to 0 (zero) deletes all data in the file.

If you set the new end-of-file beyond the current end-of-file, the file size increases on the volume to accomodate the new size. The newly-allocated space at the end of the file contains garbage data (whatever data happened to be on the volume where the new space was allocated).

**Note** If not used carefully, FileSetEOF can potentially delete data in the specified file. Don't use FileSetEOF unless you are confident with reading and writing directly to files.

**Examples**

```
Variable theFile, theData, refNum
```

```
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "This is some text in the FileIO Demo file."
refNum = FileOpen theFile
FileSetEOF refNum, 17     // Set the length of the file to 17, truncating the rest of the file
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

**Author Info** FileSetEOF, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileSetPos command

**Syntax** FileSetPos *refNum, mark*

**Description** Sets the current mark (position) in a file opened using FileOpen. Use FileSetPos to move the mark forward or backward in an open file; future FileRead and FileWrite operations always begin at the file's current mark. To set the mark to the beginning of the file, set the mark to 0 (zero). To set the mark to the byte following the last byte in the file, set the mark to the file's length (obtainable using FileGetEOF).

**Examples**

```
Variable theFile, refNum, theData
theFile = (FindFolder "desk") & "FileIO Demo"
File(theFile).Text = "First line" & Return & "Second line"
refNum = FileOpen theFile
FileSetPos refNum, 6       // Sets the file's mark to the byte following the 6th byte
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

**Author Info** FileSetPos, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## FileWrite command

**Syntax** FileWrite *refNum*, *data*

**Description** Writes data to a file opened with FileOpen, beginning at the current mark (position). After writing data, the file's mark is set to the byte following the last byte written.

**Note** If not used carefully, FileWrite can potentially overwrite data in the specified file. Don't use FileWrite unless you are confident with reading and writing directly to files.

**Examples**

```
Variable theFile, theData, refNum
theFile = (FindFolder "desk") & "FileIO Demo"
theData = "First line" & Return & "Second line"

refNum = FileOpen theFile
FileWrite refNum, theData // write the data to the file
FileClose refNum

refNum = FileOpen theFile
FileSetPos refNum, 11 // set the mark (position) for the next read/write
FileWrite refNum, "••••••" // Write text starting at the 12th byte (following the mark at 11)
FileSetPos refNum, 0 // reset the mark to the beginning of the file
theData = FileRead refNum, "all"
FileClose refNum
Message theData
```

**Author Info** FileWrite, part of FileIO Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

# Find function

**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).

Find ignores case when searching for matching text.

**Examples**

```
// Types 5
Type Find "is", "Now is the time"

// Types 0
Type Find "and", "Now is the time"
```

**See Also** Replace function (page 278)

## FindApp function

**Syntax** FindApp *creator-code*

**Description** Returns the full path to the application with the specified creator code. The FindApp function is helpful when you want to open a certain application but you don't know where it's located.

FindApp will find the application on any mounted volume. In addition to applications, FindApp also finds other executable files including desk accessories, control panels, Control Strip modules, and so on.

**Note** An easy way to find a file's creator code is to run Find File, choose "creator" from the pop-up menu, and drag the file from the Finder to the text box area. You can also use ResEdit to find a file's creator.

**Examples**

```
// Open Find File
Open FindApp ("fndf")

// With BBEdit, open the “Read Me” file located on the desktop
Open FindFolder ("desk") & "Read Me", FindApp ("R*ch")
```

## FinderAlias command

**Syntax** FinderAlias [*path-list*,] *destination* [, *wait-for-completion*]

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

*Path-list* may contain one or more files, folders, or a combination of files and folders.

If the destination folder is the same as the source folder, then the Finder simply makes an alias of the item and adds "alias" to the end of the alias' name.

In Mac OS 8 and later, the Finder's alias operation will begin and run in parallel with your script, allowing your script to continue running while the Finder makes aliases. If you include 1 (True) in the optional *wait-for-completion* parameter, OneClick waits until the Finder has completed the operation before resuming execution of the script. Always include 1 in the *wait-for-completion* parameter if your script expects the alias to exist after the FinderAlias statement.

**Examples**

```
// Make aliases of the selected Finder icons in the Apple Menu Items folder
FinderAlias "Mac HD:System Folder:Apple Menu Items:"

// Make an alias of the file "Status Report" (on the desktop) in the folder "Current Work"
FinderAlias "Mac HD:Desktop Folder:Status Report", "Mac HD:Current Work:"
```

**See Also** Alias function (page 150), FinderCopy command (page 214)

## FinderCopy command

**Syntax** FinderCopy [*path-list*,] *destination* [, *wait-for-completion*]

**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.

In Mac OS 8 and later, the Finder’s copy operation will begin and run in parallel with your script, allowing your script to continue running while the Finder copies files. If you include 1 (True) in the optional *wait-for-completion* parameter, OneClick waits until the Finder has completed the copy before resuming execution of the script. Always include 1 in the *wait-for-completion* parameter if your script expects the copied file to exist after the FinderCopy statement.

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.

**Caution** FinderCopy and FinderMove replace existing files without asking. If you copy or move File A to Folder B, and File A already exists in the folder, the file is replaced with the version being copied or moved. The Finder moves the file being replaced to the Trash. If you don’t want this to occur, use File.Exists to see if the file already exists in the destination folder.

**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 command (page 215), FindFolder function (page 216)

# FinderMove command

**Syntax** FinderMove [*path-list*,] *destination* [, *wait-for-completion*]

**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 command (page 214), FindFolder function (page 216)

## FindFolder function

**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, the default path to each folder, and the minimum Mac OS version that recognizes each folder code. The default paths assume the startup disk is named "HD."

| Code | Default path to folder | OS version |
|---|---|---|
| root | HD: | 8.0 |
| aexƒ | HD:Apple Extras: | 8.0 |
| apps | HD:Applications: | 8.0 |
| astƒ | HD:Assistants: | 8.0 |
| flnt | HD:Cleanup At Startup: | 8.0 |
| desk | HD:Desktop Folder: | 7.x |
| docs | HD:Documents: | 8.0 |
| ilgf | HD:Installer Logs: | 8.5 |
| intƒ | HD:Internet: | 8.5 |
| morƒ | HD:Mac OS Read Me Files: | 8.0 |
| odst | HD:Stationery: | 8.0 |
| macs | HD:System Folder: | 7.x |
| appr | HD:System Folder:Appearance: | 8.5 |
| dtpƒ | HD:System Folder:Appearance:Desktop Pictures: | 8.5 |
| snds | HD:System Folder:Appearance:Sound Sets: | 8.5 |
| thme | HD:System Folder:Appearance:Theme Files: | 8.1 |
| amnu | HD:System Folder:Apple Menu Items: | 7.x |
| rapp | HD:System Folder:Apple Menu Items:Recent Applications: | 8.5 |

| Code | Default path to folder | OS version |
|---|---|---|
| rdoc | HD:System Folder:Apple Menu Items:Recent Documents: | 8.5 |
| rsvr | HD:System Folder:Apple Menu Items:Recent Servers: | 8.5 |
| spki | HD:System Folder:Apple Menu Items:Speakable Items: | 8.5 |
| asup | HD:System Folder:Application Support: | 8.0 |
| prof | HD:System Folder:ColorSync Profiles: | 8.1 |
| cmnu | HD:System Folder:Contextual Menu Items: | 8.0 |
| ctrD | HD:System Folder:Control Panels (Disabled): | 8.0 |
| ctrl | HD:System Folder:Control Panels: | 7.x |
| sdev | HD:System Folder:Control Strip Modules: | 8.0 |
| oded | HD:System Folder:Editors: | 8.0 |
| odod | HD:System Folder:Editors:OpenDoc: | 8.0 |
| odsp | HD:System Folder:Editors:OpenDoc:OpenDoc Shell Plug-Ins: | 8.0 |
| extD | HD:System Folder:Extensions (Disabled): | 8.0 |
| extn | HD:System Folder:Extensions: | 7.x |
| fnds | HD:System Folder:Extensions:Find: | 8.5 |
| walk | HD:System Folder:Extensions:Location Manager Modules: | 8.1 |
| ƒmod | HD:System Folder:Extensions:Modem Scripts: | 8.0 |
| odlb | HD:System Folder:Extensions:OpenDoc Libraries: | 8.0 |
| ppdf | HD:System Folder:Extensions:Printer Descriptions: | 8.0 |
| fvoc | HD:System Folder:Extensions:Voices: | 8.0 |
| favs | HD:System Folder:Favorites: | 8.1 |
| font | HD:System Folder:Fonts: | 7.x |
| ƒhlp | HD:System Folder:Help: | 8.0 |
| issf | HD:System Folder:Internet Search Sites: | 8.5 |
| laun | HD:System Folder:Launcher Items: | 8.5 |
| pref | HD:System Folder:Preferences: | 7.x |
| trip | HD:System Folder:Preferences:Location Manager Prefs: | 8.1 |
| fall | HD:System Folder:Preferences:Location Manager Prefs:Locations: | 8.1 |
| oclk | HD:System Folder:Preferences:OneClick Folder: | (special) |
| prnt | HD:System Folder:PrintMonitor Documents: | 7.x |
| ƒscr | HD:System Folder:Scripting Additions: | 8.0 |
| scrƒ | HD:System Folder:Scripts: | 8.5 |
| fasf | HD:System Folder:Scripts:Folder Action Scripts: | 8.5 |
| shdD | HD:System Folder:Shutdown Items (Disabled): | 8.0 |
| shdf | HD:System Folder:Shutdown Items: | 7.x |
| strD | HD:System Folder:Startup Items (Disabled): | 8.0 |

| Code | Default path to folder | OS version |
|---|---|---|
| strt | HD:System Folder:Startup Items: | 7.x |
| macD | HD:System Folder:System Extensions (Disabled): | 8.0 |
| ƒtex | HD:System Folder:Text Encodings: | 8.0 |
| temp | HD:Temporary Items: | 7.x |
| fbcf | HD:TheFindByContentFolder: | 8.5 |
| empt | HD:Trash: (network trash folder) | 7.x |
| trsh | HD:Trash: (desktop trash can) | 7.x |
| utiƒ | HD:Utilities: | 8.0 |

Items in the Desktop Folder appear on the desktop.

**Notes** Newer versions of Mac OS may support additional folder codes. FindFolder will automatically support the new codes.

The folder code “oclk” isn’t part of the Mac OS but can be used to specify the path to the OneClick Folder in the Preferences folder.

To type the character “ƒ” used in some folder codes, press Option-f.

**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
```

# FKey command

**Syntax** FKey *FKey-number*

**Description** Calls the specified FKey. FKey-number must be a number from 0 (zero) to 9.

**Examples**

```
// Captures the screen
FKey 3
```

## FontMenu function

**Syntax** FontMenu *font-list* [, *cache-file*]

**Description** Pops up a WYSIWYG font manu and returns the name of the selected font. If you include the path to a cache file, FontMenu stores the bitmap for the font menu in the file so it doesn't need to be regenerated after restarting the system.

**Examples**

```
On MouseDown
	Variable theFont
	theFont = FontMenu (GetResources "FONT"), "Mac HD:Desktop Folder:MyFontCache"
	SelectMenu "Font", theFont
End MouseDown
```

## For, Next For, Exit For, End For commands

**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 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.

If *end* is greater than *start*, then the For loop counts down from *end* to *start*, subtracting 1 from *index-variable* each time through the loop. For example:

```
For X = 1 to 3    // goes 1, 2, 3
For X = 3 to 1    // goes 3, 2, 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 commands (page 277)]

# Gestalt function

**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. Gestalt is based on the Macintosh toolbox call of the same name.

*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.

