

MACROMIND
the multimedia company



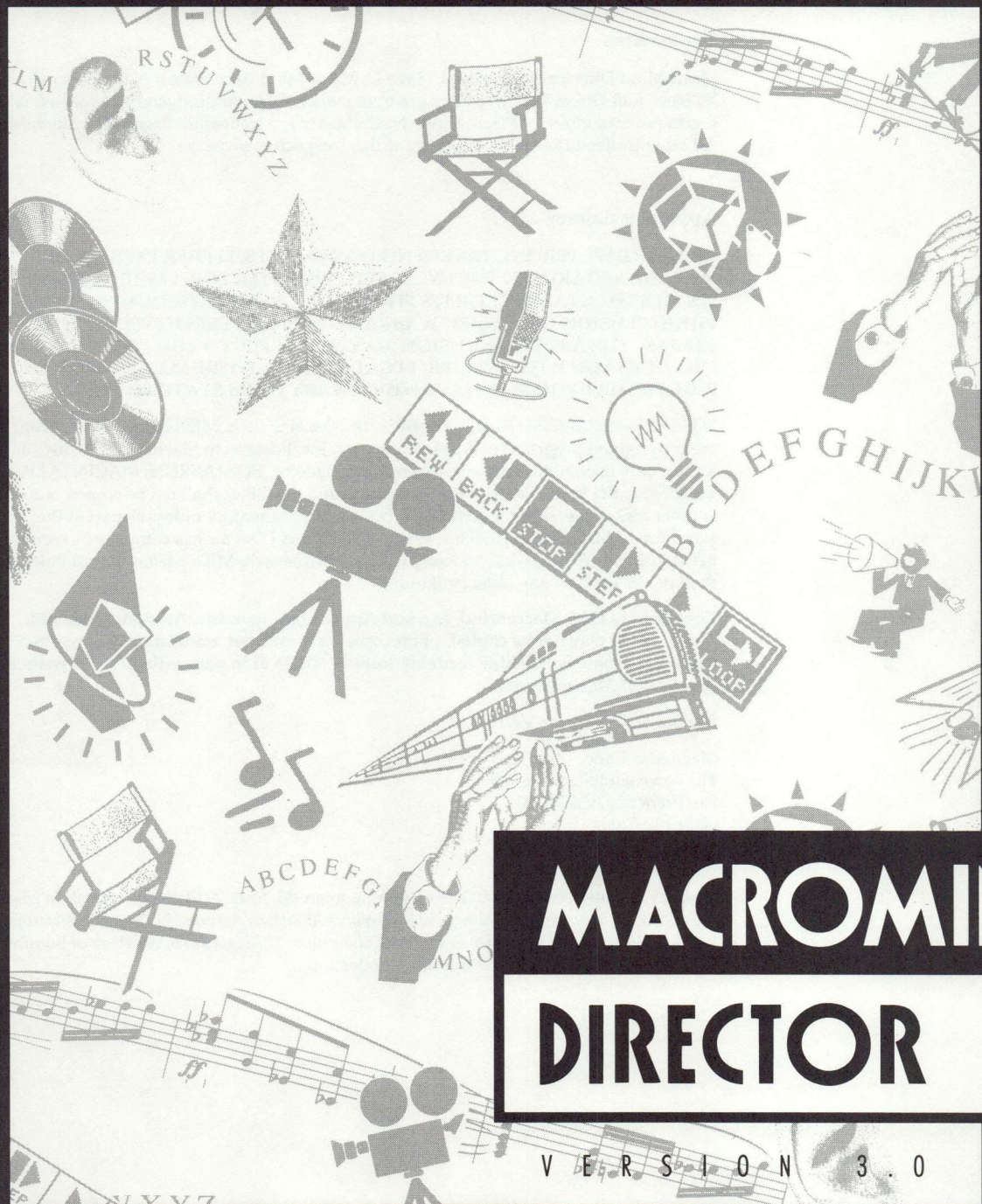
MACROMIND DIRECTOR

V E R S I O N 3 . 0

INTERACTIVITY MANUAL

MACROMIND

the multimedia company



MACROMIND DIRECTOR

VERSION 3.0

Trademarks

MacroMind Director, MacroMind Three-D, MacroMind Accelerator, Art Grabber II, Lingo, XObject, and Object Sensitive Help are trademarks of MacroMind, and MacroMind is a registered trademark of MacroMind. MediaMaker is a trademark licensed to MacroMind. All other trademarks are the property of their respective owners.

Apple Disclaimer

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

BITMAPRGN, MACINTALK, VideoSync, Ilfx Serial Switch, MIDI Manager, and PatchBay are copyrighted programs of Apple Computer, Inc. licensed to MacroMind to distribute for use only in combination with MacroMind Director. BITMAPRGN, MACINTALK, VideoSync, Ilfx Serial Switch, MIDI Manager, and PatchBay shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of the execution of MacroMind Director. When MacroMind Director has completed execution BITMAPRGN, MACINTALK, VideoSync, Ilfx Serial Switch, MIDI Manager, and PatchBay shall not be used by any other program.

Copyright © 1991 MacroMind, Inc. and Apple Computer, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine readable form in whole or in part without prior written approval of MacroMind, Inc.

Second Edition: June 1991

MacroMind, Inc.
410 Townsend St., Suite 408
San Francisco, CA 94107
(415) 442-0200

Chapter opening photograph, Deer Running, plate 86, from *Attitudes of Animals in Motion: A Series of Photographs Illustrating the Consecutive Positions Assumed by Animals Performing Various Movements; Executed at Palo Alto, California in 1878 and 1879*, courtesy of Stanford University Museum of Art, Muybridge Collection.

Contents

Preface 1

- Who This Manual is For 1
- What You Need to Know 1
- System Requirements 2
- How to Use This Manual 2
- Conventions Used in This Manual 3

Chapter 1: Lingo Basics 5

- Introduction to Lingo 6
- About Lingo Scripts 7
- The Lingo Environment 9
- Using the Script Editor 14

Chapter 2: Working With Lingo 21

- Loops 22
- Controlling a Loop's Execution 24
- Creating the Scripts for *Basic With* 32
- More About Editable Text Fields 41
- Creating and Using a Handler 43
- Creating Custom Menus 47
- Troubleshooting and Debugging 49

Chapter 3: Script Parts and Structures 53

- Statements, Scripts, and Expressions 54
- Optional Keywords and Abbreviated Commands 55
- Parentheses 55
- Character Spaces 55
- Upper- and Lowercase Letters 56
- Comments 56

Literals	57
Boolean Expressions	58
Assigning, Calculating, and Communicating Values	59
Handlers	63
Event Scripts	66
Functions	67
Operators	68
Testing and Control Structures	70
Sprites and Puppets	80
The Lingo Message Hierarchy	87

Chapter 4: *The Apartment* Sample Movies 89

Learning From <i>The Apartment</i>	90
• <i>Main Menu</i>	90
Common Script Elements	92
Using <i>editableText</i>	93
Animated and Moveable Sprites	94
The <i>rollOver</i> Movie	96
Basic Event Scripts	98
Simple Puppets	100

Chapter 5: Factories 105

Introduction to Factories	106
How Factories Are Defined	108
Creating Objects From Factories	110
Special Methods in Factories	111

Chapter 6: *The Simple Factories* Movie 113

What <i>Simple Factories</i> Does	114
Establishing the Environment	115

Chapter 7: About XObjects 125

What Are XObjects?	126
Characteristics of XObjects	127
Writing Your Own XObjects	128
XObjects Ready to Go	132

Chapter 8: Using XCMDs and XFCNs 139

Opening XCMD and XFCN Resources	140
Viewing XCMDs and XFCNs	140
Closing XCMD and XFCN Resources	141
Using an XCMD or XFCN	141

Chapter 9: Advanced Topics 149

Managing Color 150

Planning for Video Output 154

Virtual Cast Facility 158

Working With Coordinates 160

Chapter 10: Lingo Summary 165

Category Descriptions 166

Scripting Guidelines 169

Chapter 11: The Lingo Dictionary 175

Appendix A: Lingo Quick Reference 335

Appendix B: ASCII Chart 349

Appendix C: Suggested Reading 359

Glossary 361

Acknowledgments 375

Index 377

List of Figures and Tables

Chapter 1

- Figure 1.1: The elements of the scripting environment 9
- Figure 1.2: The Message window 12
- Figure 1.3: The Lingo menu 15
- Figure 1.4: The script pop-up menu in the Score 17
- Figure 1.5: The Script menu 18
- Figure 1.6: The Find dialog box 18

Chapter 2

- Figure 2.1: The Message window tracing symbols 51

Chapter 3

- Figure 3.1: Three ways to display a selected checkbox 86
- Table 3.1: Arithmetic operators 68
- Table 3.2: Comparison operators 70
- Table 3.3: Logical and string operators 70

Chapter 6

- Figure 6.1: Simple Factory opening screen 114

Chapter 8

- Table 8.1: HyperCard's callback requests 147

Chapter 9

- Figure 9.1: The screen coordinate system 161
- Figure 9.2: The Stage coordinate system 162

Preface

Lingo is an English-like scripting language that lets you expand the functionality of MacroMind Director movies. This manual provides you with an overview of Lingo and a dictionary (Chapter 11) that describes the syntax and use of each word in the language.

Who This Manual is For

This manual is for MacroMind Director users who want to add a broader range of user interactivity to their presentations.

What You Need to Know

This manual assumes that you know how to use your Macintosh computer and are familiar with basic Macintosh computer terms. You should know how to use the Studio functions of Director. They are described in the *MacroMind Director Studio Manual*.

System Requirements

To run MacroMind Director and the example movies used in the hands-on exercises in this manual, you need a Macintosh computer with at least 2 megabytes of memory and System 6.0.5 or greater. Color is required for the color examples, but most of the example movies are in black and white. A hard disk is required.

How to Use This Manual

You can use this manual as a learning tool and as a reference to Lingo.

If You Are New to Programming

- ◆ Read Chapter 2 and work through the exercises there.
- ◆ Begin experimenting with scripts in your own movies.
- ◆ Read Chapter 3 for an overview of how scripts are put together. This chapter also describes important concepts and procedures in Lingo that are not found in other languages.
- ◆ Read Chapter 4 which describes running examples of Lingo scripts in the context of a sample set of movies called *The Apartment*.

If You Are Familiar With Lingo

- ◆ Play and examine the new sample features in the movie called *Feature Examples* in the *Feature Examples* folder provided with Director 3.0. They demonstrate many of the features available to you in version 3.0.
- ◆ Use Chapter 11, “The Lingo Dictionary”, to find specific format and usage information for each Lingo word.
- ◆ Look at the new features in the script editor in Chapter 1.

If You Know Another Programming Language

- ◆ Review the language elements and scripting guidelines in Chapter 10.
- ◆ Read Chapters 4 through 9 for information that will let you use your current experience to full advantage in Director.

Conventions Used in This Manual

Throughout this manual we've used visual and naming conventions that will make things easier to pick out, read, and understand.

Text and Terminology Conventions

- ◆ The terms "Lingo" and "Director" refer specifically to version 3.0.
- ◆ Terms that have special meaning with regard to Director, Lingo, or programming appear in **boldface** at first mention. These terms are defined in the glossary at the back of the book.
- ◆ Within the text, names of buttons, menus, and other elements that appear on the screen are not put in quotation marks unless the name is long or unless it would be confusing otherwise.
- ◆ Within the text, and in the Lingo examples throughout the book, code is shown in typewriter (Courier) font.

Here is a sample line of code:

```
put currentBalls + 2 into currentBalls
```

Here is a sample of code within a line of text:

You use the `set` command to control object properties.

- ◆ Quotation marks that are part of Lingo syntax (for delimiting strings) are shown in the text and in Lingo code examples as straight quotation marks (") rather than as "curly" quotation marks.

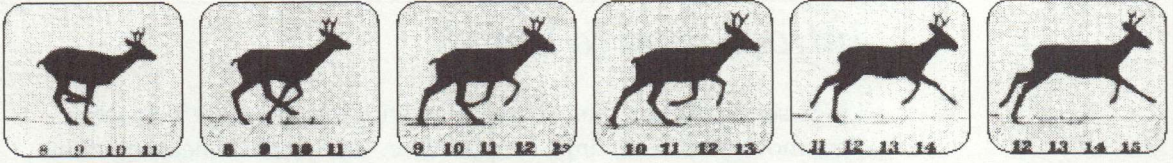
```
"This is a quoted string in Lingo."
```


⇒ **Tips & Hints** This is information that can help you in your programming endeavors.

⇒ **HyperTalk Users** This is a note or short discussion comparing or contrasting Lingo and HyperTalk. If you don't know or don't care about HyperTalk, you can safely skip these asides.

Multiline Lingo Statements

In this manual, long script statements are sometimes broken into two lines. Continued lines of script are indicated by the continuation symbol (→) (entered by pressing Option-Return or Option-L). When you see the continuation symbol in this manual, you can recombine the lines when you type them into the Lingo editor window.



Chapter 1: Lingo Basics

This chapter introduces Lingo and the programming environment, beginning with the new Lingo editor. Chapter exercises show you how to enter scripts in a Script window and how to use the Lingo and Script menus.

To do the exercises, you should already be familiar with the Score window and its component parts: the Script channel, Sprite channels, the script entry box, and so on. If you are not sure what these are or how to use them, refer to your *MacroMind Director Studio Manual*.

Introduction to Lingo

Lingo scripts let you make Director movies that are more interactive than movies without Lingo scripts. Here, “interactive” means that the user can use the mouse and keyboard to control what happens. With scripts, for example, you can control animation, text, color, sounds, and palettes. You can add familiar Macintosh-like buttons, check boxes, and menus to your presentations. Lingo scripts let you pause, jump to specific frames, do audio and graphic transitions, drag graphic objects around the screen, control the sequence of a presentations, or choose from custom menus. As you become proficient at Lingo programming, you’ll be able to use it for more complex applications, such as software simulations or prototyping.

What’s New in Lingo

MacroMind Director 3.0 provides many new Lingo commands and enhanced script management tools. Among the new features for this version are:

- ◆ An expanded script editing facility that lets you enter and change multiline scripts.
- ◆ You can now attach scripts to movies and to individual castmembers.
- ◆ You can now define handlers in scripts (similar to HyperTalk).
- ◆ A Script menu lets you search, change, and replace script elements in one script or across a range of scripts.
- ◆ You can now use HyperTalk-like chunk expressions in your scripts.
- ◆ You can now do floating-point as well as integer arithmetic.
- ◆ A virtual cast facility manages memory automatically. This helps make big movies more memory-efficient to run.
- ◆ Factories and XObjects are now easier to define and manage.

Getting Ready

Before you continue reading this chapter and doing the exercises, you need to set up your Macintosh computer and Director by performing the following tasks.

1. **Start Director 3.0.**
2. **If you are working with a color Macintosh computer, use the Control Panel to set your monitor to Black & White.**
3. **Open the Score window and make sure that Easy Select is checked in the Score menu.**
4. **Make sure Obey Scripts is checked in the Control menu.**

About Lingo Scripts

A Lingo **script** is a set of instructions that tell Director to perform certain tasks while a movie is running. A script consists of one or more Lingo **statements**, single lines of code. A Lingo statement, like an English sentence, must be put together according to a set of rules, called **syntax**. You enter and edit scripts in a **script editor window**, also known as a *script window*.

This manual shows you how to write and use four main kinds of scripts:

- ◆ Movie Scripts
- ◆ Score Scripts
- ◆ Cast Scripts
- ◆ event scripts

The first three, Movie, Score, and Cast Scripts, are named after the Director object with which they are associated. An event script can be part of any other script and has its own formatting rules. The following sections give you a brief introduction to each of these kinds of scripts. There are examples of each kind in the example movies and in the Lingo code samples in this manual. In this manual, when scripts are discussed, you may assume that the discussion applies to all scripts. If it is

necessary to distinguish between the kinds of scripts, then the explicit terms Movie, Score, Cast, or event script are used.

Movie Script

A Movie Script is attached to the movie. You access Movie Scripts with the Script button in the File menu's Movie Info dialog box (or by pressing Command-Shift-U).