An Apple-defined selector codes fall into two categories: environmental selectors, which supply specific environmental information you can use to control the behavior of your script, and informational selectors, which supply information you can't use to determine what hardware or software features are available. You can use one of the selector codes defined by Apple (listed in the "Constants" section beginning on page 1-14 of *Inside Macintosh: Operating System Utilities*) or a selector code defined by a third-party product.

**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: Operating System Utilities*, page 1–31

## GetDragAndDrop function

**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, X, theFile
      // get the number of files dropped
```

```
          fileCount = ListCount GetDragAndDrop
          For X = 1 to fileCount
                    // get the path of the dropped file from the list of dropped files
                    theFile = GetDragAndDrop X
                    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 131), DragAndDrop handler (page 192)

## GetICHelpers function

**Syntax** GetICHelpers

**Description** Returns a list of all the helper applications from Internet Preferences. Each item in the list contains the text “Helper•” followed by a protocol scheme, such as http, mailto, ftp, and so on. Use GetICPref to get a path to the preferred helper application for a specific item in the list.

**Examples**

```
Variable helperList, theHelper, pathList, X
helperList = GetICHelpers
For X = 1 To ListCount helperList
          theHelper = ListItems helperList, X
          pathList = pathList & theHelper & " – " & (GetICPref theHelper) & Return
End For
X = AskList pathList, "Paths to helper applications stored in Internet Preferences:"
```

**See Also** GetICPref function (page 222), SetICPref command (page 291)

**Author Info** GetICHelpers, part of IC Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## GetICPref function

**Syntax** GetICPref *preferenceKey*

**Description** Returns the value of the specified preference from Internet Preferences. Only simple text string preferences can be retrieved; complex preferences (such as font settings, color settings, extension mappings, and so on) cannot be retrieved.

Following is a table of preference keys supported by Internet Config 2.0 and accessible with GetICPref and SetICPref. Other applications (such as Microsoft Internet Explorer) may also store data in the Internet Preferences file; preference keys for those applications can be accessed through GetICPref and SetICPref but are not listed here.

| IC preference key | Value |
|---|---|
| ArchiePreferred | preferred Archie server |
| DownloadFolder | path to the folder where newly downloaded files are put |
| Email | user@host.domain, email address of user, ie. return address |
| FingerHost | host.domain, default finger server |
| FTPHost | host.domain, default FTP server |
| FTPProxyAccount | second level FTP proxy authorization |
| FTPProxyHost | host.domain |
| FTPProxyPassword | password for FTPProxyUser |
| FTPProxyUser | first level FTP proxy authorization |
| GopherHost | host.domain, default Gopher server |
| GopherProxy | host.domain |
| Helper • | helpers for URL schemes |
| HTTPProxyHost | host.domain |
| InfoMacPreferred | preferred Info-Mac server |
| IRCHost | host.domain, Internet Relay Chat server |
| LDAPSearchbase | LDAP thing |
| LDAPServer | host.domain |
| MacSearchHost | host for MacSearch queries |
| MailAccount | user@host.domain, POP3 or IMAP account from which to fetch mail |
| MailHeaders | extra headers for outgoing mail messages |
| MailPassword | password for MailAccount |
| NewMailSoundName | sound to play when new mail arrives |

| IC preference key | Value |
|---|---|
| NewsAuthPassword | password for NewsAuthUsername |
| NewsAuthUsername | user name for NNTP news servers that require authorization |
| NewsHeaders | extra headers for outgoing news messages |
| NNTPHost | host.domain, NNTP server |
| NTPHost | host.domain, Network Time Protocol (NTP) |
| Organization | for X-Organization string in outgoing mail and news messages |
| PhHost | host.domain, default Ph server |
| Plan | default response for finger servers |
| QuotingString | used to quote responses in news and mail (usually ">") |
| RealName | real name of user |
| Signature | appended to outgoing mail and news messages |
| SMTPHost | host.domain, SMTP server |
| SnailMailAddress | preferred postal mailing address |
| SocksHost | host.domain |
| TelnetHost | host.domain, default Telnet address |
| UMichPreferred | preferred UMich server |
| WebSearchPagePrefs | URL, user's default search page |
| WhoisHost | host.domain, default whois server |
| WWWHomePage | URL, user's default web page |

## Examples

```
Variable userName, signature
userName = GetICPref "RealName"
signature = GetICPref "Signature"
```

## See Also

GetICHelpers function (page 222), SetICPref command (page 291)

## Author Info

GetICPref, part of IC Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

# GetPalettes function

**Syntax** GetPalettes *file*

**Description** Returns the list of palette names within *file*. Palettes that are flagged as visible have a "*" character as a prefix.

**Examples**

```
// Import one or more palettes from the file dropped on the button.
On DragAndDrop
    Variable theFile, palList, selectedList, palName, theChoice, X
    theFile = GetDragAndDrop 1
    // Make sure the dropped file is really a palette file.
    If File(theFile).Kind <> "Btns"
        Message "That doesn't look like a palette file."
        Exit
    End If
    // Here is where we get the names of the palettes in the file.
    palList = GetPalettes theFile
    // Let the user choose which palettes to import.
    selectedList = AskList palList, "Import which palettes?"
    If selectedList
        theChoice = AskButton "Import as Global?", "Global", Process.Name
        // Import each palette selected in the list
        For X = 1 To ListCount selectedList
            palName = ListItems selectedList, X
            // Remove the * prefix if it exists.
            If (SubString palName, 1, 1) = "*"
                palName = SubString palName, 2, Length palName
            End If
            // Import the palette
            If theChoice = 1 // global
                Palette(palName).New Global = palName, theFile
            Else  // app-specific
                Palette(palName).New = palName, theFile
            End If
            Palette(palName).Visible = 1
        End For
    End If
End DragAndDrop
```

# GetResources function

**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 "
```

## GetScrap function

**Syntax** GetScrap *resourceType*

**Description** Returns the data of the specified resource type from the Clipboard. Use GetScrap and SetScrap instead of the Clipboard system variable when you want to work with data types other than plain text.

**Examples**

```
// Get the picture on the Clipboard and store it in a variable
Variable thePictureData
thePictureData = GetScrap “PICT"

// Get the TEXT/styl (styled text) resource pair from the Clipboard and store it in a resource file
// Copy some styled text to the Clipboard from a styled-text-aware application
// (such as SimpleText or AppleWorks) before running this script
Variable theText, theStyle, theFile
theFile = "Mac HD:Desktop Folder:My Resource File"
theText = GetScrap "TEXT"
theStyle = GetScrap "styl"
SetResource theFile, theText, "TEXT", 128
SetResource theFile, theStyle, "styl", 128
```

**Author Info** GetScrap, part of Scrap Extension
Copyright © 1999 Life OnLine Software (lr). All rights reserved.

## GetWindowText command

**Syntax** GetWindowText *text* [, *window-specifier* [, *left*, *top*, *right*, *bottom*] ]

## Description

Captures the text from the specified window, or the frontmost window if no window is specified, and puts the text in a variable.

To specify that only the text from a portion of the window be captured, specify the coordinates *left*, *top*, *right*, and *bottom*. If you don’t include the coordinates, OneClick captures the text from the entire window.

Any unspecified coordinates will use the window’s coordinates. For example, if you don’t specify *right* and *bottom*, OneClick assumes the window’s right and bottom edges.

## Examples

```
// Display the text from the bottom-right corner (status bar area) of a FrameMaker window.
Variable theStatus
GetWindowText theStatus, 1, 1, Window.Height - 34// capture 34 pixels from window bottom
Message theStatus

// This script "watches" for prompts in a communications window.
// It waits for the string "login:" to appear, then types the login name.
// It then waits for the string "Password:" to appear, then types the password.

Variable telnetScreen

// Put all the text in the frontmost window into a variable named telnetScreen.
GetWindowText telnetScreen

// Repeatedly get the window's text, once per second, until the
// string "login:" is found somewhere in the text.
While NOT (Find "login:", telnetScreen)
    Pause 10
    GetWindowText telnetScreen
End While

// When the previous loop exits, it means the string "login:" was found
// in the window, so type our login name.
Type "annie_douglas<return>"

// Now do the same for the Password prompt.
GetWindowText telnetScreen
While NOT (Find "Password:", telnetScreen)
    Pause 10
    GetWindowText telnetScreen
End While
Type "golddigger<return>"
```

# If, Else, Else If, End If commands

**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<return>Green<return>Blue"
If TheMenu = "Red"
      Type "Roses"
Else If TheMenu = "Green"
      Type "Grass"
Else If TheMenu = "Blue"
      Type "Ocean"
End If
```

# IgnoreClicks system variable

**Description** When set to True (1), OneClick causes the system to ignore all mouse activity except clicks on OneClick palettes. Use IgnoreClicks when your script musn't be interrupted by the user clicking around in applications while your script runs. Keyboard input is still allowed.

IgnoreClicks is automatically reset to 0 when the script ends so there is no need to reset it yourself at the end of a script.

**Example**

```
Button.Text = "Processing..."
Button.Color = 36 // red
Button.Update
// Prevent the user from clicking in applications during lengthy processing.
IgnoreClicks = 1
// Do some lengthy processing here that shouldn't be interrupted
// -- generate web pages, batch operations in Photoshop, copy files, etc.
Call "Do Much Stuff", "Work Palette"
// When done, restore mouse activity
IgnoreClicks = 0
// Continue processing, allowing mouse activity
Button.Color = 0
Button.Text = "All done!"
```

# Implemented function

**Syntax** Implemented *keyword*

**Description** Returns True (1) if the specified keyword is implemented in the current version of OneClick. Use Implemented to determine if a certain OneClick extension is loaded, or if a keyword is available in the active version that wasn't available in an earlier version. If a keyword is not implemented, it appears as "???" in the Script Editor.

**Examples**

```
// Check to see if the QuoteText extension (external command) is loaded
If Implemented QuoteText
    Clipboard = QuoteText Clipboard, 60, "> "
Else
    Message "The QuoteText extension isn't loaded."
End If

// See if we can use the ListFind function (not available in OneClick 1.0 and earlier)
If Implemented ListFind
```

```
    theItem = ListFind theList, searchText
Else
    Message "ListFind is not supported, please upgrade your version of OneClick!"
End If
```

**See Also** Version system variable (page 301)

## IsKeyDown system variable

**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 system variable (page 230)

## IsMouseDown system variable

**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 system variable (page 230)

## KeyPress command

**Syntax** KeyPress [Command] [Option] [Control] [Shift] *character(s)*

**Description** Types the specified character as if you had typed it from the keyboard.

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

Keystrokes typed with KeyPress may be intercepted by any Mac OS extension that allows keyboard shortcuts, such as the Control Strip, OneClick buttons, and other macro utilities. For example, if you have a OneClick button set to use the shortcut Command-Y, using KeyPress to type Command-Y will activate that OneClick button.

The difference between the Type command and KeyPress is that Type sends typed keystrokes only to the active application, bypassing any OS-level extensions that may be intercepting keystrokes. Type can type any amount of text to the active application, but KeyPress is limited to the size of the Mac OS event queue, which is 12 in Mac OS 7, 24 in Mac OS 8, and 48 in Mac OS 9.

**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**

```
// presses Command-B, then Command-I
KeyPress Command "bi"
```

```
// presses Command-Control-F12
KeyPress Command Control "<f12>"
```

**See Also** [Type command (page 298)]

## LaunchURL command

**Syntax** LaunchURL theURL

**Description** Launches a URL using a helper application set in Internet Config.

**Examples**

```
// Lauches your web browser and goes to www.westcodesoft.com
LaunchURL "http://www.westcodesoft.com/"

// Opens a new email message addressed to support@westcodesoft.com
LaunchURL "mailto:support@westcodesoft.com"
```

**Author Info** LaunchURL Extension
Copyright © 1998 Dan Crevier. All rights reserved.

## Length function

**Syntax** Length *text*

**Description** Returns the number of characters in *text*.

**Examples**

```
// Types: 6
Type Length "Banana"

// Types: 0
Type Length ""
```

## ListCount function

**Syntax** ListCount *list*

**Description** Returns the number of items in the list.

**Examples**

```
// Types: 5
Type ListCount "Red<return>Orange<return>Yellow<return>Green<return>Blue"

// Types: 4 (The empty string after the last <return> counts as one item)
```

```
Type ListCount "Red<return>Orange<return>Yellow<return>"

// Types the number of fonts in the Font menu
Type ListCount Menu("Font").List
```

**See Also** OldListCount function (page 251)

## ListDelete function

**Syntax** ListDelete *list*, *start* [, *end*]

**Description** Deletes one or more items from a list and returns the new list. *Start* indicates the first item to delete and *end* indicates the last item to delete. If *end* is not specified, ListDelete removes all items from *start* to the end of the list.

**Examples**

```
Variable oldList newList foundItem
oldList = "Red,Yellow,Orange,Green,Blue,Purple"
ListDelimiter = ","

// Delete the first item
newList = ListDelete oldList, 1, 1
Type newList, Return

// Delete the second, third, and fourth item
newList = ListDelete oldList, 2, 4
Type newList, Return

// Delete all items from the third item to the end of the list
newList = ListDelete oldList, 3
Type newList, Return

// Delete the last item
newList = ListDelete oldList, (ListCount oldList), (ListCount oldList)
Type newList, Return

// Delete the item "Orange"
foundItem = ListFind oldList, "Orange"
newList = ListDelete oldList, foundItem, foundItem
Type newList, Return
```

**See Also** ListCount function (page 232), ListFind function (page 235)

# ListDelimiter system variable

**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, shown in a script as <return>.

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 (<return>) before using a list delimited by carriage returns. The ListDelimiter value is reset to <return> 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 120)

## ListFind function

**Syntax** ListFind *list*, *item*

**Description** Searches a list for *item* and returns the position of the first occurrence of *item*, or 0 (zero) if the item isn't found. Searches are not case-sensitive.

ListFind matches whole items only, not partial strings within an item. To mach partial strings, use the '?' wildcard character to match a single character or the '*' wildcard to match zero or more characters.

**Examples**

```
Variable theList foundItem
theList = "Red,Yellow,Orange,Green,Blue,Purple"
ListDelimiter = ","

// Types: 3 (orange is item 3 in the list)
foundItem = ListFind theList, "ORANGE"
Type foundItem, Return

// Types: 0 (item not found in the list, string “urp” does not match item “Purple”)
foundItem = ListFind theList, "urp"
Type foundItem, Return

// Types: 6 (using wildcard characters, string “*urp*” matches item “Purple”)
foundItem = ListFind theList, "*urp*"
Type foundItem, Return
```

**See Also**

## ListInsert function

**Syntax** ListInsert *list*, *text*, *position*

**Description** Inserts *text* as a new item into a list before the specified item *position* and returns the new list.

**Examples**

```
Variable oldList newList
oldList = "Green,Blue,Purple"
ListDelimiter = ","

// Insert “Cyan” before item 2 (“Blue”)
// Types: Green,Cyan,Blue,Purple
newList = ListInsert oldList, "Cyan", 2
```

```
Type newList, Return

// Insert three new items at the beginning of the list
// Types: Red,Yellow,Orange,Green,Blue,Purple
newList = ListInsert oldList, "Red,Yellow,Orange", 1
Type newList, Return

// Insert a new item before the last item
// Types: Green,Blue,Blueberry,Purple
newList = ListInsert oldList, "Blueberry", (ListCount oldList)
Type newList, Return

// ListInsert doesn’t insert items after the last item. Instead, use the & operator
// to add new items at the end of a list.
// Types: Green,Blue,Purple,Indigo
newList = oldList & ListDelimiter & "Indigo"
Type newList, Return
```

**See Also**

# ListItems function

**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<return>Green<return>Blue<return>Yellow<return>Brown", 2, 4

// Types: Blue, Yellow, Brown (all on separate lines)
Type ListItems "Red<return>Green<return>Blue<return>Yellow<return>Brown", –3, –1

// Assuming the SystemFolder path is "Mac HD:System Folder", types "Mac HD"
ListDelimiter = ":"
Type ListItems SystemFolder, 1
```

**See Also** ListDelimiter system variable (page 234)

# ListSort function

**Syntax** ListSort *list*

**Description** Sorts all list items in alphabetical order.

**Examples**

```
// Types: Blue, Green, Red (all on separate lines)
Type ListSort "Red<return>Green<return>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 function

**Syntax** ListSum *list*

**Description** Returns the sum of all the numbers in the list.

**Examples**

```
// Types:30
Type ListSum "10<return>35<return>–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 "<rightarrow><return>"
// Type the sum of all the numbers on the Clipboard
Type ListSum Clipboard
```

# LoadExtensions command

**Syntax** LoadExtensions *file*

**Description** Loads or reloads OneClick extensions (external commands, functions, or system variables) from the specified file. Use LoadExtensions to install or update a OneClick extension without having to restart.

**Example**

```
// Load the QuoteText extension from a file on the desktop
LoadExtensions "Mac HD:Desktop Folder:QuoteText Extension"
```

```
// Reload all extensions in OneClick’s Extensions folder
Variable theFolder theFile fileList X
theFolder = (FindFolder "oclk") & "Extensions:" // get path to OneClick Extensions folder
fileList = File(theFolder).List // get list of extension files in folder
For X = 1 To ListCount fileList
    theFile = theFolder & ListItems fileList, X
    LoadExtensions theFile
End For
```

## Lower function

**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 function (page 299), Proper function (page 275)

## MakeNumber function

**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 function (page 239)

## MakeText function

**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 function (page 238)

# Menu object

**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 index 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 index 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.

Menus that have no name (such as icon menus) are specified by menu ID either as a number or as a string. The format for strings is "[menu_ID]". Menu.List returns menu ID numbers as strings for these kinds of menus. For example, the MegaPhone menu (which has menu ID of –16400) would be "[–16400]" in the menu list. You could use either of the following to get the list of items in the MegaPhone menu, assuming MegaPhone is installed on your system.

```
Menu(–16400).List
Menu("[–16400]").List
```

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.

### .Count

Returns the number of menus in the menu bar or the number of items in the specified menu.

```
// Display the number of menus in the menu bar
Message Menu.Count

// Display the number of items in the File menu
Message Menu("File").Count
```

### .Checked

Returns True (1) if the specified menu item is checked, or False (0) if it's unchecked.

If you specify a menu instead of a menu item, the Checked property returns a list of all checked items in the menu.

```
// 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

// Display an AskList dialog box listing all View menu items with checked menu items highlighted
Variable theResponse
theResponse = AskList Menu("View").List, "Pick an item", Menu("View").Checked
```

### .Enabled

Returns True (1) if the specified menu or menu item is enabled (not dimmed), or False (0) if it's dimmed.

```
// Check the Stop Loading menu item every 0.5 seconds until it's no longer enabled
// When it becomes disabled, beep twice to indicate Netscape's page load is complete
On MouseDown
     Schedule 5
End MouseDown
```

```
On Scheduled
        Menu("Go").Update
        If NOT Menu("Go", "Stop Loading").Enabled
                Beep
                Beep
                Schedule 0
        End If
End Scheduled
```

**.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
```

**.Height** Returns the height of the menu bar in pixels. Using Menu.Height instead of the default value 20 is useful if the script may be run on systems that use a nonstandard menu bar height. This property is read-only and no menu specifier is necessary.

```
// Position the palette just below the menu bar
Palette.Top = Menu.Height + 1
```

**.Index** Returns the corresponding index number for the menu or menu item when the menu or item is specified by name (1 for the first menu, 2 for the second, and so on). For the Menu object, this property is read-only.

**.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
```

## MenuNumber function

**Syntax**

```
MenuNumber
```

**Description**

Returns the number of the item chosen from a PopupMenu function. (PopupMenu returns the text of the chosen item; MenuNumber returns the item's position in the menu list.) MenuNumber returns 0 (zero) if you release the mouse button without choosing an item from the pop-up menu.

**Example**

```
Variable theChoice
theChoice = PopupMenu "Apple<return>Lemon<return>Strawberry"
If MenuNumber = 1
        Message "You picked the first item"
End If
```

**See Also**

PopupMenu function (page 265)

## Message command

**Syntax**

Message *text*

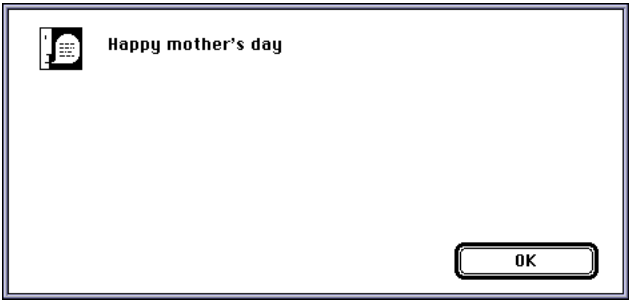
**Description**

Displays *text* 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.

*Text* is limited to 250 characters.

**Examples**

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


Happy mother's day

Sample Message dialog box

**See Also** Notify command (page 250), AskButton function (page 152)

## MountVolume command

**Syntax** MountVolume *zone*, *server*, *volume* [, *username*] [, *server-password*] [, *volume-password*]

**Description** Mounts an AppleShare server volume using AppleTalk without displaying the "Connect to the file server" dialog box or server greeting message.

*Zone*, *server*, and *volume* names are all required. If you omit both *username* and *server-password*, MountVolume attempts to mount the volume as a guest.

MountVolume looks for the server in the local zone if *zone* is either empty or "*".

*Volume-password* is required only if the volume has its own separate password. Most AppleShare volumes do not use a volume password.

**Note** If you are already logged in to a server and attempt to mount another volume from the same server, the new volume is mounted with your previous user name and password. (The Chooser works the same way.) To mount the volume with a new name and password, unmount all volumes from that server first.

If MountVolume cannot mount a volume, the Error system variable contains a number indicating why the volume couldn't be mounted. Common errors include the following.

| Error | Meaning |
|---|---|
| 3 | Parameter error in script (zone, server, or volume not specified) |
| –28 | AppleTalk is inactive |
| –35 | Volume not found on server |
| –108 | Out of memory |
| –5000 | Access denied (no permission to mount volume) |
| –5016 | Server not found or not responding |
| –5023 | Authentication failed (incorrect user name or password) |
| –5042 | Password expired |
| –5061 | Maximum volumes mounted |
| –5062 | Volume already mounted |

## Examples

```
// Mounts the volume “CD-R HD” on “Mastering Server”
// in the zone “4th Floor”, logging in as user “Joe” with password “secret”
MountVolume "4th Floor", "Mastering Server", "CD-R HD", "Joe", "secret"

// Mounts the volume “Backup HD” on “CondoNet Backup Server”
// in the local zone, logging in as Guest.  Checks to make sure the volume was mounted,
// then opens the volume in Finder if successful, otherwise displays the error number.
Variable mvError
MountVolume "*", "CondoNet Backup Server", "Backup HD"
mvError = Error
If mvError // check to see if the mount failed (0 = success)
      Message "MountVolume error: " & mvError
Else
      Open "Backup HD:"
End If
```

## Author Info

MountVolume, part of MountVolume Extension
Copyright © 1999 Jeff Jungblut. All rights reserved.