There is only one Movie Script per movie.

Movie Scripts let you define what happens when a movie starts, stops, or pauses. You can also define actions that will happen as the playback head moves from frame to frame. You can define handlers in a Movie Script which then are available for use in other scripts. (Handlers are explained in the "Handlers" section in Chapter 3.)

Score Script

A Score Script is one that you place anywhere in the Score window. You access a Score Script by selecting (highlighting) one or more of the Script channel cells, or a sprite channel cell, then clicking in the Script entry box to bring up a Score Script window. It is sometimes necessary to distinguish between a script attached to a sprite cell and a script placed in the Script channel. Whenever needed, either the term *sprite script* or the term *Script channel script* are explicitly used.

The Script channel in the Score window works just like the Tempo, Palette, Transition, and Sound channels you are already familiar with. When the playback head reaches the frame, if there is a Script Channel script for that frame, it is executed.

If there is a sprite script attached to one of the sprite cells in a sprite channel, it takes precedence over the Cast Script, if there is one.

Cast Script

A Cast Script is attached to a castmember. You access Cast Scripts through the Cast menu when the Cast window is open. There must be a castmember selected before you can see its script. Choose Cast Info to open the Cast Info dialog box, then click Script to open the Cast Script window (or Control-Option-click the castmember).

Cast Scripts are executed when the mouse is clicked on the castmember.

Event Script

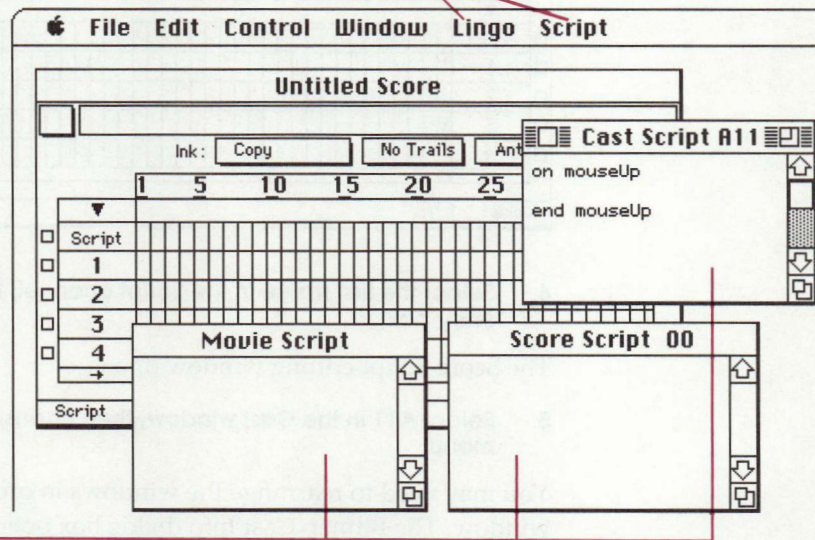
Event scripts let your movie respond to mouseclicks and keyboard actions that occur anywhere *other than* on a castmember. You can define an event script within any other script. In addition to mouse and keyboard events, an event script can also define what happens when there is no activity for a specified amount of time. See the entries for `when keyDown`, `when mouseDown`, `when mouseUp`, and `when timeOut` in Chapter 11.

The Lingo Environment

The *scripting environment* encompasses all the elements of Director that you use while writing and testing Lingo scripts: script editing windows, menus, the parts of the Score window, the keyboard, the mouse, and so on.

The illustration shows you parts of the visual scripting environment.

The Lingo and Script menus are available when you are writing scripts.



The script editing windows. Enter and edit scripts in these windows.

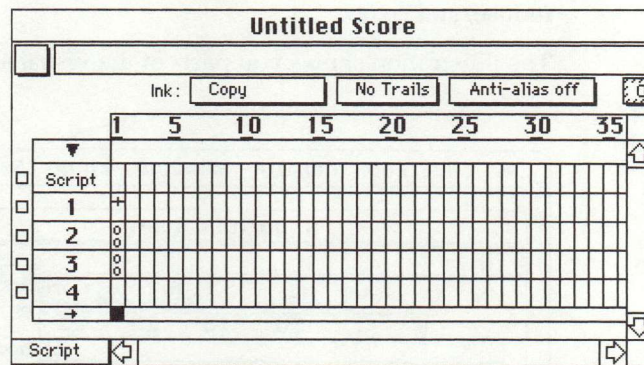
Figure 1.1 The elements of the scripting environment

To open the Script editing windows:

1. **Create a new document by choosing New from the File menu.**
2. **Use the Window menu to open the Cast window and the Score window if they are not already open.**
3. **Put a bitmap graphic castmember in A11, a button castmember in A12, and a text field castmember in A13.**

If you don't remember how to create castmembers, refer to your *MacroMind Director Studio Manual*. You'll need to use the Paint window to create a graphic castmember, and the Tool window to create a button and a text field castmember. Make sure that all three castmembers are on the Stage in the first frame. When you're finished, the Score window should look something like the one in this figure.

For now, you don't need to name your castmembers. Make sure that you've closed the Paint and Tool windows.

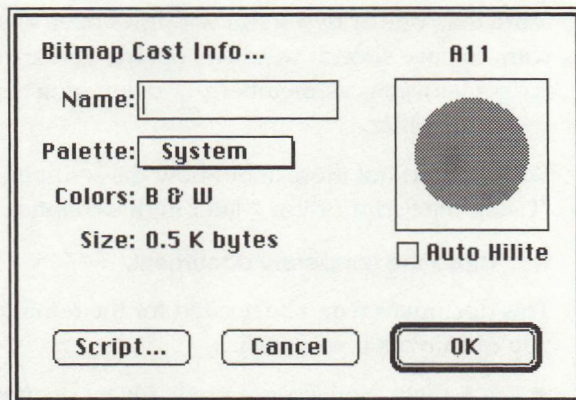


4. **Select the first frame in the Script channel, then click in the script entry box.**

The Score Script editing window opens.

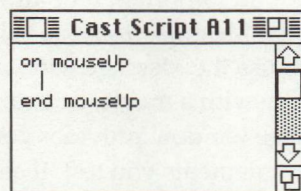
5. **Select A11 in the Cast window, then choose Cast Info from the Cast menu.**

You may need to rearrange the windows in order to get to the Cast window. The Bitmap Cast Info dialog box opens, displaying the cast information. (This dialog box opens because cast A11 is a bitmap.)



6. Click the Script button.

The Cast Script A11 window opens, with the cursor blinking between `on mouseUp` and `end mouseUp`.



7. Choose Movie Info from the File menu.

The Movie Info dialog box opens, displaying the movie information.

8. Click the Script button.

The Movie Script window opens.

Three script editor windows should now be open: Movie Script, Cast Script A11, and Score Script 00. Notice that when you make any one of these script windows active (by clicking somewhere on it), the Lingo and Script menus are available. The Lingo menu provides quick access to all of the Lingo language elements, and the Script menu contains a group of utilities that let you edit, update, and debug your scripts.

Notice, also, that you can open script windows for as many castmembers as you want. In the normal course of scripting you probably won't have

more than one or two script windows open at a time. But you might want to have several windows open if you are duplicating and moving scripts between castmembers, or debugging a set of scripts that interact with each other.

You can find out more about how the script editor windows work in "Using the Script Editor," later in this chapter.

9. Close the temporary document.

This document won't be needed for the remaining practice sessions, but you can save it if you wish.

► **Tips & Hints** You close a movie file by opening another, or by creating a new one.

The Message Window

The Message window is an important tool for Lingo scripting. To display this window, you choose Message from the Window menu, or press Command-M. You can use the Message window to test a script before you store it permanently with a movie, or to test the behavior of Lingo commands. The Message window provides you with a complete history of the commands and statements you test. It also lets you **trace** the execution of your scripts as your movie is played, to provide you with a recording of the execution sequence of scripts. You can then use the trace recording to help you debug and troubleshoot your movies. You can open the Message window any time.

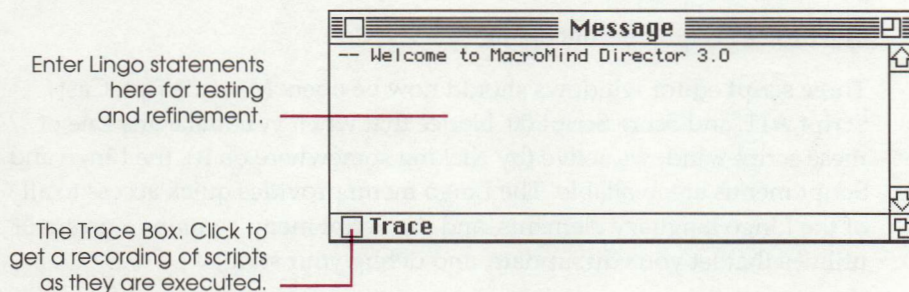


Figure 1.2 The Message window

Script statements that you type in the Message window are executed as soon as you press Return. If a statement doesn't work, you can make changes and try it again until you get the results you want. If the statement isn't valid, Lingo displays an error message box.

To test a statement in the Message window:

1. **Choose Message from the Window menu, or press Command-M.**

The Message window opens with a welcome greeting. Make sure that the Score window is open and that the Message window doesn't cover it up.

2. **In the Message window, type**
`go to frame 31`
and press Return.

In the Score window, the playback head responds to the script and jumps to frame 31.

3. **In the Message window, type**
`put 3+3`
and press Return.

The `put` command displays the result of the addition statement in the Message window.

➡ **HyperTalk Users** The Message window is similar to the Message box in HyperCard, but has several additional features. HyperCard's Message box only lets you see one command line at a time. The Lingo Message window doesn't have this restriction. The Message window also serves as a debugging and tracing window.

You can experiment more with the Message window on your own. The following sections tell you more about using the Message window.

Moving Around in the Message Window

You move around the Message window by using the arrow keys, or by scrolling and clicking.

You can move the insertion point to the top of the Message window by pressing Command-Up Arrow. You can move the insertion point to the bottom of the window by pressing Command-Down Arrow.

Clearing Text From the Message Window

You can clear all the text in the Message window, or as much as you want to clear, by using Command-Delete. To clear all the text from the insertion point to the bottom of the window, press Command-Delete.

To clear the entire window, first press Command-Up Arrow. This will move the insertion point to the top of the window. Then press Command-Delete to delete all text from the insertion point to the bottom of the window.

Script Display in the Score

As you add scripts to the Score window, they are assigned numbers in the same order as that in which you write them. You can have the Score window display these script numbers to indicate which frames and sprite cells are controlled by which scripts. A single script can be attached to more than one cell, so there may be duplicate Script numbers in a Score.

To display the script notation, choose Script from the Display pop-up menu at the bottom left corner of the Score window.

The number of the script that controls a cell is displayed in that cell. Any cell that has a castmember but no script displays the characters 00. A plus sign (+) is displayed in any sprite cell whose castmember has a Cast Script.

Using the Script Editor

You write Movie, Score, and Cast Scripts using the Lingo script editor. The script editor is active whenever you have a script window open and active. When the editor is active, you also have access to the Lingo and Script menus.

To edit a Score Script, for example, you select a Script channel cell, then click in the script entry area at the top of the Score window.

The Lingo Menu

You can use the Lingo menu to insert commands or any other language element at the cursor location. If parameters are required, Lingo will show you the required additional information using placeholder names. If there is more than one required argument or parameter, Lingo will highlight (select) the first one for you, so all you have to do is type to replace it. You'll have to select the second and subsequent ones yourself.

The Lingo menu itself is a menu of categories. First you choose the category, then you choose the actual element from the submenu. Choosing an element pastes the Lingo command into the Message window, script entry area, or Text window. Try it out now in the Message window.

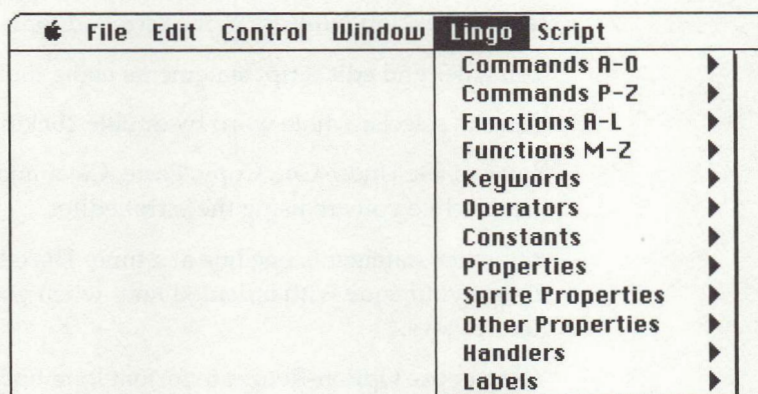


Figure 1.3 The Lingo menu

Notice that the Lingo menu also includes a submenu called "Handlers". When you've loaded a movie that has handler scripts, the names of all the handlers for that movie will appear in the Lingo menu, under the Handlers submenu. (See "Handlers" in Chapter 3 for more about handlers.)

If you forget the details of syntax for a particular Lingo word, there's a quick way to get help, as follows.

To get on-line help for Lingo words:

1. **Hold down the Shift and Option keys.**

The pointer changes to the Help pointer (a question mark).

2. **Continue to hold down the Shift and Option keys and use the mouse to choose one of the items from a submenu of the Lingo menu.**

The Help window opens to the description of that Lingo command. You can then move through the help sections by clicking Next and Previous in the Help window.

Script Editing

You enter and edit text in a script editor window much as you might use a word processor. Of course, since it's meant for writing scripts and not, say, for desktop publishing, the script editor is more streamlined.

You enter and edit script statements using the keyboard and mouse.

You can select a whole word by double-clicking the word.

You can use Undo, Cut, Copy, Paste, Clear and Select All from the Edit menu while you are using the script editor.

You enter statements one line at a time. The editor will automatically format your code with indented lines when you press the Tab or Return keys.

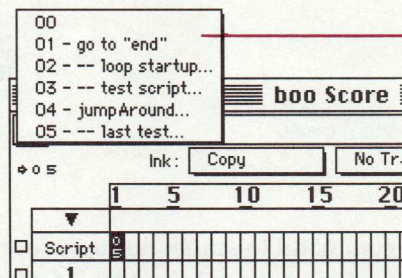
You can use Option-Return to format long lines of code.

Press the Enter key or click the close box when you are finished using the script editor. Lingo will close the window and save it as a new script.

➡ **Note** Each time you open and change an existing Score Script, whether you edit it or not, when you press Enter the script will be saved as a new script, and a new number will be assigned to it. If you want to *replace* a script wherever it has been used in the Score, press Option-Enter or hold down the Option key while you click the windows close box, which *does not* create a new script.

As you write new scripts for the Script channel of the Score window, they are assigned numbers. These numbers then appear in the Script

number pop-up menu at the upper left corner of the Score window. The number of the script you are assigning also appears in its assigned cell in the Script channel. Subsequently, you can choose to put a previously-written script into any new location, or change the script being used in a particular cell, simply by using the pop-up menu. (Movie Scripts and Cast Scripts do not appear in the numbered menu in the Score window.)



The script pop-up menu. Each script that is used in the Score window is assigned a number. Click on the pop-up box to get the menu of numbers and first lines. The one you choose will be entered in the selected cell(s).

Figure 1.4 The script pop-up menu in the Score

To see how to use the Lingo menu while writing scripts:

1. **Open the Message window.**
2. **Choose `beep` from the Commands A-O submenu of the Lingo menu.**

The statement

`beep numberOfTimes`

is entered into the Message window for you. Notice that `numberOfTimes` is highlighted so that you can enter a number to tell Lingo how many times to beep. The highlighted word is called a placeholder and serves as a reminder that you can use an optional argument with `beep`. Placeholders will also remind you of required parameters.

3. **Press Return.**

If you press Return while the placeholder is still highlighted, the selected word will be replaced by the Return, Lingo executes `beep` one time, and you hear a beep.

The Script Menu

The Script menu provides assistance during script editing, documenting, and debugging.