## See Also

MountVolumeIP command (page 247), .Unmount (page 303)

## MountVolumeIP command

**Syntax** MountVolumeIP *afp-url* [, *show-login*] [, *show-greeting*]

**Description** Mounts an AppleShare server volume using TCP/IP.

*Afp-url* is a uniform resource locator that contains the server IP address, volume to mount, and an optional user name and password. AFP URLs use the following format:

```
afp://username:password@server-address/volumename/
```

To mount a volume as a guest, use this format:

```
afp://server-address/volumename/
```

If *show-login* is True (nonzero), MountVolumeIP prompts you for your user name and password in the AppleShare login dialog box instead of using the name and password in the URL. If *show-login* is False (0) or omitted, no login dialog box appears. If you do not supply a user name and password in the URL and don't set the *show-login* parameter to 1, MountVolumeIP attempts to mount the volume as a guest. The default value for *show-login* is False (0).

If *show-greeting* is True (nonzero), the server greeting message (if any) appears when you log in. The default value for *show-greeting* is False (0).

MountVolumeIP ignores any extra path information following the volume name in the URL. For example, MountVolumeIP "afp://169.254.44.176/My Disk/My Folder/My Doc/" mounts the volume My Disk, but does not open My Doc in My Folder.

**Note** If you are already logged in to a server and attempt to mount another volume from the same server, the new volume is mounted with your previous user name and password. (The Chooser works the same way.) To mount the volume with a new name and password, unmount all volumes from that server first.

If MountVolumeIP cannot mount a volume, the Error system variable contains a number indicating why the volume couldn't be mounted. Common errors include the following.

| Error | Meaning |
|---|---|
| 3 | Parameter error in script (no URL specified) |
| –28 | AppleTalk is inactive |
| –35 | Volume not found on server |
| –108 | Out of memory |
| –128 | User cancelled the login dialog box |
| –5000 | Access denied (no permission to mount volume) |
| –5016 | Server not found or not responding |
| –5019 | AFP parameter error (usually means login is disabled) |
| –5023 | Authentication failed (incorrect user name or password) |
| –5042 | Password expired |
| –5061 | Maximum volumes mounted |
| –5062 | Volume already mounted |

## Examples

```
// Mounts the volume “CD-R HD” on “mastering.westcodesoft.com”,
// logging in as user “Joe” with password “secret”
MountVolumeIP "afp://Joe:secret@mastering.westcodesoft.com/CD-R HD/"

// Mounts the volume “Backup HD” on “169.254.44.176”, logging in as Guest.
// Checks to make sure the volume was mounted, then opens the volume
// in Finder if successful, otherwise displays the error number.
Variable mvError
MountVolumeIP "afp://169.254.44.176/Backup HD"
mvError = Error
If mvError // check to see if the mount failed (0 = success)
    Message "MountVolumeIP error: " & mvError
Else
    Open "Backup HD:"
End If
```

## Author Info

MountVolumeIP, part of MountVolume Extension
Copyright © 1999 Leonard Rosenthol and Jeff Jungblut. All rights reserved.

**See Also** MountVolume command (page 245), .Unmount (page 303)

## MouseDown handler

**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 handler (page 249), IsMouseDown system variable (page 230)

## MouseUp handler

**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 handler (page 249), IsMouseDown system variable (page 230)

# Notify command

**Syntax** Notify *text*

**Description** Displays *text* in a dialog box or a floating window. Under Mac OS 8.6 and earlier, the text appears in a modal dialog box with an OK button, just like the Message command. Under Mac OS 9, the text appears in a nonmodal floating window.

When you use Notify to display text, the script continues running while the message is still on the screen. When you use Message to display text, the script stops running until you click OK.

*Text* is limited to 250 characters.

**Examples**

```
Notify "Hello, world!"

Variable msg
msg = "Your time-consuming process has finished."
msg = msg & Return & Return & (DateTime.DateString 34)
msg = msg & " " & (DateTime.TimeString 3)
Notify msg
```

**Author Info** Notify Extension
Copyright © 1999 Jeff Jungblut. All rights reserved.

**See Also** Message command (page 244), AskButton function (page 152)

# OldDate function

**Syntax** OldDate [*format*]

**Description** Returns the current date formatted as a string.

This function is obsolete and is provided only for backward compatibility with OneClick 1.0 scripts. Use DateTime.DateString instead.

**See Also** DateTime object (page 181)

## OldDateString function

**Syntax** OldDateString *secondsSince1904* [, *format*]

**Description** Returns the date as a string, given the number of seconds since 1904.

This function is obsolete and is provided only for backward compatibility with OneClick 2.0 Preview scripts. Use DateTime.DateString instead.

**See Also** DateTime object (page 181)

## OldListCount function

**Syntax** OldListCount *list*

**Description** Returns the number of items in the list. If the list ends with a delimiter, the last item (the empty string) is **not** counted.

**Note** This function is obsolete and is provided only for backward compatibility with OneClick 1.0 scripts. Use ListCount instead.

**Examples**

```
// Types: 5
Type OldListCount "Red<return>Orange<return>Yellow<return>Green<return>Blue"

// Types: 5 (The empty string after the last <return> is not counted)
Type OldListCount "Red<return>Orange<return>Yellow<return>Green<return>Blue<return>"

// Types: 5
Type ListCount "Red<return>Orange<return>Yellow<return>Green<return>Blue"

// Types: 6 (The empty string after the last <return> is counted)
Type ListCount "Red<return>Orange<return>Yellow<return>Green<return>Blue<return>"
```

**See Also** ListCount function (page 232)

## OldListItems function

**Syntax** ListItems *list*, *start* [, *end*]

**Description** Returns a portion of the specified *list*. Works similar to ListItems, except that if the list ends with the list delimiter, OldListItems ignores the delimiter.

**Note** This function is obsolete and is provided only for backward compatibility with OneClick 1.0 scripts. Use ListItems instead. To get the name of a file or folder from a path which might end with a colon, use the File.Name property instead (see page 204).

**Example**

```
Variable thePath
thePath = "Mac HD:System Folder:"
ListDelimiter = ":"
// Displays an empty message box -- the last item in the list is the empty string
Message ListItems thePath, - 1
// Displays “System Folder”, ignoring the list delimiter at the end of the list
Message OldListItems thePath, - 1
```

**See Also** ListItems function (page 236), ListDelimiter system variable (page 234)

# OldTime function

**Syntax** OldTime [*format*]

**Description** Returns the current time formatted as a string.

This function is obsolete and is provided only for backward compatibility with OneClick 1.0 scripts. Use DateTime.TimeString instead.

**See Also** DateTime object (page 181)

# OldTimeString function

**Syntax** OldTimeString *secondsSince1904* [, *format*]

**Description** Returns the time as a string, given the number of seconds since 1904.

This function is obsolete and is provided only for backward compatibility with OneClick 2.0 Preview scripts. Use DateTime.TimeString instead.

**See Also** DateTime object (page 181)

## OnlineHelp handler

**Description** The OnlineHelp handler is a handler generated automatically by the Online Help Editor palette and inserted into an existing script on one of your own palettes. If you want to add online help to a palette, use the Online Help Editor to create the help screens and insert the OnlineHelp handler into one of your palette's scripts, then have one of your scripts call the OnlineHelp handler to load and display your help screens in the Online Help palette.

The Online Help and Online Help Editor palettes are included with OneClick. For more information on developing online help with the Online Help Editor, import the Online Help Editor palette (in the Developer Goodies folder) and click the palette's Help button.

## Open command

**Syntax** Open *path-list* [, *path-to-application*]

**Description** Opens the specified application, document, control panel, desk accessory, or folder. (The Open command can open anything the Finder can open.) *Path-list* is either a full path or a list of full paths.

If you include the optional *path-to-application* parameter, OneClick opens the file using the application at the specified path 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 "Mac HD:Quicken Deluxe 98:My Accounts"

// Open three documents in Mac HD:Administrative Stuff:
Variable theDir
theDir = "Mac HD:Administrative Stuff:"  // prepend theDir to each file name
Open theDir & "Status Report<return>" & theDir & "Month-End<return>" & theDir & "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 "Mac HD:Picture 1", "Mac HD:Applications:Photoshop:Adobe® Photoshop® 5.0.2"
If Error = 1
	Message "Not enough memory to open Photoshop."
End If

// Open the item(s) dropped on the button in BBEdit (creator code for BBEdit is "R*ch")
On DragAndDrop
	Open GetDragAndDrop, FindApp "R*ch"		// FindApp "R*ch" returns full path to BBEdit
End DragAndDrop
```

**See Also** Directory system variable (page 191)

## OpenFileList function

**Syntax** OpenFileList [*volume* [, *include-font-files*] ]

**Description** Returns a list containing the full paths of all open files. This is useful for determining which files are still open if the Mac won’t unmount a volume because it contains files that are in use, for example.

Without any parameters, OpenFileList returns a list of all open files on all volumes. If you specify a volume name in the first parameter, OpenFileList returns a list of open files on that volume only. If you include 1 (True) in the second parameter, the paths to open font files are also included in the list of files returned.

A file is included in the list twice if both its resource fork and data fork are open.

**Examples**

```
// Display a list box showing all open files (including font files) on volume Mac HD
Variable X
X = AskList (OpenFileList "Mac HD", 1)
```

## OpenResFile function

**Syntax** OpenResFile *file*

**Description** Opens a file’s resource fork into the current resource chain.

**Note** This is a technical Mac OS function and should be used at your own risk.

**Examples**

```
Variable refNum
// Open the resource fork of a file containing sound resources
// and put the file’s reference number in refNum.
refNum = OpenResFile "Mac HD:YoYoLand™:Sound Library"
// Play a sound contained in the resource file.
Sound "Mystery"
// Close the file referred to by refNum.
CloseResFile refNum
```

**Author Info** OpenResFile, part of Resource Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## OptionKey system variable

**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, X
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 X = 1 To appCount
        theApp = ListItems listOfApps, X
        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 system variable (page 177), ControlKey system variable (page 179), ShiftKey system variable (page 292)

## Palette object

**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.

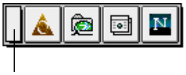
You can also specify a palette by number, which is useful for looping through all the available palettes and performing some operation on each palette. Palette(–1) refers to the palette under the cursor, if any.

**.Color** Gets or sets the color of the palette's background. Colors are numbered 1–256; see the Button.Color property (page 159) 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.



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
```

**.Front** The Front message brings the specified palette to the front so that it overlaps any other palettes.

**.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
```

**.Index** Returns the corresponding index number for the palette when the palette is specified by name (1 for the first palette, 2 for the second, and so on). For the Palette object, this property is read-only.

**.InMenu** Enables or disables the display of the palette's name in the OneClick menu. (This is the same as the "Display in Menu" checkbox in the Palette Editor.) If the property is 1 (True), the palette is always listed in the OneClick menu. If the property is zero (False), the palette appears in the OneClick menu if it is open, but not when it is closed. All palettes are listed in the menu when the OneClick Editor is open.

**.IsGlobal** Returns 1 (True) if the palette is global or 0 (False) if it is application-specific. (This is the same as the "Global" checkbox in the Palette Editor.) You can set .IsGlobal to 0 or 1 to make the palette application-specific for the active application (0) or global (1).

If you change the .IsGlobal property of the palette containing the script, the palette is temporarily unloaded from memory and script execution stops.

```
// Change the palette from global to app-specific or vice-versa.
// Same as clicking the Palette Editor's Global checkbox.
Palette.IsGlobal = NOT Palette.IsGlobal
// (Changes to 0 if set to 1, or changes to 1 if set to 0.)
Message "This line never executes because the palette gets unloaded from memory."
```

**.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
```

**.MainScreen** On multiple-monitor systems, returns the number of the screen on which the palette appears. Returns 1 on single-monitor systems.

**.Name** Returns or sets the the name of the specified palette.

```
// Display the name of the palette containing the button's script
Message Palette.Name

// Change the name of the palette named "Average Buttons" to "Super Buttons"
Palette("Average Buttons").Name = "Super Buttons"
```

**.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:

Palette(*palette-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
```

### .PICT

The .PICT property lets you change a palette's background PICT from within a script. PICT resources can be stored in the palette file or in another resource file. Use the format:

Palette(*specifier*).PICT = *resourceID* [, *file-path*]

Specify a the picture's resource ID. For example, if a file contains 10 PICT resources including IDs 128, 129, 140, 143, use one of those numbers to specify the PICT resource. You can see which resources exist in the file with ResEdit. Specify 0 (zero) to clear the background PICT.

If no file is specified, the PICT is loaded from the palette file. These "embedded" PICTs must have resource IDs in the range 1–256. This tells OneClick to not delete the PICT resource when the background PICT is changed.

```
// Set the background image to Apple Guide's alpha slider
Palette.PICT = 303, (FindFolder "extn") & "Apple Guide"
Palette.PICT = 5       // Internal PICT stored in the palette file
Palette.PICT = 0       // Clear PICT
```

**Note** OneClick 1.0.3 and earlier used resource index numbers instead of resource IDs for background PICTs. In OneClick 2.0, scripts that set the background PICT may need to be changed if the PICT's resource ID does not equal its index number (order in the resource file).

### .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
```

## PaletteMenu command

**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 command

**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.

The Pause command allows other processing to occur while the script is interrupted. Background processing resumes and you can interact with the system during the pause. It works the same as using the Wait command to wait for a certain number of ticks.

**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 command (page 303)

## PopupFiles function

**Syntax** PopupFiles [*folder*] [, *file-type-list*] [, *suppress-parent-folders*]

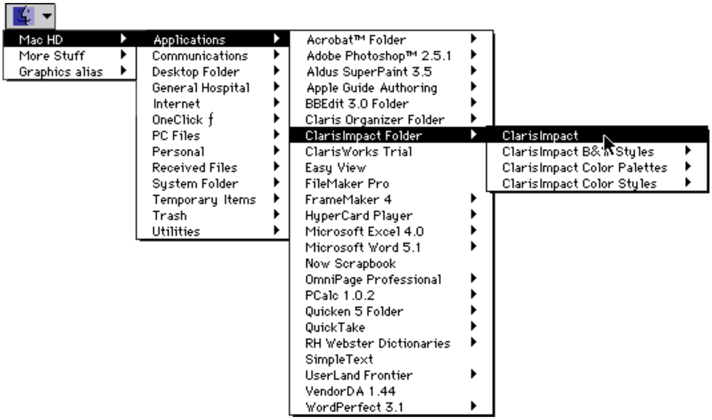
**Description** Pops up a hierarchical menu of all the files, folders, and volumes on the desktop and returns the full path to the chosen file or folder.

*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. If you pass 1 (True) in the *suppress-parent-folders* parameter, then the menu won't list any higher-level folders and volumes.

*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<return>WDBN"

// Choose a file or folder from any volume and open it
Open PopupFiles
```



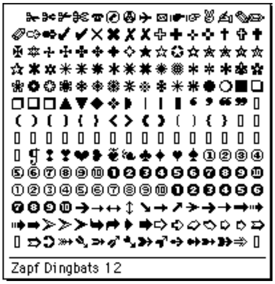
Sample PopupFiles menu

**See Also** PopupMenu function (page 265)

## PopupFont function

**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 function

**Syntax** PopupMenu *menu-list* [, *checked-item-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.

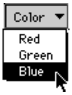
To include a divider line in the menu, include a hyphen (-) as an item in the menu list.

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 (~).

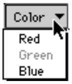
The *checked-item-list* parameter is a list of one or more items to appear checked in the menu. Checked items can be passed as a list of menu items in one parameter or as multiple separate parameters. When passed separately, the *checked-item* parameters can indicate either a menu item's text or a numeric position in the menu.

## Examples

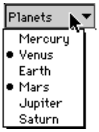
```
// Types the selected item from the Red, Green, Blue menu.
Type PopupMenu "Red<return>Green<return>Blue"
```



```
// Types the selected item from Red and Blue. Green is disabled.
Type PopupMenu "Red<return>~Green<return>Blue"
```



```
// Choose a planet from the pop-up menu. Venus is checked.
Variable theChoice
theChoice = PopupMenu "Mercury<return>Venus<return>Earth", "Venus"
```



```
// Choose a planet from the pop-up menu. Venus (item 2) and Mars (item 4) are checked.
Variable theChoice
theChoice = PopupMenu "Mercury<return>Venus<return>Earth<return>Mars", 2, 4

// Shows a pop-up menu of the Font menu.
SelectMenu "Font", PopupMenu Menu("Font").List
```

## See Also

PopupFiles function (page 264)

# PopupMenuFont system variable

**Description** Gets or sets the font used in all OneClick pop-up menus. You can specify the font by its name or ID number. The font resets to the default font (Geneva) after the computer starts up.

**Example**

```
// Set the OneClick pop-up font and size to Charcoal 12 at startup
On Startup
    PopupMenuFont = "Charcoal"
    PopupMenuSize = 12
End Startup

// Save the current pop-up menu font and size, then display a menu
Variable oldFont oldSize
oldFont = PopupMenuFont
oldSize = PopupMenuSize
PopupMenuFont = "Helvetica"
PopupMenuSize = 24
Open PopupFiles
// Restore the previous font and size when the menu goes away
PopupMenuFont = oldFont
PopupMenuSize = oldSize
```

**See Also** EditorFont system variable (page 195), EditorSize system variable (page 195), PopupMenuSize system variable (page 267)

# PopupMenuSize system variable

**Description** Gets or sets the font size (in points) used in all OneClick pop-up menus. The size resets to the default size (9-point) after the computer starts up.

**Example**

```
PopupMenuFont = "Espy Sans"
PopupMenuSize = 9
```

**See Also** EditorFont system variable (page 195), EditorSize system variable (page 195), PopupMenuFont system variable (page 267)

# PopupPalette command

**Syntax** PopupPalette *palette-name* [, *no-tear-off*]

**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. If you pass 1 (True) in the *no-tear-off* parameter, the palette pops up but can't be torn off.

**Examples**

```
PopupPalette "Launcher"
PopupPalette "Calendar", 1      // pops up the palette but prevents it from being torn off
```

**See Also** PopupMenu function (page 265)

## PrintText command

**Syntax** PrintText *text* [, *format-string*]

**Description** Prints text to the printer. You can specify the font and font size; top, left, bottom, and right margins; begin and end pages; and whether or not to display the Page Setup and Print dialog boxes.

*Format-string* can include any of the following formatting properties.

| Property | Description | Default value |
|---|---|---|
| Font: | Font ID or font name | 3, Geneva |
| Size: | Font size | 10 point |
| From: | Start page | 1 |
| To: | Last page | last page |
| Top: | Top margin | 72 (1 inch) |
| Left: | Left margin | 72 (1 inch) |
| Bottom: | Bottom margin | 72 (1 inch) |
| Right: | Right margin | 72 (1 inch) |
| Page Setup | Displays Page Setup dialog box | |
| Print Dialog | Displays Print dialog box | |

If a property is not specified, PrintText uses the default value. The Page Setup and Print Dialog properties have no default values; the dialog boxes appear only if you include the properties in the format string.

Margins are indicated in points (72 dots per inch).

The font name must be enclosed within curly quotes, such as “Helvetica”.

**Examples**

```
// Print using the Page Setup and Print dialog boxes
PrintText "This text will be printed", "Top: 36, Bottom: 144, Page Setup, Print Dialog"

// Print the first page of the file using Courier 12
PrintText File("Macintosh HD:Read Me").Text, "To: 1, Font: “Courier”, Size: 12"

// Print the Clipboard contents in Geneva 10 with default 1-inch margins
PrintText Clipboard
```

# Process object

**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.

You can specify a Process object three ways:

- by name, as the name appears in the Application menu in the menu bar
- by the application's creator code
- by number

If you don’t specify an application, the Process object uses the frontmost application.

## By name

Specify the Process object using the application's name as it appears in the Application menu.

You can use wildcards when specifying processes by name. Use an asterisk (*) to match multiple characters or a question mark (?) to match single characters. For example, Process("Adobe Photoshop*") matches “Adobe Photoshop™ 2.5.1”, “Adobe Photoshop 3.0”, and so on.

Specifying a process by its name is generally not a good idea. For example, if you specify Process("WordPerfect™ 9.2.16"), it isn’t going to work with all the other versions of WordPerfect. Also, names of some system applications (such as Finder) may be different in other languages.

## By number

When specifying a process by number, Process(1) is the active application, Process(2) is the previous application used, and so on. Process(0) is a shortcut for referencing the Finder.

The Process object can access background-only applications which don't appear in the application menu (such as File Sharing Extension, Express Modem, and so on). Like visible processes, you can specify them by name or creator. You can also refer to them by a negative number (–1, –2, and so on). To loop through all background applications, start with –1 and check Process.Exists, decrementing the specifier number each time until Process.Exists returns 0 (False). Background-only processes do not appear in Process.List and are not counted in Process.Count.

## By creator code

When specifying a process by creator code, use the format:

```
Process("[type]:creator")
```

The four-character type code is optional; the colon (:) and creator code are required.

**Note** An easy way to find a file's creator is to run Find File, choose "creator" from the pop-up menu, and drag the file from the Finder to the text box area. You can also use ResEdit to find a file's creator.

```
Process("APPL:MSWD")   //  Finds application "Microsoft Word"
Process(":MSWD")   //  Finds anything (application, DA, etc.) with creator 'MSWD'
Process("FNDR:MACS")   // Finds Finder
```

All Process properties (except for .Selection and .Visible) are read-only.

**.Count** Returns the total number of open, running processes. The .Count property is a shortcut for ListCount Process.List.

**.Creator** Returns the four-character creator code for the specified process. For example, if SimpleText is the active application, Process.Creator returns "ttxt".

**.Exists** Returns True (1) if the specified process 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 process.

```
// Get the path to the folder containing the SimpleText application
Variable theAppPath
theAppPath = Process("SimpleText").Folder
// Alternate method of getting the path to the folder containing SimpleText
theAppPath = Process(":ttxt").Folder
```

**.Free** Returns the amount of free memory (in bytes) in the specified process' 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** Brings the specified process to the front (makes it active).

To get the name of the active application, use Process.Name.

```
// Switch to the Finder
If Process.Name <> Process(0).Name      // Process(0) means Finder
    Process(0).Front
End If

// Switch back and forth between the two frontmost applications
Process(2).Front
```

**.Index** Returns the corresponding index number for the process when the process is specified by name or creator code. Processes are indexed in front-to-back order using 1 for the first process (the active application), 2 for the second, and so on.

Faceless background processes, such as File Sharing Extension and Express Modem, have negative index numbers beginning with –1 for the first background process, –2 for the second, and so on.

**.Kind** Returns the four-character file type code for the specified process. 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 processes. This list includes only applications that appear in the Application menu. Background-only processes (such as File Sharing) aren't included.

**.Name** Returns the name of the specified process. The .Name property is useful if you want to get the name of a process specified by number.

To make bring an inactive application to the front, use Process.Front.

```
// 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 process as if you had chosen Quit from the application's File menu. The .Quit message sends an Apple Event to the specified process, 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 process. 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 Mac OS 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..."
```

**.SendAE** Sends an Apple event to the specified process. If the event returns a response, you can get the response by assigning the SendAE statement to a variable or by using the SendAE statement in an expression. Sending Apple events with SendAE is much faster than sending the equivalent events with AppleScript.

OneClick does not directly send Apple events to an application; instead, it causes the application to send events to itself. If the application can receive events but does not support sending events, then the Finder sends the events.

The event to be sent must be formatted as a string in AEBuild format. This format may look complicated in the examples below, but in most cases you don't need to write these strings yourself—the Capture AE control panel can record SendAE statements for you. For more information on Capture AE, SendAE, and the AEBuild string format, see the online document *Using Capture AE and SendAE*.