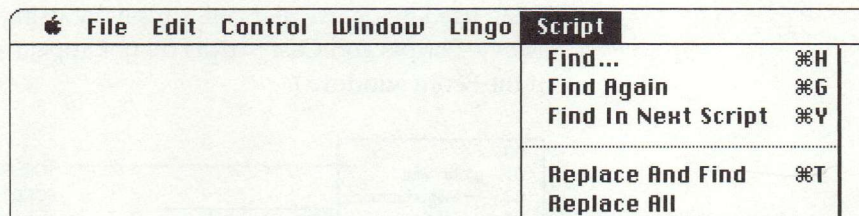


Figure 1.5 The Script menu

Find

Use the Find command to search through your scripts for a particular word or words. The keyboard shortcut is Command-H. When you choose Find, you see the Find dialog box. Enter the string you want to find and press Return or click the Find button. Find is not case-sensitive: `ThisHandler`, `thisHandler`, and `THISHANDLER` are all the same for search purposes.

➡ **Tips & Hints** A Find operation begins at the point where you've positioned the cursor in a script. If you want to search from the beginning, make sure the cursor is at the upper-left corner of the script window, at the beginning of the script, before you begin a search.

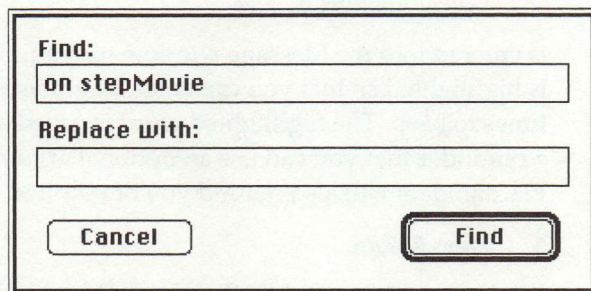


Figure 1.6 The Find dialog box

Find Again

Find Again searches only the current script, starting from the insertion point. The keyboard shortcut is Command-G. Find Again does not display a dialog box; it assumes that you've already entered something to find in the Find dialog box.

Find In Next Script

Use Find In Next Script to search through more than one script location at a time by skipping to the next script location. The keyboard shortcut is Command-Y. The search order is as follows:

- ◆ Score Scripts as numbered in the pop-up menu
- ◆ the Movie Script
- ◆ Cast Scripts

If the text is not found after searching through all the scripts, the computer beeps.

Replace And Find

Use this command to replace a word or group of words in your scripts. The keyboard shortcut is Command-T. Enter the string you want to find and the string you want to replace it with in the Find dialog box. Press Return or click Find to find the first occurrence of the searched-for string. Choose Replace And Find to replace and find the next occurrence of the searched-for string.

Replace All

Use Replace All to replace all occurrences of the string specified in the Find dialog box in the current script.

Removing a Script From the Score

To remove a script or scripts from any cells in the Score, select the cell(s) containing the script(s) you want to remove, then choose 00 from the Script pop-up menu.

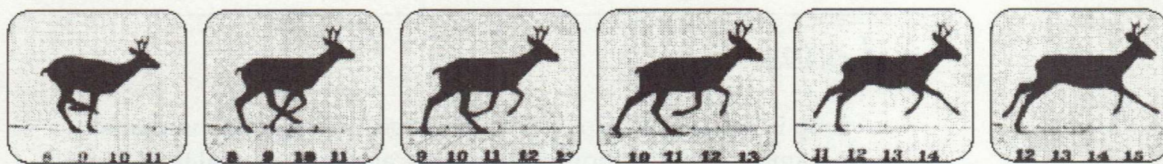
If the Script display is on, Lingo immediately removes the script number from the selected cells, replacing them with 00 in cells in which sprites have been placed. Cells without castmembers remain blank. Sprite cells whose castmembers have scripts display a plus sign (+).

Script Numbers

In the Score window, script numbers start with 01 and are incremented for each new script you write, in the order in which you enter them. A script is entered in the Score whenever you press Enter to finish creating or editing a script, or when you close the Script editing window. When you make changes to an existing script, the changes are saved as a new script, and the old one, with its earlier number, will be preserved (unless the Option key was down).

If you inadvertently create an unwanted script, you can remove it from the cell by selecting the cell and choosing 00 from the pop-up menu, as described above. Scripts in the pop-up menu that haven't been placed in at least one of the cells of the Score are deleted when you close the movie, and the scripts are renumbered accordingly. Script number 00 is always available in the script pop-up menu.

Now that you've seen how the script editor works and how to access the various kinds of scripts, you can go on to the next chapter for more hands-on exercises.



Chapter 2: Working With Lingo

This chapter gives you hands-on exercises that show you how to make loops in an animated sequence. Loops are fundamental techniques used in movies and scripts. You'll also learn how to use buttons to control an animation, and get a first look at handlers. The exercises are meant to be used with three demonstration movies: *Loops*, *Basic With*, and *Basic Without*. You can find these files in the *Interactive Tutorials* folder.

Loops

A **loop** is any sequence of events that repeats. Loops are used in all programming and authoring languages, and are essential to Director movies.

With Lingo scripts you can create loops that are controlled by the click of a button, by typed text, or some other interactive sequence you've defined.

These first exercises explore loops: how to make them, how to add start and stop controls in your loops, and how to put loops to work.

To create a loop movie for the exercises:

1. **Create a new document by choosing New from the File menu.**

You may want to save this movie and give it a name such as *My Loop* to distinguish it from the *Loops* file in the *Interactive Tutorials* folder. The *Loops* file contains a finished version of the scripts that you'll be creating in the following exercises.

2. **Create a ball castmember and place it on the stage to make a sprite that moves horizontally across the top or bottom of the Stage. Use the first ten frames of channel 1 for the animation.**

Create either a bitmap ball or a shape using the Tool window. Use In-Between to create the animated sequence. The idea is to create a simple 10-frame animation. If you need more instructions on how to create this sequence, see your *MacroMind Director Studio Manual*.

3. **Use the same ball castmember to make an animation that moves vertically from top to bottom at the left side of the Stage. Use frames 11 through 20 of channel 1 for this sequence.**
4. **Click the Play button in the Control Panel to play the movie you have just created.**

Assuming that you have selected the Loop button in the Control Panel, you can watch your movie repeat itself, with the ball moving horizontally, then vertically. If you open the Score window while you play the movie, you can watch the playback head cycle through the frames.

This is an example of a simple loop. When you turn off the Loop button, your movie reverts to a linear presentation that begins when you click Play and ends at the last frame.

Now that you've created a loop movie, you can add a Script channel script to the Score window that demonstrates another way of making a loop.

To create a loop using a Score script:

1. **In the Score window, select the Script channel cell for frame 10.**
2. **Click the Script Entry box.**

The Score Script window opens, with the cursor positioned ready for you to type in the script.

3. **Type:**
`go to frame 1`

You could also have used the Lingo menu to select `go to`.

4. **Press Enter.**

Lingo automatically numbers the script and displays the number in the selected cell and in the script pop-up menu in the upper-left corner of the Score window. The script line is displayed in the Script Entry box.

5. **Play back the movie.**

Make sure you have turned off the Loop button in the Control Panel. If the Score window is open while you play the movie, you can watch the playback head go through its paces. When you're tired of watching it, press Command-W to stop playback.

In this example, the statement

```
go to frame 1
```

sends the playback head to the first frame each time frame 10 is displayed, creating the loop.

The process you used to create this loop is fundamental to working with all scripts: you choose where you want the script to be active, open a script window, type a script, and press Enter. Then you run and test the

movie to see if it executes as you expected. You can think of different loops (whether a series of looping frames or a set of repeating Lingo statements in a script) as different ways of controlling the action in your movies.

➡ **Note** If a movie does not perform as described here (or, for that matter, the way you expected it to perform when you wrote the script), check the script to make sure you've entered commands correctly. Misspelled Lingo words are a common source of script error messages.

Controlling a Loop's Execution

Your loops will continue to execute unless you program in some way to get out of them. You (as the designer and as the surrogate user) want to be able to stop, re-start, and move between different loops easily. In the preceding example movie, the only way you can stop it is to press Command-W.

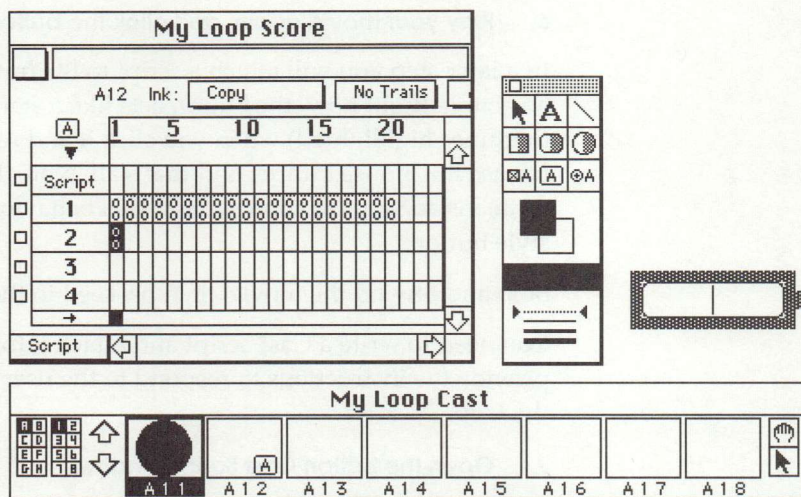
The Command-W method of stopping a loop may be adequate for some kinds of movies, but it has the disadvantage of stopping the movie completely, including execution of any scripts.

For real life applications, you'll probably want a more elegant way to break out of a loop or to move between loops. The following exercises will show you how to achieve this flexibility. You'll begin by creating a button to stop a loop's execution.

To create a button to stop the simple loop:

1. **Stop and rewind your loop movie (from the previous tutorial) so that the playback head is in frame 1.**
2. **Open the Tool window.**

You use the Tool window to create regular buttons, check boxes, and radio buttons. The regular button tool is the middle tool in the third row.



3. Select the button tool (the middle tool in the third row), then draw a rectangle in the lower-right corner of the Stage.

Place it so it will be out of the way of the horizontal movement of the ball.

The rectangle displays a cursor in the middle of the button, indicating that you can type in a text label for the button. The label should clearly describe the button's action. Since this button will start the vertical sequence, you can call it the Vertical button.

4. **Type**
vertical

Any button you create has all the standard Macintosh text **properties** for the font, size, and style. (A property is an attribute of an object such as a button. You'll learn more about properties and how to change them with scripts in later exercises.)

5. In the Score window, select cells 1 through 10 in channel 2, then choose **In-Between** from the Score menu.

This places the button on the Stage in every frame of the horizontal sequence. Leave the cells selected when you have completed this step.

6. Play your movie again, and click the button you have created.

In a later step you will attach a script to the button so that it can control an action. Right now, the button acts like a standard button: it inverts (becomes highlighted) when you click it and returns to normal when you release the mouse button, and that's all. Note that you do not have to write instructions to control the visual behavior of standard Macintosh-style buttons.

Stop and rewind the movie, then proceed to the next step.

You need to write a Cast Script and attach it to the button. The script will provide the instructions to respond to the user's mouseclick and leave the loop.

7. Open the button Cast Script window.

Select the button, either on the Stage or in the Cast window, and then use the Cast menu to open the Cast Script window (or Control-Option-click the castmember). The script window opens with the `mouseUp` handler definition begun for you.

8. Type

```
go to frame 11  
and press Enter.
```

9. Play the movie again.

Now when you click the Vertical button, the movie jumps to the vertical sequence once, then stops (you should not have the Loop button selected in the Control Panel).

Using Text Field Input For Loop Control

Let's say you want to change the playback sequence of a movie based on information that your user types into the computer. You can do this by adding an editable text field for input. You can then use the input to make further branching decisions.

To create an editable text field:

- 1. Stop and rewind your loop movie (from the previous tutorial) so that the playback head is again at frame 1.**

2. **Open the Tool window.**
3. **Select the Text tool and draw a rectangle in the lower-right corner of the Stage.**

Place it above or below the button that is already on the Stage. Be sure it will also be out of the way of the horizontal movement of the ball.

The text box is ready to accept any text you want to enter at this time. If you don't type anything, the box will start out empty.

4. **Type**
Enter a greeting here.
5. **In the Score window, select cells 1 through 10 in channel 3 (or whatever channel you used for the field), then choose In-Between from the Score menu.**

This action copies the text field onto the Stage in every frame of the horizontal sequence. Leave the cells selected when you have completed this step.

6. **Make sure the text castmember is selected. Choose Cast Info from the Cast menu. Click the Editable Text checkbox and click OK.**

This makes the text sprite editable. Now, the user can type into the field while the movie is running, and even replace the greeting you entered earlier.

The next exercise shows you how to use frame markers and labels to organize the movie sequences you build.

Using Frame Markers and Labels in Loops

You have used markers and labels in Studio to indicate the beginning of specific animated sequences. Once you've labeled a frame, you can use the label name in your scripts. This is important because references to frame numbers may become invalid if you insert or delete frames in the Score. Markers and labels let you use label names that remain constant no matter how much you edit the Score.

In these practice loops, you'll use labels to send the playback head to specific sequences.

To create a frame marker and label:

1. **Using the loops movie you made in the previous exercises, drag a marker from the Marker Well to frame 11.**

The marker shows a blinking cursor, ready for you to enter a label for the frame.

2. **Type**
vertical
and press Return.

Testing Text Input in a Loop

You've already made a field for text input. Once the user has typed in the text, you need have a way for the user to tell Lingo the input is finished and is ready for testing.

Rewind the movie by pressing Command-R. Use the button tool to create a new button on the Stage, just as you did to place the first button. Place the button below the editable text field, in frame 1 of channel 4. Name it "Test the Text."

To place the new button:

1. **In the Score window, select cells 1 through 10 in channel 4, then choose In-Between from the Score menu.**

Leave the cells selected when you have completed this step.

2. **Open the Test the Text button's Cast script window and enter the following script statement for the mouseUp handler:**

```
if field A13 contains "hello" →  
    then go to frame "vertical"
```

The script uses an `if` statement to test the text for the word "hello." Notice, in the script above, that the text field you created is referred to by its Cast number (A13). If your button's text appears in a different Cast position, use its actual Cast number in place of A13. Better yet, you can name the text castmember (in the Cast Info dialog box) and thereafter refer to it by name in your scripts.

► **Tips & Hints** The character created by pressing Option-Return (↵) or Option-L, when used in scripts, causes the next character to be ignored (the Return character). The ↵ character gives you a line break when typing very long scripts. This is useful for visual formatting and won't affect the execution of your scripts. When you are typing in the text of scripts from this manual, you do not need to include Option-Return to break the line. Instead, you can just type in the two lines from the example in a single line, omitting the ↵. Throughout the examples in this book we have broken up long lines to make them more readable.

Now, each time the user clicks Test the Text, if the word “hello” appears in the text, the playback head will jump to the frame labeled vertical.

3. **Run your movie again.**
4. **Type “hello” in the text field, replacing the “Enter a greeting here” text, and click Test the Text.**

The ball drops and the movie stops.

This concludes the Loops exercises. The next series of exercises will expand on what you've learned about loops so far.

Previewing the Basic With Movie

In the Interactive Tutorials folder, there are two sample movies: *Basic With* and *Basic Without*. These two movies look identical when you first open them. The difference between them is that *Basic With* includes the scripts that make it an interactive movie, and *Basic Without* doesn't.

In the practice sessions to follow, you will use the *Basic Without* interactive movie. You will apply and expand the techniques you learned in the loop exercises.

First, preview how the finished *Basic With* movie is supposed to run.

To preview *Basic With*:

1. **Open *Basic With*.**
2. **Choose Stage from the Window menu.**

This hides all windows other than the Stage window.

3. **Press Command-A to start the movie (this is the same as choosing the Play command from the Control menu).**

The *Basic With* interactive movie includes two segments called "Same or Different?" and "Name the Animal."

Try clicking either of these buttons, or choose an item from My menu. Try some correct and incorrect responses to see what happens. Try doing nothing at all: don't press the mouse button or any key on the keyboard for at least 20 seconds. When you're done exploring, continue with the remaining steps.