```
// Tell the active web browser to open WestCode Software's home page
// This works with any web browser that supports the standard GetURL event
Process.SendAE "GURL,GURL,'----':“http://www.westcodesoft.com”"

// Tell the active web browser to open the URL on the Clipboard
Process.SendAE "GURL,GURL,'----':“" & Clipboard & "”"

// Tell America Online 2.7 to go to a keyword (may not work with AOL 3.0)
Variable keyword
keyword = "MUT" // Macintosh Utilities Forum
Process("America Online").SendAE "AOae,KWRD,'----':“" & keyword & "”"

// Tell BBEdit 4.0 to return the number of open windows
Variable numWindows
numWindows = Process("BBEdit 4.0").SendAE "core,cnte,'----':'null'(), kocl:type(cwin)"
Message numWindows
```

### .Size

Returns the total amount of memory (in bytes) allocated to the specified process. 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 process 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
```

**.Window** Represents the Window objects of the specified process. Use the Process(*specifier*).Window(*specifier*) notation to access windows in applications other than the active application.

When you access a Window object as a property of a Process object, you can get any window property and set most properties. You cannot set properties that require a click on the window, such as sizing, zooming, windowshading.

```
// Move the second window of the second application to 35 pixels below the top of the screen
Process(2).Window(2).Top = 35

// Bring the selected Finder window to the front, no matter what application is active
Variable winList, theChoice
winList = Process("Finder").Window.List
theChoice = PopupMenu winList
If theChoice
	Process("Finder").Front
	If theChoice <> "Desktop"// don’t try to make the Desktop active; weird things happen!
		Window(theChoice).Front
	End If
End If
```

# Proper function

**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 function (page 238), Upper function (page 299)

# QuicKey command

**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"
```

# QuoteText function

**Syntax** QuoteText text [, *line_length* [, *quote_char*]]

**Description** Puts a ">" at the beginning of every line and a <return> at the end of each line and returns the quoted text.

You can optionally specify the maximum number of characters on each line (the default is 74). To prevent QuoteText from word-wrapping text, specify a line length equal to or larger than the text length.

If you specify a value for *quote_char* it will be used instead of ">". *Quote_char* may consist of any number of characters.

**Examples**

```
Clipboard = QuoteText Clipboard
SelectMenu "Edit","Paste"

// Save the Clipboard contents
Variable oldText
oldText = Clipboard
Type Command "c"
ConvertClip
// Add "### " to the beginning of each line without automatic word-wrap
Clipboard = QuoteText Clipboard, Length Clipboard, "### "
ConvertClip 1
Type Command "v"
// Restore previous Clipboard contents
```

```
Clipboard = oldText
ConvertClip 1
```

## Random function

**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 negative, Random returns a negative number.

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

// Types 100 negative random numbers between –2,147,483,647 and 1
Repeat 100
      Type Random –2147483647, Return
End Repeat
```

## Repeat, Next Repeat, Exit Repeat, End Repeat commands

**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 "–"
```

```
End Repeat

// 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 commands (page 219)

## Replace function

**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 function (page 212)

## Return function

**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 function (page 295)

## Schedule command

**Syntax**

Schedule *tenths* [, *run-always*]
Schedule *schedule-type* [, *run-always*]

**Description** Causes the script to automatically run its Scheduled handler every *tenths* tenths of a second, or when the event specified by *schedule-type* occurs. Use Schedule 0 to turn off all of a script's schedules.

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. If you are doing a timed schedule, you must set it first, followed by any of the event schedules.

### Scheduling for certain event types

In addition to having a script's Scheduled handler called at regular intervals, you can have the handler called when any of the following events occur.

| Schedule Number | Type | Description |
|---|---|---|
| –1 | Process start | Runs when an application starts up. Process.Name contains the name of the application that just started. |
| –2 | Process quit | Runs after an application has quit. The application that quit is no longer in Process.List when this handler runs. |
| –3 | Process switch | Runs when you switch from one application to another. Process.Name contains the name of the application after the switch. |

| Schedule Number | Type | Description |
|---|---|---|
| –4 | Window switch | Runs when you switch from one window to another in the active application. Window.Name contains the name of the window after the switch. Window switch also occurs if you close a window and there are no more open windows or if you open the first window. The schedule does not run when switching between windows in two different applications if the window in the new application does not change. |
| –5 | Menu bar redraw | Runs any time the menu bar is redrawn. This is usually done whenever the application adds, removes, enables, or disables menus. Generally it is a signal of a major mode change in the application. |
| –6 | Mouse enter/exit | Runs when the cursor moves over a different OneClick button or when the cursor moves off a button. |
| –7 | Palette show/hide | Runs when a palette is going to change state between visible and invisible. The script's Scheduled handler is called before the palette changes state, so the Palette.Visible flag indicates the status before the change. Only buttons that are on the palette that is changing will receive this scheduled event. |
| –8 | Cursor change | Runs whenever the cursor changes (from a watch to the arrow, for example). |
| –9 | Monitor change | Runs whenever a monitor changes resolution or position. For example, the handler runs if you change the monitor from 640x480 to 1024x768, or if you change monitor positions in the Monitors & Sound control panel. |
| –10 | Editor closed | Runs whenever the OneClick Editor window has closed. |

Use the Schedule command with one of the specified negative numbers to set up an event schedule. For example:

```
Schedule 30  // Call Scheduled handler every 3 seconds (timed schedule)
Schedule –1  // Call Scheduled handler every time a process starts
Schedule –4 // Call Scheduled handler every time a different window is in front (not
called on process switch)
```

## 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

// Play sounds whenever applications start or quit
On Startup
	Schedule - 1	// process start
	Schedule - 2	// process quit
End Startup

On Scheduled
	If ScheduleType = - 1
		Sound "AppOpen"
	Else If  ScheduleType = - 2
		Sound "AppClose"
	End If
End Scheduled
```

**See Also** Scheduled handler (page 281), ScheduleType system variable (page 282), Startup handler (page 294)

# Scheduled handler

**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 136), Schedule command (page 279), Startup handler (page 294)

## ScheduleType system variable

**Description** Returns the type of scheduled task for which the handler is being called. See the Schedule command (page 279) for a list of schedule types and what they’re used for.

**Example**

```
On Scheduled
    If ScheduleType = 0
```

```
        // Timed schedule
    Else If ScheduleType = –3
        // Process switch
    End If
End Scheduled
```

**See Also** Schedule command (page 279)

## Screen object

**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 & Sound control panel.

**.Color** Gets or sets the Colors and Grays options in the Monitors & Sound 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
```

**.CursorScreen** Returns a number indicating which screen the cursor is on.

**.CursorX** Sets or returns the current global horizontal position of the cursor.

```
// Move the cursor to 40 pixels from the left, 80 from the top
Screen.CursorX = 40
Screen.CursorY = 80
```

**.CursorY** Sets or returns the current global vertical position of the cursor.

**.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 & Sound control panel; your system may not support all the bit depths listed here.

| .Depth value | 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.

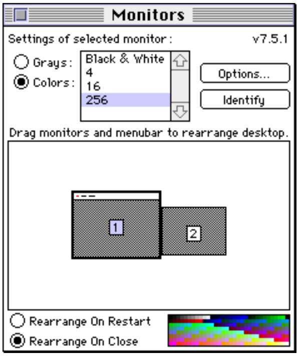
```
// Display a message if only one monitor is connected
If NOT Screen(2).Exists
      Message "Can’t find a second monitor"
End If
```

**.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



Settings of selected monitor:

○ Grays:
● Colors:

Black & White
4
16
256

Drag monitors and menubar to rearrange desktop.

○ Rearrange On Restart
● Rearrange On Close

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.

**Note** Depending on your computer model and system software version, your Monitors control panel may look different than the picture above.

```
// 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
```

**.Maximum** Returns the maximum bit depth supported by the specified screen. The .Maximum property returns its value as a number of bits (1, 2, 4, 8, 16, or 32). See the .Depth property for a table of bit depth values and their meanings.

**.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
```

## ScriptInterrupts system variable

**Description** Stops or restarts the script's ability to be interrupted during execution.

The ScriptInterrupts system variable is normally True (1), which means that scripts that take a while to execute will get interrupted occasionally to allow other system processes to run. Setting it to False (0) allow the script to run uninterrupted. This may be necessary in scripts where the script environment can't be allowed to change by other scripts, the user, or an application.

Normally, script interrupts shouldn't be left off for extended periods of time.

It is not necessary to restore ScriptInterrupts to True at the end of the script. It is automatically restored.

**Example**

```
// Turn off script interrupts
ScriptInterrupts = 0
```

## Scroll command

**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 command

**Syntax**

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

**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.

You can also specify dialog box buttons by number. Specifying by number is necessary when buttons have duplicate names. To determine the number of a dialog box button, use the Button item in the Parameters pop-up menu in the Script Editor. The first button is 1, the second is 2, and so on.

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 object (page 187)

## SelectMenu command

**Syntax** SelectMenu [Command] [Option] [Control] [Shift] [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.

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

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 command (page 290), Menu object (page 240)

## SelectPopUp command

**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 command (page 288)

## Set command

**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 command (page 300)

## SetICPref command

**Syntax** SetICPref *preferenceKey*, *value*

**Description** Assigns a value to the specified preference in Internet Preferences. Only simple text string and file path preferences can be set; complex preferences (such as font settings, color settings, extension mappings, and so on) cannot be set.

For a list of valid preference keys supported by Internet Config 2.0, see "GetICPref function" on page 222.

**Examples**

```
SetICPref "RealName", "Cole Deschanel"
SetICPref "DownloadFolder", "Mac HD:Downloads:"
```

**See Also** GetICPref function (page 222), GetICHelpers function (page 222)

**Author Info** SetICPref, part of IC Extension
Copyright © 1998 Life OnLine Software (lr). All rights reserved.

## SetScrap command

**Syntax** SetScrap *data*, *resourceType*

**Description** Puts data on the Clipboard as the specified resource type. Use GetScrap and SetScrap instead of the Clipboard system variable when you want to work with data types other than plain text.

**Examples**

```
SetScrap "just some words", "TEXT"
SetScrap pictData, "PICT"

// Get a TEXT/styl (styled text) resource pair from a file and put the styled text on the Clipboard
Variable theText, theStyle, theFile
theFile = "Mac HD:Desktop Folder:My Resource File"
theText = GetResource theFile, "TEXT", 128
theStyle = GetResource theFile, "styl", 128
SetScrap theText, "TEXT"
SetScrap theStyle, "styl"
```

**Author Info** SetScrap, part of Scrap Extension
Copyright © 1999 Life OnLine Software (lr). All rights reserved.

## ShiftKey system variable

**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, X
netFolder = "Mac HD:Internet:"
theAppList = "Eudora<return>NewsWatcher<return>Netscape"
If ShiftKey
    For X = 1 to ListCount theAppList
        Open netFolder & (ListItems theAppList, X)
    End For
Else
    Open netFolder & (PopupMenu theAppList)
End If
```

**See Also** CommandKey system variable (page 177), ControlKey system variable (page 179), OptionKey system variable (page 255)

## Sound command

**Syntax** Sound [*sound-name*]

**Description** Plays the specified sound, or a random sound if no sound name is given. 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
Sound "Quack"
```

**See Also** Beep command (page 157)

## SoundLevel system variable

**Description** Returns the current system sound level (volume), or sets the sound level to a new value. It's the same as adjusting the Computer System Volume on the Sound panel of the Monitors & Sound control panel.

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 system variable (page 158), Beep command (page 157), Sound command (page 292)

## Speak command

**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 Mac OS 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"
```

## Startup handler

**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
```

## SubString function

**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 characters from *start* to the last character in the string. 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: ello
Type SubString "Hello", 2

// Types: SS
Type SubString "Annuities (SS)", –3, –2
```

**See Also** ListItems function (page 236)

## SystemFolder system variable

**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 function (page 216)

## SysVersion system variable

**Description** Returns the Mac OS version as a whole number. Version 7.5 returns 750, 8.5.1 returns 851, and so on. You cannot set this system variable.