4. **Press Command-W to stop the movie.**
5. **Open the Score window.**
6. **Play the movie again.**

As the movie plays and you click the buttons in the two games, you can watch the playback head responding to the scripts. Scroll the Score to see where the playback head is, or click the jump button in the lower left of the Score to automatically scroll to the playback head.

To look at the scripts for the sample movie:

1. **Choose Script from the Display pop-up menu (in the lower left corner of the Score window).**

The Score changes to display Script notation. Numbers appear in the cells that contain scripts. Each script has an identifying number associated with it, so that you can tell where in the Score the script is assigned. Cells without scripts display 00. Sprite cells that have Cast scripts associated with their castmembers display a plus sign (+).

You are already familiar with the Script display in the Score window. In the *Basic With* score, you can see several script numbers in the Script channel. In addition, this interactive document makes extensive use of labeled markers, which you can see in the marker channel.

2. **Click frame 29 in the Script channel to select it.**

When you select the cell, you see the script in the script entry area. It says go to frame "main loop". This instruction sends the playback head to the frame with the marker labeled `main loop` in frame 8.

Every time the playback head reaches this frame, it returns to the earlier frame, thus creating a loop.

Notice that the loop does not have to go all the way back to frame 1. The interactive movie starts playing with frame 1, so you can use the first frames to initialize any features that need to be in place for the rest of the presentation. You can select these cells to see what happens here.

3. Open the Movie Script.

To open the Movie Script, first choose Movie Info from the File menu. Then click Script in the Movie Info dialog box.

This script includes two handlers: the `startMovie` handler and the `checkKey` handler. The `startMovie` handler is a special handler that Director executes every time the movie is started. The `checkKey` handler is called by a `keyDown` event script, set up in frame 93. It is executed whenever the user presses a key on the keyboard.

4. Review an event script.

The first statement in the `startMovie` handler is an example of an event script. The event that is being controlled is `timeOut`. The statement sends the playback head to a marked frame when the `timeOut` event happens.

The second statement in the script sets the timeout length to 20 seconds. It says:

```
set the timeoutLength to 20*60
```

The `timeoutLength` function determines how long Lingo will wait for the user to do something. The `timeoutLength` is measured in units of computer time called **ticks**, at 60 ticks per second.

An event script stays in effect no matter where the playback head is located. If nothing happens on the Stage for a specified time (20 seconds in this case), the event script is activated.

5. Review a menu installation.

The last statement in the `startMovie` handler installs a menu. The menu itself is defined in castmember A36.

All of the scripts in the `startMovie` handler are executed once, at the beginning of the interactive movie. They do not need to be activated again.

Let's look next at a Cast Script.

6. Open the Cast Script for castmember A11.

To open the Cast Script, first open the Cast window and select castmember A11. Then select Cast Info from the Cast menu. Finally, click Script in the Cast Info dialog box.

You see a `mouseUp` handler with a single statement: `go to frame "same or different"`. This instruction sends the playback head to the marker labeled "same or different".

This script is activated whenever the user clicks the Same or Different button while the movie is playing. A cast script is activated when the user clicks the cast member on the Stage.

7. Return to the Score Window and select some other cells to see their scripts.

Notice that each script has a number that is displayed in the script pop-up menu. You can display the pop-up menu to see all the scripts used in this presentation.

➡ **Tips & Hints** Be careful that you don't inadvertently select a script when displaying the pop-up menu. You must move the pointer entirely off of the menu if you don't want to inadvertently install an inappropriate script in a selected cell.

Don't worry about what all the scripts specifically mean at this point. The meaning and structure of the scripts are explored in later exercises.

Creating the Scripts for Basic With

The rest of the exercises in this chapter use *Basic Without*, which doesn't have any scripts. The goal is to make *Basic Without* look like and thus behave like *Basic With*.

You'll go through the process of adding scripts to recreate the *Basic With* movie.

Many of the steps will be familiar to you if you completed the preceding "Loops" exercises. Here, you'll learn about four new commands: pause, continue, play, and play done.

The pause and continue commands allow you to halt the movie until a particular sprite is clicked. The play and play done commands allow you to control the playback head more efficiently than with the go to command (in appropriate circumstances). Together, these four important commands give you a new way to control your movies.

To test *Basic Without*:

1. **Open the document called *Basic Without*.**
2. **Press the Shift key and click the Play button in the Panel.**

Pressing the Shift key and clicking the Play button is a shortcut for choosing Stage from the Window menu and then clicking Play in the Panel.

The presentation plays straight through, without letting you interact with it. You cannot initiate actions by clicking or typing as you could in the *Basic With* example.

3. **Press Command-W to stop the movie.**

Now you can add scripts that will provide the remaining interactive capability.

To add the first script:

1. **Open the Score window and drag a marker from the marker well to frame 8.**
2. **Type `main loop` and press the Return key.**
3. **Select frame 29 in the Script channel and open its script window.**

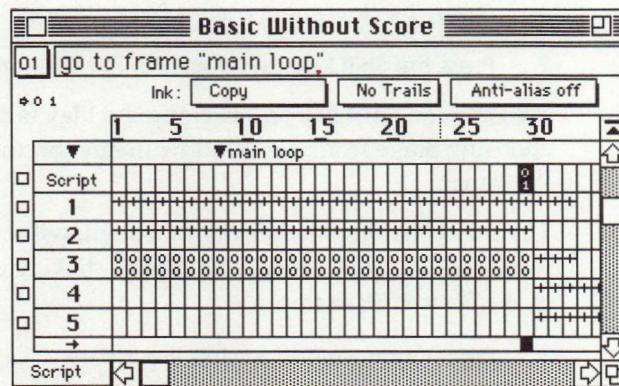
4. **Type**
go to frame "main loop"
and press the Enter key.

This script is executed when the playback head reaches frame 29.

Notice that when you press the Enter key, a code number (01) appears in frame 29, signifying that the first script has been placed there. The number lets you see where the script is used.

Don't worry if your script numbers are sometimes not the same as the ones in these exercises. This can happen if you add different scripts or modify existing ones while you're exploring on your own. The important thing is to make sure that the scripts are written exactly as they are shown here. You can let the script numbers take care of themselves.

The Score window should now look like this through the first 30 frames:



To test the new script:

1. **Click the Rewind button in the Panel.**
2. **Press the Shift key and click the Play button in the Panel.**

You've created a loop for the sequence of frames containing the main menu. Instead of the entire movie playing through from beginning to end, the playback head jumps back to frame 8 whenever it reaches frame 29, and the ball bounces.

3. Press Command-W to stop the movie.

In the last exercise you placed a script into the Script channel. Now it's time to assign scripts to the buttons in the application's opening sequence. These scripts will send the playback head to another frame when a button is clicked.

First, you need to place markers on the frames for the button scripts to use.

To add the button scripts:

1. Drag a new marker to frame 30.

2. Type same or different and press Return.

This marker indicates one point to which the playback head must jump, the beginning of the same or different sequence of frames.

Now you can write the Cast Script that controls the playback head.

3. Open the Cast Script window for the "Same or Different" button.

4. Type go to frame "same or different" and press the Enter key.

When you press Enter, the `mouseUp` handler is saved with the castmember.

5. Drag a new marker to frame 93.

6. Type name the animal and press the Return key.

The marker indicates a second set of frames to which the playback head must jump, the beginning of the "What is the name of this animal?" sequence.

7. Open the "Name the Animal" button's script window.

8. **Type**
go to frame "name the animal"
and press the Enter key.

Now you're ready to test the movie again.

To test the scripts you just entered:

1. **Click Rewind in the Panel.**
2. **Press the Shift key and click Play in the Panel.**

After clicking either of the buttons, the go to frame script causes the playback head to jump either to the same or different marker or to the name the animal marker. The playback head then continues moving through frames not yet controlled by scripts—which you can now install.

3. **Press Command-W to stop the presentation.**

The pause and continue commands

Now you will add scripts to sprites in the same or different sequence. This sequence asks which of three pictures does not belong in the group.

To allow the user time to choose a response, you will use pause and continue commands. When the user clicks one of the pictures, you want an animated response to appear. In the case of the correct response, you want a congratulatory message. In case of an incorrect response, you want to present a helpful message and to give the user another chance.

To stop the movie at the question "Which of these objects doesn't belong?" you will set a pause in frame 32 of the Script channel.

To enter the pause and continue scripts:

1. **Select the Script channel cell for frame 32.**
2. **Open the script window and type**
pause
and press Enter.

The correct answer is the car, and the animated sequence for the correct choice follows immediately in frame 33. So you associate the `continue` command with the car. This lets the playback head move on to the correct animation when the user clicks the car.

3. **Select frame 32 of channel 4 (the car).**
4. **Open the script window, type**
`continue`
and press Enter.

Now the presentation will pause when it reaches frame 32, and it will continue only if the correct sprite is clicked.

Before looking at how to handle incorrect responses, you should go ahead and finish the scripts for the correct response sequence.

You will add a script to the end of the correct animation sequence to return to the `main menu` loop (the bouncing ball and the two buttons). This will allow the user to select the other sequence. Since this script is the same as one you typed in earlier, you can reuse the earlier script.

5. **Select cell 69 in the Script channel.**
6. **Choose** `go to frame "main loop"` **from the script pop-up menu. It should be number 01, the first in the pop-up list.**

Recall that this script was the first one you created in *Basic Without*. (If you forgot to save your earlier work, just go back and enter the script again.) It appears at the top of the pop-up menu. When you choose the script from the menu, it is displayed in the script entry area, and its number is also displayed in frame 69.

Now when you click the "Same or Different?" button, the movie will pause when it reaches frame 32. When the car is clicked, the animation resumes at frame 33, and the car drives off the stage. The sequence for a correct answer plays until frame 69. The `go to frame "main loop"` script then causes the playback head to jump back to the opening animation.

The play and play done commands

An incorrect answer in the same or different sequence requires a different response. When either the dog or cat are clicked, the action branches to the animation to respond to an incorrect click, and then returns to the frame of the original pause. You could easily do this by using the `go to frame` statement again. Instead, you'll see how to accomplish the same thing with two new commands: `play` and `play done`.

You will use the `play` statement to move the playback head to a particular sequence of animation. At the end of the sequence you will use `play done` to send the playback head back to its original position. By using `play` and `play done`, you don't have to specify where to return—because the `play` command remembers the starting frame so that this information can be used by `play done`.

To create a new marker for the incorrect response sequence:

1. **Drag a marker to frame 71.**

2. **Type**
`incorrect answer`
and press Return.

Now you can use the `play` command in scripts attached to the incorrect castmembers.

3. **Open the script window for castmember A13 (the dog).**

4. **Type**
`play frame "incorrect answer"`
and press Enter.

5. **Repeat this procedure with castmember A14 (the cat).**

To complete the sequence, you need to add a `play done` script at the end of the `incorrect answer` sequence. The `play done` statement causes the playback head to jump to the frame containing the original `play` statement, so the user has another chance to give the correct answer.

To enter the `play done` command:

1. **Select the Script channel cell for frame 91.**
2. **In the Score Script window, type**
`play done`
and press Enter.
3. **Rewind and play the movie to test the new scripts.**

Now, when you click “Same or Different?”, you should see “Which of these things doesn’t belong?” and the three choices. Clicking the cat or dog plays the `incorrect answer` sequence. When the playback head reaches frame 91, the `play done` statement is activated. The animation returns to the frame from which the `play` command was given, and you can try again. As before, clicking the car starts the `correct answer` animation sequence and returns you to the main menu.

The `play` command is like the `go to` command, in that it immediately moves the playback head to a particular frame in the current or another movie.

The advantage of the `play` command over the `go to` command is that you don’t have to specify the frame to which the playback head should return. `Play done` performs that task automatically, returning the playback head to the frame from which the sequence started. For example, you can jump to the same sequence from different locations and have the playback head automatically return to the frame you jumped from.

With `play`,

- ◆ The playback head returns automatically to the frame from which it initially jumps.
- ◆ Either the `play done` statement or the end of a document causes a return to happen.

You can specify the frame to play in several ways:

- ◆ Use the frame number:

```
play frame 6
```


- ◆ Use the marker label, entered as text:

```
play frame "Choice1"
```

- ◆ Use another movie's file name:

```
play movie "Joining of the Rails"
```

- ◆ Use a frame in another movie:

```
play frame "Begin Special" of movie "Chicago"
```

or

```
play frame 5 of movie "Bouncing Ball"
```

You can use expressions or variables to specify the movie's name, frame number, or marker label. (Expressions and variables are discussed again in Chapter 3.)

Note that you can use any of the above variations with the `go to` command as well.

Use `play if`:

- ◆ the movie you want to play does not have instructions about where to return.
- ◆ you want to nest sequences (put one inside another, inside another... and so on). In conjunction with `play done` your scripts can return to any level. (Each `play done` command returns to the previous loop).
- ◆ you want to jump to one loop from several different locations.
- ◆ you want to play several movies from a single script. When one movie finishes, control returns to the part of the script that issued the `play` command and resumes executing the subsequent line(s).

More About Editable Text Fields

The `name the animal` sequence includes an editable text field to accept keyboard input. In the upcoming practice sessions, you'll see how to test the text entered by the user. You'll then program alternative actions to be taken depending on the test result.

The frames between 93 and 109 contain an animated jumping frog. First, you'll need to create a loop to animate the frog.

To make the frog loop:

1. **Select the Script channel cell for frame 109.**
2. **Open the script window and type**
`go to frame "name the animal"`
and press Enter.

This script causes the playback head to jump back to the marker previously set at frame 93.

The text `castmember type here` was created with the text tool from the Tool window. It is castmember A35. To make it editable, click Editable Text in its Cast Info dialog box.

Now when you play the movie, you can enter text from the keyboard in the text field while the frog continues jumping. The text field is highlighted.

Go ahead and try out any text entry you want. In the next section you will see how to set up scripts that evaluate what's typed. But right now there's no way out of this loop, so stop the movie.

3. **Press Command-W to stop the movie.**

Because you have made the text editable, whatever is typed into the text field on the Stage replaces the text that was previously contained in that castmember. You can check this now by looking at castmember A35 in the Cast window. Its contents should reflect whatever you happened to type in during your previous testing.

Now it's time to establish a correct sequence and an incorrect sequence, and then set up a test that will evaluate what the user has entered.

To add labels for the correct and incorrect sequences:

1. **Drag a new marker to frame 112.**
2. **Type**
`wrong frog`
and press Return.
3. **Drag another marker to frame 134.**
4. **Type**
`correct answer`
and press Return.

Now you can add the scripts that test the input. The test has three parts:

1. **A test for all keystrokes. You test all keystrokes so you will know when the Return key is pressed, indicating that the user is finished.**
2. **When the Return key is pressed, you compare the text in castmember A35 to see if it contains "frog."**
3. **Play the `correct answer` sequence if the text contains "frog," and the `incorrect answer` sequence if it doesn't.**

Adding an Event Script

A `keyDown` event script tests the keystrokes.

To add the first event script:

1. **Open the script window for frame 93 in the Script channel.**
2. **Type**
`when keyDown then checkKey`
and press Enter.

Event scripts always start with a `when` statement. In this script, `when keyDown then checkKey` gets executed as soon as the playback head reaches frame 93. Subsequently, whenever a key is pressed, a handler named `checkKey` will be executed. You'll write the handler definition in the next exercise.

Creating and Using a Handler

A **handler** is a named script that you define with the Lingo words `on` and `end`. Once you have defined it, you can use just the handler name in other scripts to execute the handler script.

To save you the trouble of typing the `checkKey` handler into the Movie Script, it has been entered for you in castmember A37. You can copy it from there and place it into the Movie Script.

To copy the pre-written handler:

1. **Open the castmember A37 script window.**

This is where a copy of the following text of the handler definition is stored.

```
on checkKey
-- keyDown handler
if the key = RETURN then
    if field "entry" contains "frog" then
        go to frame "correct answer"
    else
        go to frame "wrong frog"
    end if
    when keyDown then nothing
end if
end checkKey
```

In Lingo, every handler definition *must* begin with `on`, followed by the handler's name. Then enter the script statements that will be executed when the handler is called, and finally finish the definition with the `end` statement. Notice that the line following the `on checkKey` line is a comment that identifies the handler's purpose.

2. **Select all of the text in castmember A37 and copy it to the Clipboard.**

3. **Open the Movie Script window.**

Choose Movie Info from the File menu, then click the Script button in the Movie Info dialog box.

4. Paste the text on the Clipboard into the Movie Script, and press Enter.

That completes the handler definition and installation. The following section discusses the structure of the handler in more detail.

Testing For the Return Key in Scripts

The first line of the handler you created in the preceding exercise is a test for the Return key, placed in an `if...then` statement:

```
if the key = RETURN then
```

All of the following statements (everything between `if` and the last `end if`) will be executed only if the user presses Return.

Testing Text Strings

These lines test to see if the user typed in the correct text:

```
if field "entry" contains "frog" then
  go to frame "correct answer"
else
  go to frame "wrong frog"
end if
```

If the field `A35` contains the word "frog," then the playback head goes to the `correct answer` sequence, otherwise it goes to the `wrong frog` sequence.

Resetting an Event Script

An event script stays in effect until it is turned off. Earlier, you defined an event script that told Lingo what to do when the user pressed a key. When the `checkKey` is no longer needed, you must stop the event script from further action. The statement that turns off the `checkKey` handler is contained in the handler script itself, in the final `when keyDown` statement:

```
when keyDown then nothing
```

Here, the statement `when keyDown then nothing` replaces the previous `when keyDown` script.

Naming a Field

In addition to its number, castmembers can also have a name. The on checkKey handler refers to castmember A35 as field "entry". In order to do this, you must first name the castmember.

To give the castmember a name:

1. **Select castmember A35 in the Cast window.**
2. **Choose Cast Info from the Cast menu.**
3. **Type "entry" and click OK.**

Now this castmember can be referred to as cast "entry" or as field "entry".

More About go to Statements

The go command is essential for controlling where the playback head is located. In these examples, you need to add a script that returns the playback head to the name the animal segment after the incorrect answer animation has played.

To enter the go to script:

1. **Select frame 132 in the Script channel.**
2. **Choose go to frame "name the animal" from the script pop-up menu.**

Finally, you need to add a statement at the end of the correct answer animation that causes the playback head to return to the main menu.

3. **Select frame 151 in the Script channel.**
4. **Choose go to frame "main loop" from the script pop-up menu.**

Now is a good time to test what you've done so far (you can leave the Score window open if you want to see the playback head moving through the different sections). After clicking "Name the Animal", when

you type an incorrect answer and press Return, the `incorrect answer` animation should play, and when you type "frog" the `correct answer` animation should play.

Notice that right now the movie doesn't give you much time to look at the results of your text input. The next exercises will add controlling scripts that will fix this problem.

Timeouts

It's often useful to check whether a user has been inactive for a long time. For example, a long period of inactivity may indicate that the user is having difficulty. Maybe she isn't familiar with the mouse or has just gone away. If you want, you can have your movie do something after the user has stopped interacting with the movie for a given period. You accomplish this by setting a `timeOut` event script.

To create a timeout of twenty seconds:

1. **Drag a new marker to frame 154**
2. **Type**
`timeout sequence`
and press Return.

Next, add an event script at the beginning of the presentation to set the commands for `timeOut`. This will be part of the Movie Script, and, because it is inside the `on startMovie` handler, it will be executed when the movie starts playing (in response to the `startMovie` message). Like all event scripts, this one stays in effect unless and until you specifically deactivate it. In this case, you always want the timeout sequence to be active, so you don't need to worry about turning it off.

3. **Open the Movie Script window.**
4. **Type in the following `startMovie` handler:**

```
on startMovie
  set the timeoutLength to 20*60
  when timeOut then ↵
    go to frame "timeout sequence"
end startMovie
```


The length of the timeout is preset to 3 minutes. This means that the user would have to do nothing for 3 minutes before the timeout sequence would take effect. The second statement in the handler will shorten the timeout to 20 seconds.

The expression for twenty seconds is 20*60 rather than just 20 because Lingo measures time (using the Macintosh's internal clock) in ticks, each of which represents 1/60 of a second. Therefore, in order to specify seconds, you multiply the desired number of seconds by 60 (the asterisk is the multiplication symbol). You could achieve the same effect with the following script, but it's not as readily apparent that 1200 is the same as 20 seconds:

```
set the timeoutLength to 1200
```

5. Press the Enter key to close the Movie Script.

Now as soon as 20 seconds pass without any user action, the playback head jumps to the frame labeled `timeout` sequence, where the remedial instructions to the user appear.

Creating Custom Menus

You can use the `installMenu` command to create your own Macintosh-style pull-down menus.

To create a pull-down menu for *Basic Without*:

1. **Open the Movie Script window.**
2. **Place the cursor within the `startMovie` handler and press the Return key twice to open up space for the new command.**

Place the new command line before the existing lines, right after the `on startMovie` command.

3 Type

```
installMenu A36
```


This statement tells Lingo to look at the text that's stored in text castmember A36 and use the menu definition that was placed there earlier. You can put your own menu definitions in any text castmember; our choice of castmember A36 was arbitrary.

The menus that you create with the `installMenu` command appear at the top of the screen only after you start the presentation.

4. Press the Enter key to close the Movie Script window.

5. Choose Text from the Window menu.

You can see that the menu is defined as follows:

```
menu: Mymenu  
Same or Different ≈ go to frame "same or different "  
Name the Animal ≈ go to frame "name the animal "
```

The `menu:` command establishes the name of the menu. After that, the text that you type on each new line in the menu definition creates a new menu item. The "≈" sign (Option-X on the keyboard) does not appear in the menu item, but indicates that choosing the item executes the statement that is specified after the ≈.

If you want to add more menus, simply add another set of menu statements, either before or after existing ones. The menus will be installed and appear in the same order as you defined them.

Remember that your custom menus appear only while the movie is playing.

To test the installed custom menus:

1. Choose Rewind from the Control menu.

2. Press the Shift key and choose Play from the Control menu to begin the movie.

Your custom menu appears and you can try choosing items from it. Notice that when a custom menu is in place, you can stop the movie only with Command-W (or the period on the numeric keypad).

To add command key shortcuts to menu commands:

1. **Open the Cast Script window for castmember A36 and change the script as follows:**

Same or Different/S ≈ go to frame "same or different"
Name the Animal/N ≈ go to frame "name the animal"

By appending *"/S"* and *"/N"* to the Lingo statements that define the menu items, those letters are automatically shown with the Command key symbol in your menu, and those Command keys execute the same commands as do choosing the menu items themselves.

► **Note** There are other special symbols that you can use in defining menus (for example, to make items bold or disabled). Refer to the menu: keyword in Chapter 11 for descriptions of these symbols.

Troubleshooting and Debugging

If you see an error dialog box while running a movie, it means that Lingo can't execute the instructions you have written.

The Error dialog box allows you to:

- ◆ Click Script to display the script with the offending handler or script displayed, ready for editing.
- ◆ Click Cancel to close the dialog box.

You can click the Trace checkbox in the Message window to obtain a journal of every Lingo statement as it is being executed. Such a record of what is being executed and in what order is very useful in figuring out what is happening when something mysterious is causing things to go wrong.

To use the trace function:

1. **Open the Message window.**
2. **Click the Trace check box at the bottom on the Message window.**

3. Rewind and play your *Basic With* movie.

You'll see all the Lingo statements appear in the Message window as they are executed. Try clicking the buttons to see more statements being executed. Tracing a script this way can help you determine exactly what is being executed, in what order, and for how long, and makes it easier to track down the problem.

➡ **Note** The Trace feature slows down playback. Remember to turn it off when it's no longer needed. If Trace is checked, it will remain active even when the Message window is closed.

Using the Message Window for Testing

To use the Message window to test a one-line Lingo statement, type in the statement and press Return. Lingo executes the statement immediately.

If the statement is valid, the results are visible on the Stage, or in the Message window itself if you have used the `put` command, which forces a value to be displayed. If the script is invalid, the standard error dialog box appears.

You can test a handler by writing it in the Movie Script, then calling it from the Message window by entering the handler name.

You can also create objects from factories and XObjects in the Message window. For example, the following script operates the `serialPort` XObject:

```
put SerialPort(mNew,0) into modemPort
```

XObjects and factories are covered in greater detail in Chapters 5, 6, and 7.

The Tracing Symbols in the Message Window

Figure 2.1 shows a sample recording that was made by clicking the Trace box and then running a script for a few moments. Notice that Lingo has placed different symbols at the beginning of each line.

- ◆ The -- (double-dash) indicates that the line is a comment and will not be executed.
- ◆ The number shows the current frame.
- ◆ The > (greater than) pointer shows the nesting level of the script. > for the first level, >> for a script called by a script, >>> for a script called by a script called by a script, and so on.
- ◆ The == (double equal signs) indicate where the script (called by the preceding line) was found.

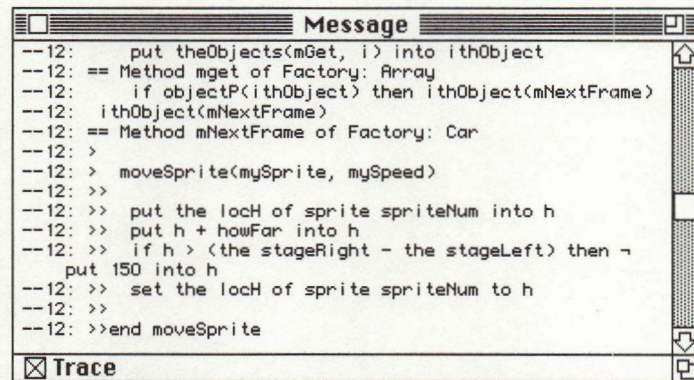


Figure 2.1 The Message window tracing symbols.

Figure 1 shows a sample recording that was made in Alaska. The data
was recorded on a continuous paper for a 10-minute period. The data was
plotted at 1000 samples per second. The beginning of each line

is the 1000th sample. The number indicates that the line is a component and will
not be recorded.

The number shows the component name.

The 1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The

The 1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The
1000th sample is the first sample of the component. The

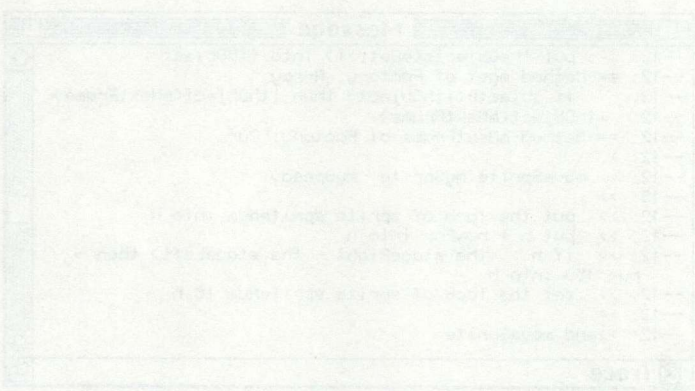
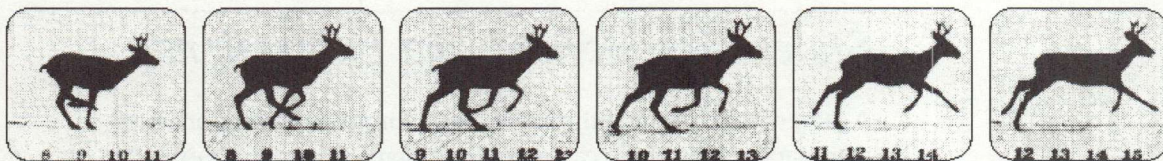


Figure 1. The Tachygraph recording of the Alaska Pipeline.



Chapter 3: *Script Parts and Structures*

This chapter discusses the various components of Lingo scripts and how they are put together. Many of these elements—such as arithmetic operators, logical operators, and if statements, for example—can also be found in many other programming languages. General discussions of keywords and commands that are specific to Lingo—such as puppets and Movie Scripts—are also presented.

Statements, Scripts, and Expressions

A **statement** is any valid instruction (a Lingo command or set of commands) that Lingo can execute. When a statement is executed, its instructions cause Director to perform some action.

A **script** is any statement or group of statements.

An **expression** is any part of a statement, meant to be taken as a whole, that returns a value. For example:

```
"this string" & " and another string"
```

is a legitimate expression, but is not a valid statement all by itself. This particular expression would return the following value:

```
"this string and another string"
```

A Lingo statement consists of a command and any values required to complete the instruction. For example:

```
go to frame 23
```

In this example, `go to` is the command, and `frame 23` is the value that the command requires in order to carry out the instruction.

Most statements can be completed in one line. For example, a script can contain an `if` statement that takes the following form:

```
if condition then result
```

Here, *condition* and *result* are placeholders for any valid expression. (For more information about `if` statements, see the “Tests” section later in this chapter.)

Some keywords can only be used in a multiline form. For example, the `repeat...while` test or keywords such as `on` and `end`, among others, require more than one line. The exact formatting for multiline statements is provided in the reference section.

Optional Keywords and Abbreviated Commands

You can write some Lingo statements in an abbreviated form. Abbreviated versions of a command are easier to enter, but may not be as readable as the longer versions. The `go` command is a good example. All the following statements are equivalent. The last one takes the fewest number of keystrokes.

```
go to frame "This Marker"
```

```
go to "This Marker"
```

```
go "This Marker"
```

If a command can be abbreviated, the acceptable abbreviations are shown in the Lingo dictionary.

Parentheses

Some Lingo functions *require* parentheses. The ones that do are clearly marked as such in the reference section. Parentheses are *always* required in the Message window.

Even though they are optional, you can use parentheses to override Lingo's **precedence**, or to make your Lingo statements easier to read.

Character Spaces

Words within statements are separated by spaces. Lingo ignores extra spaces, so you can put them in if you want to use them for visual formatting and readability.

In string literals (described in “Literals,” below), spaces are treated as characters. If you want spaces in a string, you must put them in explicitly. The string concatenator symbol `&&` will join two string expressions with a space inserted between them automatically.

Upper- and Lowercase Letters

Lingo is not case sensitive—you can use upper- and lowercase letters however you want. The following statements are equivalent:

```
Set the hilite of cast "Cat" to TRUE
```

```
Set the hiLite of cast "cat" to True
```

```
set the hilite of cast "Cat" to true
```

```
SET THE HILITE OF CAST "CAT" TO TRUE
```

```
Set The Hilite Of Cast "Cat" To True
```

However, even though Lingo is not case sensitive, we recommend that you follow script writing conventions, such as the ones that are used in this manual.

Comments

Comments are defined by double hyphens (`--`). You may place a comment on its own line, or following any statement. Lingo ignores any text following the double hyphen on the same line.

Comments can be anything you want: notes about a particular script or handler, or about a statement that might not be self-commenting. Commenting makes it easier for you or someone else to understand a procedure later after you’ve been away from it for awhile.

Literals

A **literal** is any part of a statement or expression meant to be taken at its face value, rather than some symbolic value. Each type of literal (string, integer, decimal, cast number) has its own rules. Examples and types of literals are described next.

Text Strings

You must enclose text string literals within double quotation marks. In this example, there are two string literals:

```
put "Hello" into field "fieldText"
```

Similarly, if you are testing a character string, each string must be surrounded by double quotes:

```
if "Hello Mr. Bob" contains "Hello" ↵  
then soundHandler
```

Suppose you create two editable fields that control variables `vUserEntry1` and `vUserEntry2`. The following script example returns the combination of the characters entered by the user, with a space between them:

```
Put "My thoughts amount to " & vUserEntry1 ↵  
&& vUserEntry2
```

Integers

MacroMind Director works with integers between -2,147,483,648 and +2,147,483,647. (An integer is always a whole number, without any fractional decimal places.) You enter numbers without commas, using a minus sign for negative numbers. Some Lingo commands and functions require a whole number argument that's within a given range. The requirements for specific Lingo words can be found in the Lingo reference in Part 4.

Floating Point Numbers

You can use decimal numbers in scripts. All that's required to make a number a decimal is to use a decimal point. You can also use exponential notation (for example: `-1.1234e-100` or `123.4e+9`).