**Example**

```
// Display a message and exit if we’re running a version of Mac OS earlier than 8.1.
If SysVersion < 810
    Message "You need Mac OS 8.1 or later to use this palette."
    Exit
End If
```

**See Also** Version system variable (page 301)

## Tab function

**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 function (page 278)

## TextWidth function

**Syntax** TextWidth *text* [, *fontname*] [, *size*] [, *style*]

**Description** Returns the width of a line of *text* (in pixels) using the specified *fontname*, *size*, and *style* number. If you omit any of the optional parameters, TextWidth uses default values of Geneva, 9-point, plain style in place of any missing parameters.

The *style* parameter is a number you can specify yourself or get from a button's .TextStyle property. Add style numbers together to combine styles.

| | | |
|---|---|---|
| 0: Plain Text | 1: Bold | 2: Italic |
| 4: Underline | 8: Outline | 16: Shadow |
| 32: Condense | 64: Extend | |

**Examples**

```
// Get the width of the text in Palatino 12-point bold, italic
Variable W
W = TextWidth "sample text to measure", "Palatino", 12, 3

// Get the width of the text in Geneva 9-point plain
W = TextWidth "sample text to measure"

// Set the button's width to the width of its text.
// Allow extra room for the button's border width, which can be
// anywhere from 3 to 17 pixels depending on the border style.
Variable Extra
Extra = 7   // left + right border widths
With Button
        .Border = 1
        .Width = Extra + TextWidth .Text, .TextFont, .TextSize, .TextStyle
End With
```

**Author Info** TextWidth, part of FitText Extension
Copyright © 1999 Jeff Jungblut. All rights reserved.

**See Also** TruncText function (page 297), Button object (page 158)

## Ticks system variable

**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 command (page 263), Wait command (page 303)

## Trim function

**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  "
```

## True system variable

**Description** Returns the number 1.

**See Also** False system variable (page 199)

## TruncText function

**Syntax** TruncText *width*, *where*, *text* [, *fontname*] [, *size*] [, *style*]

**Description** Truncates a line of *text* in the given *fontname*, *size*, and *style* so that it fits within *width* pixels and returns the truncated text, or the original text if no truncation is necessary.

*Where* indicates where the truncation should occur: 0 (zero) for the end of the text or 1 for the middle. TruncText places an ellipsis character (…) at the point where the truncation occurred, if any.

If you omit any of the optional parameters, TruncText uses default values of Geneva, 9-point, plain style in place of any missing parameters.

The *style* parameter is a number you can specify yourself or get from a button's .TextStyle property. Add style numbers together to combine styles.

| | | |
|---|---|---|
| 0: Plain Text | 1: Bold | 2: Italic |
| 4: Underline | 8: Outline | 16: Shadow |
| 32: Condense | 64: Extend | |

### Examples

```
// Truncate a button's text so that it fits within the button's width.
// Allow extra room for the button's border width, which can be
// anywhere from 3 to 17 pixels depending on the border style.
Variable Extra
Extra = 7    // left + right border widths
With Button
    .Border = 1
    .Text = TruncText .Width - Extra, 0, .Text, .TextFont, .TextSize, .TextStyle
End With
```

### Author Info

TruncText, part of FitText Extension
Copyright © 1999 Jeff Jungblut. All rights reserved.

### See Also

TextWidth function (page 296), Button object (page 158)

## Type command

### 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 11/9/99, 5:00 PM
Type Date, ", ", Time

// types Command-B
Type Command "b"

// type Command-S, then Command-P
Type Command "sp"
```

**See Also** KeyPress command (page 231)

## Upper function

**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 function (page 238), Proper function (page 275)

## UserHandler1 ... UserHandler5 handlers

**Description** UserHandler1 through UserHandler5 are five handlers you can use as subroutines—containers for common code used by more than one button. In cases where you write a single group of statements that needs to be included in several different buttons, instead of repeating the statements in each button, you can place the statements in a UserHandler in one button and have the other buttons Call that handler.

**Examples**

```
Call UserHandler1, "Common" // Executes statements in UserHandler1 on button “Common”

// This script is in a button named “Routines”
On UserHandler4
    // This is an error-handling subroutine that plays the sound named in g_theErrSound,
    // displays the error text in a message box, and appends the error text and the time
    // the error occurred to a log file whose path is in g_theErrLog.
    // When the handler is finished, it returns to the script that called it.
    Variable Global g_theErrSound, g_theErrMessage, g_theErrLog
```

```
          Sound g_theErrSound
          Message g_theErrMessage
          If g_theErrLog // path of error log (a text file)
                    File(g_theErrLog).Append = (Time 1) & " " & g_theErrMessage & Return
          End If
End UserHandler4

// This script is in a button named “Check Files” and calls the UserHandler4 handler in “Routines”
Variable Global g_theErrSound, g_theErrMessage, g_theErrLog
g_theErrSound = "Indigo"
g_theErrLog = (FindFolder "desk") & "Sample Error Log"
// Check to see if Drag and Drop is available. If not, call our error routine -- UserHanlder4.
If NOT (Gestalt "drag", 0)
          g_theErrMessage = "Drag and Drop isn't available!"
          Call UserHandler4, "Routines"
End If
// Check to see if the file "Rosario" exists on the desktop. If not, call our error routine.
If NOT File((FindFolder "desk") & "Rosario").Exists
          g_theErrMessage = "Can’t find the Rosario file!"
          Call UserHandler4, "Routines"
End If
```

**See Also** Call command (page 171), Calling scripts as subroutines (page 128), Calling scripts as functions (page 128)

## Variable command

**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 94)

## Version system variable

**Description** Returns the OneClick version as a whole number. Version 1.0 returns 100, 1.0.3 returns 103, 2.0 returns 200, and so on. You cannot set this system variable.

**Example**

```
// Display a message and exit if we’re running a version of OneClick earlier than 1.5.
If Version < 200
      Message "You need OneClick version 2.0 or later to use this palette."
      Exit
End If
```

**See Also** Implemented function (page 229), SysVersion system variable (page 295)

## Volume object

**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.

Volume.Size and Volume.Free do not support volumes larger than 2 GB. For volumes larger than 2 GB, .Size returns 2097138 (approximately 2 GB).

```
// 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
```

**.Index** Returns the corresponding index number for the volume when the volume is specified by name. Volumes are indexed in the order they were mounted (1 for the first volume mounted, 2 for the second, and so on).

**.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.

Volume.Size and Volume.Free do not support volumes larger than 2 GB. For volumes larger than 2 GB, .Size returns 2097138 (approximately 2 GB).

```
// 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
```

## Wait command

**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 a window named “Welcome” appears or until 10 seconds are up, whichever is first
Variable startTicks timeoutSecs windowName
startTicks = Ticks
timeoutSecs = 10
windowName = "Welcome"
Wait NOT (((Ticks - startTicks) < (timeoutSecs * 60)) AND (NOT Window(windowName).Exists))

// Waits until the time is 12:15 PM, then plays a sound
Wait (Time 0) = "12:15 PM"
Sound "Quack"
```

**See Also** Pause command (page 263)

## WeekdayGregorian function

**Syntax**

WeekdayGregorian *year*, *month*, *day*

**Description**

Returns the day of the week (1=Sunday, 2=Monday, ... 7=Saturday) for the specified *year*, *month*, and *day*. *Year* should be four digits; 99 is not interpreted as 1999.

The DateTime.Weekday property returns the correct day of week only for dates supported by the Mac OS date format (January 1, 1904 through February 6, 2040). WeekdayGregorian returns the correct day of week for any date, either B.C. or A.D.

**Examples**

```
Variable dayList day
dayList = "Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday"
ListDelimiter = ","
day = ListItems dayList, (WeekdayGregorian 1999, 4, 8)
Message "April 8, 1999 is a " & day
```

**Author Info**

WeekdayGregorian, part of Calendar Extension
Copyright © 1999 Jeff Jungblut. All rights reserved.

## While, Next While, Exit While, End While commands

**Syntax**

```
While condition
    statements
    [Next While]
    [Exit While]
End While
```

**Description**

Executes all of the statements between While and End While 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
X = 1
```

```
While X <= 10
        Type X & ", "
        X = X + 1
End While

// Type 1 to 10 separated by commas, skipping number 5
X = 1
While X <= 10
        If X = 5
                Next While
        End If
        Type X & ", "
        X = X + 1
End While
```

**See Also** Repeat, Next Repeat, Exit Repeat, End Repeat commands (page 277); For, Next For, Exit For, End For commands (page 219); Repeating statements while a condition is true (page 106)

# Window object

**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.

To work with windows in an inactive application, specify the process along with the window.

Process(*specifier*).Window(*specifier*).Property

When you access a Window object as a property of a Process object, you can get any window property and set most properties. You cannot set properties that require a click on the window, such as sizing, zooming, windowshading. To set properties that require a click, use Process(*specifier*).Front first to make the application active, then change the window properties by accessing the Window object without the Process specifier.

**.Collapsed** Gets or sets the windowshade status of the specified window. If the window is collapsed (shaded), the value is 1. When the window is expanded, the value is 0 (zero).

The .Collapsed property is only available in Mac OS 8.0 and later.

```
// Collapse or expand the active window.
Window.Collapsed = NOT Window.Collapsed

// Toggle the windowshade for all windows in the active application.
Variable X
For X = 1 To Window.Count
      // If it's collapsed, expand it; if it's expanded, collapse it
      Window(X).Collapsed = NOT Window(X).Collapsed
End For
```

**.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** Brings the specified window to the front (makes it active). Use Window.Front to switch windows from a script instead of using the Click command to click a window and make it active.

.Front works as both a message and a property; that is, Window(2).Front is the same as Window.Front = 2.

```
// Switch to the "Bookmarks" window if it’s not already active
If Window.Name <> "Bookmarks"
      Window("Bookmarks").Front
End If

// Cycle through all open windows. Don’t try to make the Desktop active or Finder may hang.
If Window(Window.Count).Name <> "Desktop"
```

```
        Window.Front = Window.Count
Else
        Window.Front = Window.Count - 1
End If
```

**.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
```

**.Index** Returns the corresponding index number for the window when the window is specified by name. Windows are indexed in front-to-back order (1 for the first window, 2 for the second, and so on).

**.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.

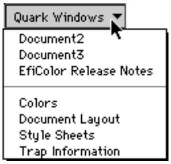
Use the .Kind property to determine which windows in Window.List are document windows and which are other special types of windows. By evaluating the .Kind property of each window in the list, you can create a new list that contains just the desired windows.

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 |

| Value | Window style |
|---|---|
| 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 write-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
```

**.Name** Returns the name of the window specified by index number. When used without a specifier, Window.Name returns the name of the active window.

To bring an inactive window to the front, use Window.Front.

**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.Name 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 tool bars or other kinds of windows.

**.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
```

**.TitleBar** Returns the height in pixels of the specified window's title bar.

**.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.

An alternative to hiding windows with Window.Visible=0 is to move the window off screen. It accomplishes the same thing as making it invisible, but you can still interact with the window by sending clicks and keystrokes to it. This is basically what Dialogs = 0 does.

One drawback is that you need to remember the original position of the window to restore it. A solution is to add a specific constant to move it off screen (say 8000, for example), then subtract that same constant when moving the window back on screen.

**.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
```

# With command

**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.

**Note** Don't use a For, Repeat, or While loop inside a With statement to iterate through several objects or properties; unpredictable results may occur. You can put the With statement inside the For, Repeat, or While loop to get the desired results.