Cast Numbers and Names

Cast numbers are treated the same as reserved words. Cast numbers always begin with letters A to H and are always followed by two digits; for example, `A11`, `C88`, `H23`.

Lingo can refer to bitmap images, buttons, text, palettes, and sounds in the Cast in one of three ways: either by literal cast number—`cast A11`, `cast C16`, and so on—by cast number relative to the first cast position (`A11`)—`cast 1`, `Cast 15`, and so on—or by cast name (if you've named it)—for example, `"newFigure"`.

You can add and subtract cast numbers in an expression to refer to different castmembers. For example, `cast (1 + 1)` returns `cast 2`. `cast (A11 +1)` returns `cast A12`.

If you have named more than one castmember with the same name and then use the name in a script, Lingo will use the first (lowest numbered) castmember it finds with the specified name.

Boolean Expressions

Boolean expressions return one of two values: `TRUE` or `FALSE`.

When you are writing Boolean expressions you can use the Lingo **constants** `TRUE` and `FALSE`, or the numeric values `1` and `0`. The following statements are equivalent:

```
set the hilite of cast A11 to TRUE
set the hilite of cast A11 to 1
```

This allows you to use expressions that return either characters or numbers to set Boolean values.

Assigning, Calculating, and Communicating Values

A **value** is any quantity assigned to a constant, variable, parameter or symbol. A value can also result from the computation of an expression. In your Lingo scripts you:

- ◆ assign values to variables (discussed next)
- ◆ get (read) values from variables and from constants such as `TRUE` or `FALSE`
- ◆ recalculate values based on combinations of expressions and operators
- ◆ test values to make branching or looping decisions
- ◆ represent values in expressions with literals, variables, or constants.

Variables

A **variable** is a named location in memory that holds a value. Values in variables are temporary. The value can be a whole number (integer), a decimal number (1.5, 678.90, and so on), a character string ("xyz", "this string" and so on), or the results of an expression that is, itself, calculated as it's needed. For example, in the expression $(5 + y)$, the value assigned to y may change depending on, say, user input. Each time $(5 + y)$ is calculated, the user-input value is used for the value of variable y .

You establish a variable (called "initializing") when you first assign it a value in a script or handler. You then make use of the variable by placing a new value in it, based on whatever criteria you want, and by using it in other expressions. After a value has been placed in a variable, the variable name, used alone, returns the value stored there.

If you place a new value in an integer or decimal variable, the old value is replaced.

A variable does not exist until you define it the first time in a script, handler, or in the Message window. How long a variable persists in memory depends on whether it's a global or local variable.

A **global** variable exists and retains its value for as long as Director is running. Global variables can be defined within a handler. Variables you define in the Message window are automatically global.

A **local** variable exists and retains its value for only as long as a particular handler is executing.

You assign a value to a variable with `set` or `put` as follows:

```
put "Gotcha!" into vCapture
set vCapture = "Gotcha!"
set vCapture to "Gotcha!"
```

These three statements do the same thing. The value placed in the variable (in this case "Gotcha!") can be determined with many kinds of expression. For example, suppose that you want the contents of `vCapture` to depend on whether the user clicks a particular sprite. The following script places one of three values in `vCapture`, including the value `EMPTY` if the user doesn't click either of the two specified sprites:

```
if the clickOn = 3 then
    put "Gotcha!" into vCapture
else
    if the clickOn = 4 then
        put "Almost Gotcha!" into vCapture
    else
        put EMPTY into vCapture
    end if
end if
```

Note that in this example, the word `EMPTY` specifies a string containing no characters.

Global Variables

You can define and set global variables within any script.

A global variable that you've defined in a handler can be referred to, and the value in it changed or used, by any other script or handler. You cannot define global variables in a Score script, however.

For example, if you want to use the user's name several times in the course of your presentation, you might establish a global variable in the movie script named `vName` that takes a value typed by the user at the beginning of the presentation. Suppose the editable text field that you want the user to type into is a field named "User Input." The following script creates a global variable that contains the name:

```
global vName
put the text of field "User Input" into vName
```

The `global` command can be placed in front of several variables to establish them all at the same time. The following statement defines three global variables:

```
global vName, vCapture, vHelp
```

If you define global variables in this fashion, using the `global` command in a single line, they are automatically initialized as empty or 0. To assign other values to them, use `set` or `put`.

You can display all current global variables and their current values with the `showGlobals` command in the Message window.

Local Variables

You can use a local variable in any script or handler. It is subsequently available only while that script or handler is being executed.

Any variable created in a handler or script that is defined without using `global` is automatically a local variable.

You can display all current local variables by using the `showLocals` command in the Message window. This command can only be used in the Message window.

Symbols

A symbol is a data type (like a string or other value) that begins with the pound sign (#). Symbols are useful because they can be returned from memory quicker than strings. For example, the symbol `#Steve` in the statement:

```
set userName = #Steve
```

will be returned from memory quicker than the equivalent string ("`Steve`") in this statement:

```
set userName = "Steve"
```

Constants

A constant is a named value whose content never changes. You refer to constants by their names. The constants `BACKSPACE`, `ENTER`, `QUOTE`, `RETURN`, and `TAB` let you put these characters in your scripts where needed.

Using set and put to Establish Values

The `set` and `put` commands let you define variables or change a **property** (attribute) of an object or of the Macintosh system:

```
set variable to expression
```

```
set the property to expression
```

The `put` command defines a variable or returns the value of an expression:

```
put expression into variableName
```

```
put expression
```

This second use of `put`, when used in the Message window, calculates the value of the expression and displays it in the Message window.

➡ **Tips & Hints** You can use `put` for debugging. Use a `put` statement at test points in a script to check the values during execution. While the movie is running, these values will be displayed in the Message window. When you are done debugging, remove the `put` statements.

You can also use the keywords `before` and `after` with `set` and `put` to change the contents of variables and fields that contain string values. Let's say, for example, that the variable `vText` contains the string value `"old text"`. Then:

```
put " new text" after vText
```

will result in the value `"old text new text"` in `vText`.

```
put "new text " before vText
```

will result in the value `"new text old text"` in `vText`.

► **Note** If you want spaces in text strings, you must put them in explicitly, as demonstrated in the previous example.

Handlers

A handler is a named statement or group of statements that you define in Movie or Cast Scripts. (You cannot define handlers in a Score script.)

Handlers that you define in a Movie Script are available from any other script while the movie is running. A handler that you define in a Cast Script is available only within that Cast Script. That is, you cannot call a handler in a Cast script from another Cast Script, or from Movie or Score Scripts.

You call a handler from another script by using its name in the calling script.

Handlers give you more flexibility with less work. Defining a handler that performs a frequently used set of actions is easier than copying or rewriting a given script every time you want to use it (assuming that you want to use it more than once).

Handlers can call other handlers. When the called handler stops executing, the handler that called it resumes.

You can define as many handlers in one script location as you want. It's a good idea to group related handlers in a single place, though, for ease of maintenance.

Defining Handlers

You define a handler with the `on` and `end` keywords.

A handler name is made up of alphanumeric characters only (no special characters or punctuation). A handler name must be one word—no spaces are allowed.

Handlers can initiate an action or return a result. You can write handlers that take arguments or **parameters** and either perform some action based on the parameters, or return a result based on a calculation or expression that uses the parameters. Handlers that return a result are sometimes called *function handlers*, or just *functions*.

The following handler is a custom command (you can try this out by putting the handler example in a Movie script):

```
on beepSome
  --generates a random number of beeps
  set x = random(6)
  beep x
end beepSome
```

Once this handler has been defined, the following statement in the Message window executes the handler:

```
beepSome
```

You may establish **arguments** for the handler following the handler name. Arguments are values (optional or required) that the handler uses to perform its work. Multiple arguments are separated by commas. Subsequently, when you use the handler in a script, you provide specific values for the arguments (called **parameters**) that the handler then uses to execute the script.

The format for defining a handler with arguments is:

```
on handlerName [argument1] [, argument2]...
  [... additional statements ...]
end handlerName
```

► **Note** In the preceding example, words in `typewriter` type are those elements that you enter exactly as shown. The words or phrases in *italics* are placeholders that describe the general parameter or argument for which you supply specifics. The square brackets [] enclose optional

elements that you include if needed. (You don't type the square brackets, though.) Optional elements may or may not change what a statement does. For more about these conventions, see Chapter 10, and Appendix A.

The following handler definition creates a function that takes two arguments, adds them together, and returns the sum:

```
on addIt a, b
    -- a and b are argument placeholders
    set c = a + b
    return c
end addIt
```

The calling script passes the numbers for `a` and `b`. For example:

```
Addit (8, 15)
-- here, 8 and 15 are the arguments
```

The result is 23.

You can use handlers as parts of expressions. For example, the following script establishes the value of a variable `d` and assigns it a value calculated by using the `addIt` handler (defined above):

```
set d = Addit (8, 15)
```

You can define function handlers that do not require arguments, yet still return some value. For example, the following function handler returns one of the string values `cat`, `dog`, or `ape`.

```
on someAnimal
    return item random(3) of "cat, dog, ape"
end someAnimal
```

► **Tips & Hints** Even when you define a function that doesn't require arguments, you must still use the parentheses when you call the function from another script. For example:

```
put someAnimal()
```

Event Scripts

Event scripts are executed in response to three main kinds of events: the press of a key, the click of the mouse, and timeouts.

Event scripts are global and remain in effect until you specifically deactivate them. For example, suppose you defined an action that you want to happen when the user clicks anywhere on the screen, throughout the entire movie. You can define an event script for the `mouseUp` message in the Movie script, for example.

➡ **Note** Event scripts will execute even if the user clicks on a button or sprite that has a Cast or sprite script, or both. The sprite script, if any, is executed first. Then the Cast Script, if any, is executed. Finally the event script will be executed. To stop the event message from being sent to the next level of execution, use `dontPassEvent` in the script where you want the message to be stopped.

Event scripts are assigned with the `when` keyword. See the `when` entry in the Lingo dictionary.

Using `pause`, `continue` and `delay`

If you want a presentation to stop at a particular frame, you can use the `pause` command.

➡ **Tips & Hints** The `pause` command will also stop the sound. If you want the sound to continue playing, mark the frame and use `go marker(0)`.

Typically you would place `pause` in the Script channel of the frame where you want the playback head to stop, and a `continue` command that undoes the pause effect in a Cast script of a sprite in the same frame.

A screen that provides the user with several choices is a typical example of how you might use `pause`.

You can put `pause` and `continue` in event script statements like this:

```
when mouseDown then pause
when keyDown then continue
```


The `pauseState` function tests whether a presentation is currently paused:

```
if the pauseState then continue
```

You can use the `delay` command to stop the presentation for a certain amount of time, measured in ticks (60ths of a second):

```
delay 3*60 -- 3 seconds at 60 ticks/second
```

This statement halts movie playback for three seconds. During this time, all events (mouse clicks, keystrokes, and so on) are ignored. Scripts continue to execute; `delay` merely stops the playback head from advancing.

The `delay` command counts from the time the frame is first displayed. Place the `delay` command in the Script channel cell or in a handler.

► **Tips & Hints** The `delay` command does not work with `repeat` while loops or within `when... then` tests. You can still halt the presentation in relation to these commands, however; see “Monitoring Time” later in this chapter.

Functions

You use functions to set or return values. The value can be based on arguments passed to the function, on the current state of the system (for example, the system date), or on properties of objects (text properties, for example). For a complete list of functions, see Chapter 11.

Reading and Setting Properties

A **property** is any attribute of an object (such as a menu, castmember, or field), or of the Macintosh system, that can have one of several settings. Lingo allows you to read and set properties for the system, for sprites, and for text objects. You use the `set` and `put` commands to change and return the values of properties. Note that you can't set all properties. Some property values can only be read.

A list of all properties is provided in the Lingo Quick Reference section at the back of this manual. Default values and value options are provided in Chapter 11, along with each property word listing.

Operators

Operators are symbols that tell Lingo how to combine or otherwise act on the values of an expression. You are already familiar with some kinds of operators, such as those used in arithmetic: `+`, `-`, `*`, `/`. Some operators make comparisons between two arguments (for example, `>`, `>=`, `=`). Other operators (`not`, `and`, `or`) are logical operators that join conditions.

The precedence order of an operator determines when it is performed. For example, multiplication is always performed before addition. You can use parentheses to change the order of operation:

$$2 + 4 * 3 = 14$$

$$(2 + 4) * 3 = 18$$

The operators, with their precedence orders, are described in the following sections. Operators with higher order numbers are performed first. Operators with the same order numbers are performed left to right.

Arithmetic Operators

Arithmetic operators perform addition, subtraction, multiplication, division, and other arithmetic operations. Parentheses and the negation sign are arithmetic operators.

Table 3.1 Arithmetic operators.

Operator	Effect	Precedence
()	Groups operations to control precedence order	5
-	Negation; reverses the sign of a number	5
*	Multiplication	4
mod	modulo	4
/	Division	4
+	Addition	3
-	Subtraction	3

Comparison Operators

Comparison operators are used to compare two values. They return a boolean true or false, depending on whether the comparison is true or false.

Operators unique to Lingo are `sprite...within` and `sprite...intersects`.

`sprite...within` uses the following format:

```
sprite sprite1 within sprite2
```

This expression returns `TRUE` if the bounding rectangle of *sprite1* is entirely within the bounding rectangle of *sprite2*.

`sprite...intersects` uses the following format:

```
sprite sprite1 intersects sprite2
```

This expression returns `TRUE` if the bounding rectangle of *sprite1* touches the bounding rectangle of *sprite2*.

The hot areas can be defined by the non-white areas (instead of the bounding rectangles), if the ink of the sprites has been set to matte (with both `sprite...within` and `sprite...intersects`).

Two text comparison operators can determine if one string appears within another, or if one string is at the beginning of another.

The `contains` operator tests whether one string appears in a second:

```
string1 contains string2
```

This expression returns true if *string1* appears anywhere within *string2*.

The `starts` operator tests whether one string begins a second:

```
string1 starts string2
```

This expression returns true if *string1* appears at the beginning of *string2*.

You can compare strings or numbers with all the other comparison operators. In the case of strings, "greater than" (>) means "later in alphabetical order." String comparisons ignore case.

Logical and String Operators

Table 3.2 Comparison operators.

Operator	Effect	Precedence
sprite...within	True if sprite 1 is entirely within sprite 2	5
sprite...intersects	True if sprite 1 touches sprite 2	5
<	Less than	1
>	Greater than	1
<=	Less than or equal to	1
>=	Greater than or equal to	1
=	Equal to	1
<>	Not equal to	1
contains	True if string 1 contains string 2	1
starts	True if string 1 starts with string 2	1

Logical operators compare boolean values (or expressions) and yield a boolean result.

For example, the `or` operator allows you to create a condition for a test that is true if the value meets either condition:

```
if vNumber <= 100 or vTime > 180 then set the  $\neg$ 
text of cast 14 to "Do you want some help?"
```

Table 3.3 Logical and string operators.

Operator	Effect	Precedence
not	Logical negation	5
and	Logical and	4
or	Logical or	4
&	Concatenation	2
&&	Concatenation with space	2

Testing and Control Structures

Lingo can evaluate conditions and proceed based on whether the evaluation is true or false. Such conditional operations are called **tests**.

For example, a test can evaluate the user's action (such as which sprite is clicked), so that the user's choice determines the next sequence. An

expression can evaluate a button selection, a sprite click, a text entry, or a numeric entry.

A conditional expression is one that evaluates to either true or false, depending on some state of affairs. For example:

```
if the text of cast "fieldText" contains ¬
  "Mickey" then trueScript
```

If `fieldText` contains the characters "Mickey" the statement is true; if it does not contain these characters, the statement is false. This comparison could be used to determine which sequence to play next, based on whether the user correctly typed the name of a famous mouse.

There are two basic types of tests: a test of an instance, and a test of a state. An instance test, typically used in an `if` statement, returns true or false based on some property of a sprite, or some other element in the movie. A state test, used in a `when` statement, returns true or false based on a user action such as typing a key or clicking the mouse.

Lingo continually checks such user actions; if a state test is in place, the test returns true when the user performs the tested action. Lingo evaluates instance tests only when it executes the script in which the test occurs.

if...then...else

The `if...then...else` statement tests a condition and executes one or more statements if the condition is true, and one or more statements if the condition is false. An `if` statement has the following structure:

```
if test then
  true script(s)
else
  false script(s)
end if
```

The test in the first line is either true or false. It can be any expression that evaluates as either true or false (`n > 100`, `value = "start"` for example). The result of the test evaluating true is a statement or set of statements. Likewise, the result of the test evaluating to false is execution of one or more statements.

The following example shows you how to order `if` statements in a series that provides the equivalent of a case statement:

```
if test then
  true script(s)
else if test2 then
  true script(s)2
else if test3 then
  true script(s)3
[ ... additional script statements ... ]
end if
```

Lingo evaluates the tests in order, and executes the first result linked to a test that evaluates as true.

Use an `end if` statement when you place the true script expression on its own line:

```
if test then
  true script
end if
```

Or

```
if test then true script
else
  false script
end if
```

A single-line script does not need an `end if` statement:

```
if test then true script
```

The following variations are also allowed:

```
if test then true script
else false script

if test then
  true script
  true script
  [...]
else false script
```


repeat while

A **repeat while** statement continuously repeats a command as long as a test associated with it continues to be true. This statement has the following structure:

```
repeat while test
  statements to be executed
end repeat
```

The test must be an expression that returns either true or false. You may include as many statements to be executed as you want (or none at all, if you prefer).

You use **exit repeat** to leave repeat statements. For example:

```
if the clickOn = 6 then exit repeat
```

In this example, if the user clicks the sprite in channel 6, the repeating statements are interrupted.

If the test cannot become false (which would result in an endless loop), and there is no exit, you will run into problems. In most cases, the only way you can stop such a loop is to press Command- (period).

► **Note** Do not use **repeat while** statements for delays. Delays are best controlled by using the timer, using a delay in the Tempo channel of the Score, or by adding extra frames to the Score. See “Monitoring Time,” later in this chapter.

repeat with

A **repeat with** statement repeats an operation for a specified number of times.

Suppose you wanted to set up an animation display that uses a sequence of the first ten castmembers for the animation. The following script will display the sequence:

```
repeat with n = 1 to 10
  set the castNum of sprite 3 to n
end repeat
```

when...then *and* Event Scripts

The `when` and `then` keywords let you define event scripts that tell Lingo what to do when a user has pressed or released the mouse button, typed a key, or when a certain amount of time has passed. The results of these tests can be used in other scripts. Once defined, an event script remains in effect until it is explicitly removed.

This section describes mouse and key event examples. The `timeOut` event is described in "Monitoring Time," later in this chapter. Handlers for other events (`on startMovie`, `on stopMovie`, `on stepMovie`, and `on idle`) are discussed in Chapter 11.

The format for these tests is as follows:

```
when event then action
```

The event can be any of the following:

```
keyDown  
mouseDown  
mouseUp  
timeOut
```

The *action* can be any statement, including one that performs a test or calls a handler or factory method. Here are some examples:

```
when mouseDown then go to frame "intro"  
  
when keyDown then if the key = Return then ↵  
    go to frame "ending"  
  
when mouseUp then basicHandler
```

If you want two or more operations to occur when an event happens, you must use the `when` statement to call a handler that performs the multiple operations. For example, if you want to set the location of a sprite to the location of the mouse pointer when the user presses the mouse button, you would use a handler:

```
on locationHandler  
    set the locH of sprite 9 to mouseH  
    set the locV of sprite 9 to mouseV  
end locationHandler
```


Now your event script can call the handler:

```
when mouseDown then locationHandler
```

Event scripts must be turned off when they are no longer appropriate. You do this with the `nothing` command:

```
when mouseDown then nothing
```

You use the word `nothing` when you want the results of a test to perform no action at all.

Overriding an Event Script

Occasionally you want to prevent an event script from executing (such as when the user holds down a modifier key during the event). You use the `dontPassEvent` command to intercept the event script.

For example, a regular `mouseDown` event causes Lingo to look for the mouse button being pressed and then, if a button is clicked, for that button's script to be activated. Suppose you want something else to happen (2 beeps) in the special case of the Option key being pressed when the button is clicked:

```
when mouseDown then if the optionDown then →
    beep 2
```

This statement will cause two beeps, but the button's regular script will then be executed. To prevent the regular script from executing, use a handler:

```
on mouseOption
    if the optionDown then
        beep 2
        dontPassEvent
    end if
end mouseOption
```

Activate the handler with a special `mouseDown` event script:

```
when mouseDown then mouseOption
```

With the `mouseOption` handler, the mouse event also tests whether the Option button is being pressed at the same time as the mouse button. If it is not, the regular `mouseDown` event sequence is initiated. If the

Option key is being pressed, the `dontPassEvent` command prevents the execution of the default `mouseDown` event.

The `dontPassEvent` command applies only to the specific case where it is used. It does not constitute a new state of affairs—subsequent events will still be passed to the proper handler. In other words, it doesn't have to be turned off.

Monitoring Time

You can use Lingo to monitor the length of time since events have occurred and to activate scripts based on the results.

Lingo provides two ways to monitor time:

- ◆ Using the timer
- ◆ Using the `timeoutLength` property

Both methods use the Macintosh computer's internal clock. Time is always measured in ticks (60ths of a second).

Using the Timer

You can use the `timer` to measure the time since Director was launched, or since the user last clicked or moved the mouse, or typed a key.

The timer begins counting when Director is launched. You can reset the timer with the `timer` property. Following are two different scripts that do the same thing:

```
startTimer  
  
set the timer to 0
```

Resetting the timer in this way resets the timers for the `lastClick`, `lastKey`, `lastRoll`, and `lastEvent` functions.

The `lastClick`, `lastKey`, `lastRoll`, and `lastEvent` functions measure the number of ticks since the user clicked the mouse, typed a key, moved the mouse, or did any of these. These functions can be used as tests in `if` statements:


```
if the lastEvent >= 15*60 then
    then go to frame "help"
```

This statement causes the playback head to go to the frame labeled “Help” when the user does nothing for 15 seconds.

Here is one way to specify a delay during a repeat while sequence:

```
on wait ticks
    startTimer
    repeat while the timer < ticks
    end repeat
end wait
```

The script that calls this handler would send the number of ticks as an argument. A tick is a 60th of a second.

Using when timeout

A timeout occurs when the user does nothing for a specified time. You use the `when timeout` command to establish the event script that is activated when such a timeout occurs.

Typically, you use the timeout feature of Lingo to display further instructions for a user who has stopped interacting with the movie.

You specify the length of time to wait before a timeout occurs with the `timeoutLength` property. The default setting is 3 minutes. The `timeoutLength` property is explored in more detail in the “Basic With” interactive movie section of Chapter 2.

The length of time since the user has done something is kept track of in the `timeoutLapsed` property. The `timeout` state returns true when the length of time in `timeoutLapsed` equals the value in `timeoutLength`.

The value of `timeoutLapsed` is reset to zero whenever the user does something, so it usually never reaches the value in `timeoutLength`.

Lingo continually tests whether a timeout has occurred. You can activate a script with a `when` statement:

```
when timeout then script
```


Or you can set a standard script to be executed when a `timeOut` occurs:

```
set the timeoutScript to "script"
```

The `timeoutLapsed` property is reset to 0 by a command, a keystroke, a mouse click, or the playing of a movie. You can restrict the setting of `timeoutLapsed` to any of these, or prevent any of these from resetting the function. For details, see the `when timeOut` command in the Lingo reference.

Turning Off Event States

It is important to turn off certain event states when you are finished using them, or when they become inappropriate. States that persist can cause problems if they conflict with later intentions. These states include event scripts, puppets, the `immediate` sprite property, and the `perFrameHook` property (used primarily for recording frame-by-frame to videotape).

Using `when...nothing`

To turn off an event script created with a `when` statement, use the `nothing` command. For example, if you have created an event script for a `keyDown` event, turn it off with:

```
when keyDown then nothing
```

The following statement uses the `set` command to do the same thing as the previous example:

```
set the keyDownScript to EMPTY
```

Other event scripts that must be turned off include `mouseDown`, `mouseUp`, and `timeOut`.

Setting States to False

States created by being set to true are turned off by being set to false. For example, to turn off a puppet:

```
puppetSprite 9, FALSE
```

(Puppets are discussed in "Sprites and Puppets" later in this chapter.)

Creating Custom Menus

You can create your own custom menus with Lingo so that, when a movie is running, the user can choose from items that you provide. When the user chooses an item, the script called by the item is executed.

■ **Tips & Hints** If you want a menu to be available throughout the running of a movie, define it in a Movie Script, inside the `startMovie` handler.

You create custom menu with the `installMenu` command, referring to a cast number where the actual menu definition is stored:

```
installMenu castNumber
```

The menu definition stored in the text `Cast` specifies the name of the menu (the menu title), followed by the lines that specify the commands to appear within that menu, and the script(s) to be activated when a command is chosen. Each menu definition can create several menus.

```
menu: menuName1  
  itemName ≈ script  
  itemName ≈ script  
menu: menuName2  
  itemName ≈ script
```

You enter the “≈” character by pressing Option-X.

You can create Command-key equivalents for each menu item. In fact, all the standard Macintosh menu definitions are supported, including checkmarks and enabled/disabled characteristics. The details for specifying these options are in the `installMenu` and `menu` entries in the Lingo reference section.

You can remove a custom menu by issuing the `installMenu` command without an argument.

Sprites and Puppets

Sprites are the basic elements of animation. A **sprite** is an instance of a castmember on the Stage, and consequently in a cell (or series of cells) in the Score. A sprite is ordinarily controlled by the Score.

Puppets are sprites that you can control with Lingo scripts from anywhere in the movie (a Movie script, a Cast script, and so on). Sounds, tempos, transitions, and palettes can also be puppets. One example of how you might use this feature is when you want a sprite to respond to data that's generated by a script statement, or by input from other sources, such as the Macintosh computer's serial port.

Any sprite property that you can control in the Score can also be controlled from a Lingo script once you've declared it to be a puppet, including the Cast number, the size, shape, and location of the castmember on the Stage, its color, ink, line thickness, and pattern, and its height, width, and type.

You can learn more about puppets from the "Simple Puppets" movie.

Handling Sprites

You can make sprites moveable or editable, and you can measure whether they intersect or lie within other sprites. The `moveableSprite` command must be attached to individual sprites in the graphics channels of the Score.

The `moveableSprite` Command

You can easily allow the user to move any sprite around the screen. Tests associated with that sprite's location can activate other scripts when the user moves the sprite.

You declare a sprite to be moveable by entering `moveableSprite` in the sprite's channel.

This command does not work when used in a handler, or in a script in the Script channel. It only works if it is attached to a sprite channel cell.

You can track the location of a sprite with the sprite properties `locH`, `locV`, `left`, `right`, `bottom`, and `top`. In addition, you can determine whether a sprite intersects another sprite, or is contained within it:

```
if sprite 6 intersects 7 then beep
if sprite 6 within 7 then set the text ↵
  of field "target" to "BINGO!"
```

The editableText Command

You can create a text sprite that can be edited by a user in two ways. A text sprite is created with the text tool from the Tool window. You can declare the field thus created to be editable by checking Editable in the Cast Info dialog box. You can also declare the field to be editable by attaching the command `editableText` to the sprite channel cell where the field is placed. When you check the Editable box, the field will always be editable. If you use `editableText` in a sprite cell, you can turn this property off again when it is no longer required.

Tests associated with the Cast number of the sprite's text can then activate scripts based on what the user enters in the field.

For example:

```
if field "userInput" contains "Harvard" ↵
then go to frame "ivy league"
```

Using Puppets

Any property of a sprite that you can control from the Score can be controlled by Lingo as well. To do so, you must make the sprite a puppet. Once you've declared it to be a puppet, a sprite can no longer be controlled by the Score. To return control to the Score, you must remove the puppet state from the sprite channel.

You declare a sprite to be a puppet with the `puppetSprite` command:

```
puppetSprite channel number , true/false
```


The following statement makes the sprite in channel 9 a puppet:

```
puppetSprite 9, TRUE
```

If you don't turn off a puppet when you are finished manipulating it, unexpected results can occur when you use that channel later in the presentation. The entire channel remains a puppet until you declare the puppet to be false:

```
puppetSprite 9, FALSE
```

Lingo provides a sprite property that acts exactly the same as the `puppetSprite` command:

```
set the puppet of sprite 9 to TRUE
```

You can think of "puppethood" and "nonpuppethood" as an attribute of sprites that you can test. For example:

```
if the puppet of sprite 9 = TRUE then go to →  
frame "new segment"
```

Setting Puppet Properties

The initial properties of a new puppet are established by the properties of the sprite in the frame in which it was declared to be a puppet. That is, the puppet's initial location, color, castmember and other properties are determined by the initial location, color, castmember and other properties of the sprite. Subsequently, you can use other scripts to change these properties.

► **Tips & Hints** If a puppet is declared in a frame which has no sprite (in the specified channel), then any sprites you subsequently place in that channel will also be puppets. Always declare a puppet in a frame that contains a sprite that has already been placed in the Score.

To allow the user to control the vertical and horizontal location of the sprite when pressing the mouse button, you can use the following statements:

```
puppetSprite 9, TRUE  
when mouseDown then locHandler
```

The `locHandler` handler would establish the location of the sprite:

```
set the locH of sprite 9 to mouseH  
set the locV of sprite 9 to mouseV
```


The puppetTempo, puppetTransition, puppetPalette, and puppetSound Commands

You can create puppets for the tempo, a transition, the palette, or the sound. You use these puppet commands when you want to control the sound, transition, palette, and tempo effects from scripts.

Typically, you would use these puppets to allow the user to choose from a group of options.

For example, the following statement sets the palette to a specific palette:

```
puppetPalette "Rainbow"
```

You make the sound, palette, tempo, or transition a puppet with the following commands:

```
puppetPalette
puppetSound
puppetTempo
puppetTransition
```

Each of these puppets has a slightly different syntax. For details, see Chapter 11.

Of these puppets, only `puppetSound` and `puppetPalette` can be turned off with statements such as the following:

```
puppetSound 0
puppetPalette 0
```

The `puppetSound` command turns off a continuous sound and returns control of the sound to the Sound channel to the Score.

Using and Controlling Buttons

Buttons, check boxes, and radio buttons are special purpose sprites that are used to present choices to a user.

When a user clicks a button, its state changes. A check box toggles between empty and selected (marked with an X). A radio button toggles between an empty circle and a bullseye. A button highlights on `mouseDown` and returns to its original state on `mouseUp`.

You can use Lingo to react to these button clicks. In addition, you can control the properties associated with buttons.

Creating Buttons, Check Boxes, and Radio Buttons

Director provides three tools in the Tool window for making the common Macintosh buttons, check boxes, and radio buttons.

To make any button:

1. **Choose Tool from the Window menu.**
2. **Click the tool you want to use: the check box, the button, or the radio button.**
3. **Drag a rectangle on the Stage.**
4. **Type the display text you want to appear on or next to the button.**
5. **If necessary, set the font, style, and size.**

The button is placed in the Cast as a button castmember. You add it to the movie by dragging it onto the Stage.

Lingo automatically takes care of the visual feedback that tells the user if the button has been clicked. All you need to do is provide the Lingo script that tells Director what to do when the button is clicked. You can change the visual state of any check box or radio button (selected or deselected) through a script.

Reacting to a Clicked Button

Consider a button placed on the Stage over several frames. If the user does not click the button, nothing associated with the button happens. If the user does click the button, any script associated with the button is executed when the mouse button is released (`on mouseUp`). If there is a `mouseDown` handler, it is executed when the button is pressed. You can put just about any script in the button, including one that calls other handlers defined in a Movie script or the button's own Cast script.

Reacting to a Clicked Check Box or Radio Button

Check boxes and radio buttons can be either selected or deselected. For Lingo these states are identified as `TRUE` or `1` for selected and `FALSE` or `0` for deselected.