**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

// Clear the text, color, and keyboard shortcut for every button on the palette
Variable X
For X = 1 To Button.Count
        With Button(X)
                .Text = ""
                .KeyShortCut = ""
                .Color = 0
        End With
End For
```

**See Also** Objects (page 109)

## 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 8, "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 command | Indicates that the following lines are AppleScript statements. |
| Beep command | Plays the Macintosh system beep. |
| Call command | Calls another script or handler as a subroutine of the current script. |
| Click command | Simulates clicking coordinates on the screen or within a window or dialog box. |
| CloseWindow command | Closes the active window. |
| ConvertClip command | Forces conversion of the Clipboard contents to plain text. |
| Dial command * | Dials a phone number through the speaker or a modem. |
| DragButton command | Drags the specified text from a button. |
| DrawIndicator command | Draws a progress bar or pie indicator. |
| Editor command | Opens the OneClick Editor window and displays the specified tab. |
| 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 command | Copies files to the specified folder. |
| FinderMove command | Moves files to the specified folder. |
| For | Repeats one or more statements while incrementing the specified variable. |
| GetWindowText command | Captures text from the specified window and puts it in a variable. |
| If | Indicates statements to execute if a logical expression evaluates to True. |
| KeyPress command | Simulates a key press of the specified key. |
| LoadExtensions command | Loads or reloads OneClick extensions from the specified file. |
| Message command | Displays a message in a dialog box with an OK button. |
| Next | Forces the next iteration of a For, Repeat, or While loop. |
| Notify command | Displays a message in a notification dialog box or floating window. |
| On | Specifies the start of a handler. |
| Open command | Opens the specified application, document, or folder. |
| PaletteMenu command | Displays the OneClick menu as a pop-up menu. |

| Command | What it does |
|---|---|
| Pause command | Pauses for the specified time interval, then resumes. |
| PopupPalette command | Displays another palette as a pop-up palette. |
| QuicKey command * | Runs the specified QuicKey macro. Requires QuicKeys™ from CE Software. |
| Repeat | Repeats one or more statements the specified number of times. |
| Schedule command | Specifies when or how often the script's Scheduled handler should run. |
| Scroll command | Simulates clicks in the active window's scroll bars. |
| SelectButton command | Clicks a named button or checkbox within a window or dialog box. |
| SelectMenu command | Simulates choosing a command from a pull-down menu. |
| SelectPopUp command | Simulates choosing a command from a pop-up menu in a window or dialog box. |
| Set command | Assigns a value to a variable. |
| Sound command | Plays the specified sound. |
| Speak command * | Speaks the specified text. |
| Type command | Types the specified text or command keys. |
| Variable command | Declares variable names for use in a script. |
| Wait command | Waits until the specified condition evaluates to True, then resumes. |
| While | Repeats one or more statements while the specified condition is true. |
| With command | 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 function | Returns the absolute value of a number. |
| Alias function | Returns an alias for the specified file as a string. |
| AskButton function | Displays an alert box and returns a value indicating which button was used to dismiss the alert. |
| AskFile function | Displays a directory dialog box and returns the full path of the file or folder selected. |
| AskKey function | Prompts you to press a key combination and returns the keystroke in text format. |

| Function | What it does |
|---|---|
| AskList function | Displays a list box and returns the selected item. |
| AskShortcut function | Reserved for WestCode use. |
| AskText function | Displays a dialog box with a text field and returns the text typed in the field. |
| Char function | Returns the character indicated by the ASCII code parameter. |
| Code function | Returns the ASCII code of the string parameter's first character. |
| Find function | Returns the character position of one text string within another. |
| FindApp function | Returns the path to the application with the specified creator code. |
| FindFolder function | Returns the full path of the specified folder. |
| Gestalt function | Returns Gestalt information for the specified selector. |
| GetDragAndDrop function | Returns a list of paths of dropped files or returns the text in a dropped text clipping. |
| GetResources function | Returns a list of all the resources of the specified type. |
| Implemented function | Returns True if the specified OneClick keyword is available. |
| Length function | Returns the number of characters in a text value. |
| ListCount function | Returns the number of items in a list. |
| ListDelete function | Deletes items from a list and returns the new list. |
| ListFind function | Returns the position (index) of an item in a list. |
| ListInsert function | Inserts new items into a list and returns the new list. |
| ListItems function | Returns a subset of items from the specified list. |
| ListSort function | Sorts the specified list and returns the sorted list. |
| ListSum function | Returns the sum of all numbers in a list. |
| Lower function | Returns a text value as all lowercase letters. |
| MakeNumber function | Converts a text value to a number and returns the result. |
| MakeText function | Converts a number to a text value and returns the result. |
| MenuNumber function | Returns the item number of the last item selected with the PopupMenu function. |
| OldListCount function | Obsolete function. Use ListCount instead. |
| OldListItems function | Obsolete function. Use ListItems instead. |

| Function | What it does |
|---|---|
| OpenFileList function | Returns a list containing the paths of all open files. |
| PopupFiles function | Displays a hierarchical pop-up menu of files and folders and returns the path of the chosen item. |
| PopupFont function * | Displays a pop-up menu of all the characters in a font and returns the chosen character. |
| PopupMenu function | Displays a list as a pop-up menu and returns the chosen item. |
| Proper function | Returns a string with the first letter of each word capitalized. |
| Random function | Returns a random number between 1 and the specified value. |
| Replace function | Replaces occurrences of one string within another. |
| Return function | Returns the carriage return character. Use with Type to press the Return key from a script. |
| SubString function | Returns a portion of a string. |
| Tab function | Returns the tab character. Use with Type to press the Tab key from a script. |
| Trim function | Returns text with extra spaces removed. |
| Upper function | Returns a text value as all uppercase letters. |

| Variable | What it does |
|---|---|
| ASResult system variable | Returns the result of the last AppleScript statement. |
| BeepLevel system variable | Returns or sets the system beep volume level. |
| Clipboard system variable | Returns the contents of the Clipboard. |
| CommandKey system variable | True when the Command key is pressed, otherwise False. |
| ControlKey system variable | True when the Control key is pressed, otherwise False. |
| Cursor system variable | Returns the resource ID number of the current cursor. |
| Dialogs system variable | Enables or disables display of dialog boxes while a script runs. |
| DialogText system variable | Gets or sets the text of the active text box in a dialog box. |
| Directory system variable | Returns or sets the current folder for Open and Save dialog boxes. |
| EditorFont system variable | Gets or sets the font displayed in the Script Editor. |
| EditorSize system variable | Gets or sets the font size displayed in the Script Editor. |

| Variable | What it does |
|---|---|
| Error system variable | Returns the error code of the most recent script error. |
| IgnoreClicks system variable | Causes the system to ignore all mouse activity except clicks on OneClick palettes. |
| IsKeyDown system variable | Returns True when a key is currently pressed, otherwise False. |
| IsMouseDown system variable | True when the mouse button is clicked or held down, otherwise False |
| ListDelimiter system variable | Returns or sets the delimiter character to use with lists. |
| OptionKey system variable | True when the Option key is pressed, otherwise False. |
| PopupMenuFont system variable | Gets or sets the font used in all OneClick pop-up menus. |
| PopupMenuSize system variable | Gets or sets the font size used in all OneClick pop-up menus. |
| ScheduleType system variable | Returns a value indicating why a Scheduled handler was called. |
| ScriptInterrupts system variable | Stops or restarts the script's ability to be interrupted during execution. |
| ShiftKey system variable | True when the Shift key is pressed, otherwise False. |
| SoundLevel system variable | Returns the current speaker volume level (0—7) or sets the volume to a new level. |
| SystemFolder system variable | Returns the path to the System folder on the startup disk. |
| Ticks system variable | Returns the number of ticks (1/60th of a second) since the computer was started. |
| Version system variable | Returns the OneClick version number as an integer. |

| Object | What it does |
|---|---|
| Button object | Accesses or manipulates the properties of OneClick buttons. |
| DialogButton object | Accesses the properties of buttons in a window or dialog box. |
| File object | Accesses or manipulates the properties or contents of files and folders. |
| Menu object | Accesses the properties of menus or menu items. |
| Palette object | Accesses or manipulates the properties of OneClick palettes. |
| Process object | Accesses or manipulates the properties of running applications. |
| Screen object | Accesses or manipulates the properties of monitors. |
| Volume object | Accesses or manipulates the properties of mounted volumes (disks). |
| Window object | Accesses or manipulates the properties of windows. |

| Handler | What it does |
|---|---|
| DragAndDrop handler | Indicates statements to execute when text or a Finder object is dropped on the button. |
| DrawButton handler | Indicates statements to execute whenever the button is drawn or redrawn. |
| MouseDown handler | Indicates statements to execute when the button is clicked, before the mouse button is released. |
| MouseUp handler | Indicates statements to execute when the button is clicked, after the mouse button is released. |
| Scheduled handler | Indicates statements to execute upon a Schedule event. |
| Startup handler | 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. |
| ’ | Encloses a literal text string. Use ’ instead of " if the string already contains the " character. |
| & | 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

# Integration with AppleScript

## 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’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 Mac OS 7.5 and later.

**Note** AppleScript knowledge is not required to use OneClick or write EasyScript scripts, and the OneClick control panel works just fine without AppleScript installed. However, some third-party OneClick palettes may require AppleScript for full functionality.

## 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 Mac OS 8.
AppleScript "Mac HD:Apple Extras: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 Mac OS 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 and other resources for 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.
- **AppleScript for Dummies**, published by IDG Books Worldwide. Designed as a reference for the important features of AppleScript including AppleScript syntax, the AppleScript tools, Script Editor, Scriptable Text Editor, and the Scriptable Finder, this book covers commands associated with the Scriptable Finder, as well as the best of the available freeware, shareware, and commercial scripting additions.
- **AppleScript for the Internet: Visual QuickStart Guide**, published by Peachpit Press. This guide covers AppleScript language basics and is geared towards scripters who want to automate Internet-related tasks and who want to write CGI (Common Gateway Interface) scripts for Mac-hosted web servers.
- **AppleScript Language Guide for AppleScript 1.3.7**, published by Apple Computer, Inc. This book is the definitive description of the AppleScript scripting

language and is an essential reference for anyone using AppleScript to modify existing scripts or write new ones. It also contains useful information for programmers who are working on scriptable applications or complex scripts. The book begins with an introduction to scripting and an overview of AppleScript's main features. Most of the book consists of detailed definitions of AppleScript terminology and syntax. An online version of this book is available:

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

- **AppleScript Finder Guide: English Dialect**, published by Apple Computer, Inc. This is Apple's reference manual for the Scriptable Finder included in Mac OS 7.5 and later. 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. An online version of this book is available:

  http://developer.apple.com/techpubs/mac/pdf/AS_Finder_Guide.pdf

- **AppleScript Scripting Additions Guide: English Dialect**, published by Apple Computer, Inc. This book describes the scripting additions that accompany the English dialect of the AppleScript language. Scripting additions are files that extend AppleScript's capabilities by providing additional commands or coercions for use in scripts, analogous to OneClick extensions for EasyScript. An online version of this book is available:

  http://developer.apple.com/techpubs/mac/pdf/Scripting_Additions.pdf

- **http://www.apple.com/applescript/** Apple's web site includes links to AppleScript online resources, tutorials, and documentation, including HTML and Acrobat PDF versions of the AppleScript Language Guide, Finder Guide, and Scripting Additions Guide.

# Index