In your scripts you test the state of the check box or radio button and then cause some action based on the result. In Lingo, you refer to a button by its cast name or number. For example, the following script tests the state of a check box or radio button in cast "Button Choice":

```
if the hilite of cast "Button Choice" =
  = TRUE then go to frame "new sequence"
```

The `hilite` property of a button stores the value determined by whether or not the user selected it. Each click on the button causes the value to switch between `TRUE` and `FALSE` (a `1` or `0`). In the preceding example, if the check is `TRUE` the script places the playback head in frame 10. If the check is `FALSE` nothing happens.

Selecting a Check Box or Radio Button

You do not need to depend on the user to set the state of a check box or radio button. For example, if you provide a set of radio buttons, you'll need to make sure that the user can only select one at a time. Your script will need to establish that all the buttons except the one selected are set to `FALSE`.

To cause a check box or radio button to be selected, you would use a `set` statement similar to this one:

```
set the hilite of cast "Button Choice"
  to TRUE
```

If you set the `hilite` property to `FALSE`, the check box or radio button is deselected.

The checkBoxAccess Property

By default a user can select or deselect a check box or radio button. There are two more restricted types of access:

- ◆ The user can only select the check box or radio button (can't deselect).
- ◆ The user cannot change the check box or radio button (cannot select or deselect).

To change the kind of access the user has to check boxes and radio buttons, you use the `checkBoxAccess` property. The following script prevents the user from changing *any* check box:

```
set the checkBoxAccess to 2
```

A `checkBoxAccess` setting of 0 allows the user to select and deselect check boxes and radio buttons. A setting of 1 allows the user to only select check boxes and radio buttons, and prevents a selected check box or radio button from being deselected.

⇒ **Tips & Hints** If the `checkBoxAccess` is set to 2, the user is prevented from changing the check state of a check box. If you want it to change, you need to provide a script that changes the setting.

The checkBoxType Property

By default a selected check box displays an X in the box. You can change this display to be a small black square within the check box, or to be a filled check box.

- ☒ one checkbox
- ☒ another checkbox
- ☒ a third checkbox

Fig. 3.1 Three ways to display a selected check box

You use the `checkBoxType` property to change this setting:

```
set the checkBoxType to 2
```

A `checkBoxType` setting of 0 displays a standard X check. A `checkBoxType` setting of 1 displays a small square within the check box. A `checkBoxType` setting of 2 displays a filled check box.

The Lingo Message Hierarchy

When Lingo is running a script and encounters a reference to a word, it looks for a definition in the following places and order:

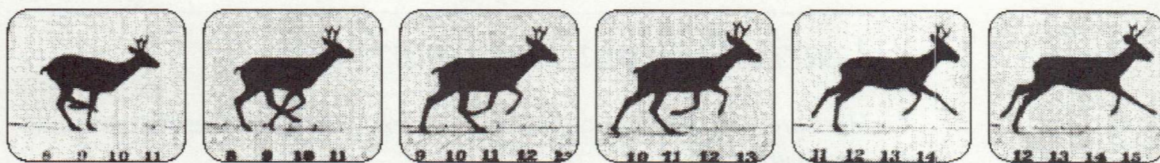
1. Object assigned to local or global variable in current script (the one in which the reference is contained)
2. The current Cast script (if any) or current Score Script.
3. The Movie script.
4. Factory definitions (XObjects and internal).
5. Macros that have been defined in a Text window.
6. The built-in function, keyword, and command list.

If a definition is not found in any of these places, an error message is displayed.

Duplicate Names

You can have handlers with the same name at different levels in the hierarchy. A handler named `myHandler` in a Cast script will be executed before a handler named `myHandler` in the Movie script, but both will be executed. (To prevent execution of the second handler, you would need to put `dontPassEvent` in the first handler definition.)

You can use handler names in Movie scripts that are the same as Lingo's built-in words, but if you do, you'll lose access to the built-in word. This is not recommended.



Chapter 4: The Apartment Sample Movies

A good way to understand how Lingo works is to look at movies that others have made. *The Apartment* is composed of a set of related movies that contain working Lingo scripts. These movies show you examples of most of the main features of Lingo. You can find them in *The Apartment* folder.

This chapter highlights selected examples. You're encouraged to look at and experiment with the others on your own. Each movie is self-contained and self-documented, and can be run by itself.

Learning From The Apartment

Here are some suggestions to help you explore the movies that make up *The Apartment*:

- ◆ Play each movie. Interact with it and observe how it works from a user's point of view.
- ◆ Stop the movie (Command-W) and look at the Score. Investigate the handlers that are defined in the Movie Script and castmembers. Notice how sprite scripts are used in frames where the castmember's script isn't appropriate.
- ◆ If you have a large screen, or two monitors, display the Score and the Stage windows at the same time, then play the movie so that you can follow the logic and the animation together.
- ◆ Likewise, open the Message window and, with Trace checked, play the movie and watch the lines of script as they are executed while the movie plays.
- ◆ Look at the text in the Text window.
- ◆ Experiment with variations of the handlers and scripts you find. You can temporarily disable a statement by entering two hyphens in front of the statement, thus turning it into a comment (this is called "commenting out" a statement).

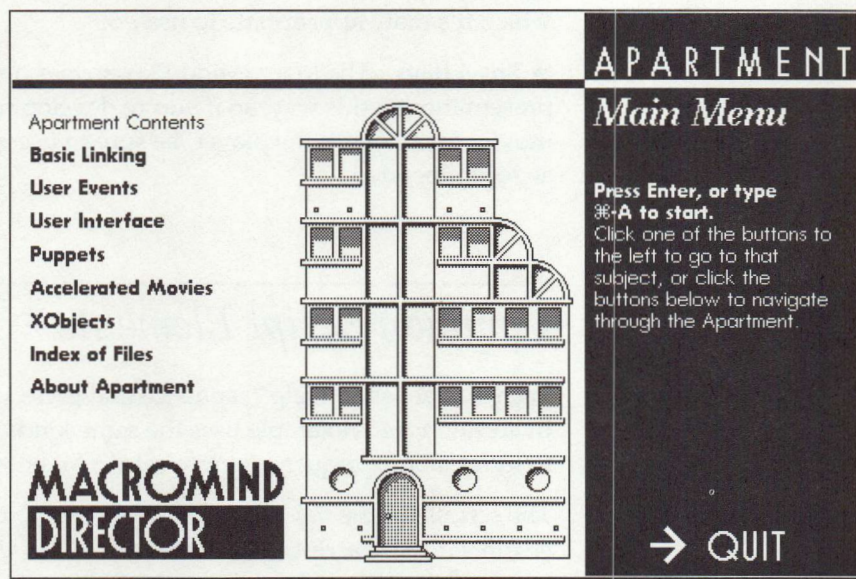
◆ *Main Menu*

The ◆ *Main Menu* movie is the primary document from which you run the other related movies.

Before continuing, make sure that you've installed the contents of the *Interactivity-1* and *Interactivity-2* disks to your hard disk. The remaining directions assume that you have started MacroMind Director, of course.

To start up the sample movies:

1. Open **•Main Menu**.



The **•Main Menu** movie provides a set of buttons that are linked, in turn, to submenus for different groups of examples. The **•Main Menu** is, itself, a good example of how to use a group of movie documents in concert with each other. Each button sends the playback head to a different movie or range of frames.

2. Click **Basic Linking**.

If you follow playback with the Score window open, you will notice that the scripts associated with each button open the appropriate Director document, but that they use the `go` command rather than the `play` command. The return arrow button in the examples issues another `go` command to return to the **•Main Menu** movie.

Why is this important?

Normally, we would recommend that you use the `play` command to start a secondary document. Then, when the secondary movie was finished, you would issue a `play done` command to return to the main movie.

Here, though, we've used `go` so that you can stop and examine any of the movies. If we had used `play done`, then, when you stopped a movie, Lingo would "forget" where to return. So this is a circumstance in which it's more appropriate to use `go`.

➡ **Tips & Hints** The MacroMind Player does not allow a user to stop a presentation in this way, so if you're developing a movie or group of movies for use with the player, be sure to use `play` and `play done`, as recommended.

Common Script Elements

Each movie has a "Help" sequence that gives you more information. In addition, each example uses the same kinds of navigation controls to move among sequences, or to return to the •*Main Menu* movie.

The scripts for the navigation controls are typically in high-numbered channels. These scripts return the user to the •*Main Menu* movie, or jump to a later or earlier portion of the current movie.

In general, scripts for on-screen buttons are best located in the Cast Script for that particular castmember. If the script for a castmember will change from frame to frame within a movie, you can override its Cast Script with a sprite script (a Score Script attached to a cell in a sprite channel).

In our typical examples we've used `go to frame "label"` to jump between segments.

The castmembers for the navigation controls—as well as the graphic elements common to all the *Apartment* movies—are not actually contained in each movie file, although they appear in bank E of the Cast window. They exist only in the special movie named *Shared Cast*.

Whenever you open a movie, MacroMind Director looks for another movie named *Shared Cast* in the same folder. If it finds one, any castmembers it contains in banks E, F, G, and H appear in the Cast window of the movie you opened.

The fonts DecoText and DecoMind have also been installed into *Shared Cast* so that they are available to all the *Apartment* movies.

Using *editableText*

The *editableText* example movie shows you how to manage the process of letting the user type a name, click OK (or press Return), and see the entered text fly up from the bottom of the screen.

This effect is achieved by declaring the text sprite into which the user types to be editable, using the `editableText` command. Then, the castmember is animated in a predetermined on-screen sequence.

The Script channel script for the first frame is:

```
put EMPTY into field "Name"
when keyDown then ↵
    if the key = RETURN then checkField
pause
```

After clearing the name field, this script establishes a `keyDown` event action which calls the `checkField` handler (which we'll discuss below) if the user presses the Return key. It then pauses to let the user type a name.

The sprite script `editableText` for sprite 3, the name field, makes its text editable by the user. The `editableText` command can be assigned to any text sprite that you've made with the Text tool. It has no effect on bitmapped text that you've entered in the Paint window. You may only use the `editableText` command in one of the numbered channel cells, with a text sprite. It has no effect when placed in any other script location. The text is editable only during playback of the frames to which the command has been assigned.

► **Tips & Hints** You can define a text field as being always editable by checking Editable Text in the Cast Info dialog box. However, by using the `editableText` command in the sprite channel, you can turn the editing properties on and off. Thus, in one part of a movie the user can input text, while in another part the sprite with its new text can be used in an animation which does not allow editing.

Now let's look at what happens when the user has finished entering the text.

The Cast Script for the OK button has a `mouseUp` handler:

```
on mouseUp
    checkField
end mouseUp
```

This handler calls `checkField` when the user clicks the button. (Remember that `checkField` is also called if the user presses Return.)

The `checkField` handler is defined in the Movie Script as:

```
on checkField
    if field "name" <> EMPTY then
        when keyDown then nothing
        go to frame "fly"
    end if
    dontPassEvent
end checkField
```

First the `checkField` handler checks whether a name has been entered. If so, it turns off the `keyDown` event script action, established in the first frame's script, and then jumps to the flying name animation. (Also, to prevent the Return character from being entered into the field, `checkField` uses `dontPassEvent`. You can read about this command in the Lingo Dictionary.)

In the sequence labeled "fly," the flying name is the same castmember as the entry text, but it is no longer editable. If we had checked `EditableText` in the Text Cast Info dialog box—instead of using `editableText` in the sprite script—it would be editable in all frames.

Animated and Moveable Sprites

The *Face Kit* movie lets you interact with two eyes and a mouth. The Score is essentially a series of loops that provide animated sequences, moveable sprites, and constrained moveable sprites, alone or in combination.

Clickable Sprites

The Clickable Sprites segment lets you click any of the images (each eye and the mouth) to animate it.

This segment starts by pausing in the frame labeled “click.” (This frame has the script `pause` in the Script channel.) Nothing happens until the user clicks one of the sprites.

The sequences for each eye and for the mouth are all similar. The following discussion concentrates on the left eye, but applies equally to the other sprites.

When you click the left eye, its sprite script—play frame “leftBlink”—sends the playback head to the animated sequence labeled “leftBlink”. The “leftBlink” sequence is ended with the Score Script `play done`. This returns the playback head to the frame from which the `play` command was given.

Notice that these sprites are not moveable. They use a simple sequence of frames to provide animation.

Moveable Sprites

The Moveable Sprites segment lets you drag images around the screen with the mouse.

These sprites are declared moveable with a `moveableSprite` script, placed in the sprite channel cells in the “move” sequence.

Note that the loop is set up with the `go to marker(0) + 1` Script channel script, which sends the playback head to the frame after the previous marker. This same script sets up the loops in the next two examples.

The `moveableSprite` command, like the `editableText` command, functions only when the playback head is executing those frames that contain sprites to which the command is attached. It has no effect when used anywhere else.

Animated Moveable Sprites

The Animated Moveable Sprites segment adds user-moveability to an animated sprite.

Notice that each sprite in the animated sequence has a `moveableSprite` script. This command allows the user to move the sprite no matter which frame is being played.

Constrained and Animated Moveable Sprites

The Constrained Animated Moveable Sprites segment adds a constraint to the previous example. The constraint limits the area in which you can drag the animated face parts.

The constraint is achieved with the `constraint` property, used in the `constrainFace` handler. This handler is defined in the Movie Script and is called by the Script channel script in the frame labeled “constrain”.

The result of the script is that the user cannot move the eyes outside the invisible rectangle in channel 4, and cannot move the mouth outside the invisible rectangle in channel 5. The constraint is on the registration point of each image.

➡ **Tips & Hints** Since the images that make up the eyes and mouth are bitmapped castmembers, you can reset the location of the registration points in the Paint window.

Notice that the `constraint` property is set to 0 to remove the constraint from each sprite with `unconstrainFace`. The Constrained Animated Moveable Sprites segment does this in the first frame of the “move” and “animate” sections.

The rollOver Movie

The *rollOver* example is a connect-the-dots game. You must connect the dots in their numbered sequence. The line connecting the dots appears

when you touch the correct number with the mouse pointer. There is nothing to click. This shows you how to use the `rollOver` function to test whether the mouse pointer is on a sprite.

This example also illustrates use of the `marker` function to control the order in which the user must touch the sprites.

When *rollOver* starts, the playback head stays in the frame labelled “start.” Its script is:

```
connectDot 1
```

This script calls the `connectDot` handler in the Movie Script and sends a value (1) as its argument. The entire handler definition is:

```
on connectDot dotNumber
    if rollOver(dotNumber) then
        go to marker(1)
    else
        go to marker(0)
    end if
end connectDot
```

This handler tests whether the user has moved the mouse pointer over the sprite identified by the `dotNumber` argument. If the user does roll the pointer over the sprite, the playback head moves to the next marker [`marker(1)`] and leaves the handler.

If the user does not roll the pointer over a sprite, the playback head remains at the current marker [`marker(0)`].

Each marked frame (“dot1,” “dot2,” “dot3,” “dot4,” and “dot5”) has a similar script. Each script sends its own corresponding argument to `connectDot`. As soon as the test for the proper sprite `rollOver` is true, the handler sends the playback head to the next marked frame.

The connecting lines are separate sprites in channels 7, 8, 9, 10, and 11. Each appears when the playback head is sent to its frame. For example, the effect of a line appearing between point one and point two is controlled by the sprite in channel 7, which is not displayed until the playback head moves to the marked frame labeled “dot2.”

The `marker(n)` function is a convenient way to move the playback head relative to its current location.

Note that the use of `marker(0)` keeps the playback head in the current frame, because the current frame is marked. If the current frame is not marked, `marker(0)` sends the playback head to the previous marked frame.

Basic Event Scripts

The *BasicUserEvents* movie provides a sampler of three event scripts—`mouseDown`, `mouseUp`, and `keyDown`—as well as another look at monitoring mouse motion with the `rollOver` function. Based on each action of the user, a message appears on the screen indicating what event has just occurred.

The event scripts for `mouseDown`, `mouseUp` and `keyDown` events are set up in the Movie Script's `eventActionsOn` handler, called by the Score script in frame 1, defined as follows:

```
on eventActionsOn
    when mouseDown then ↵
        put "The mouse button is DOWN." ↵
        into field "Button Status"
    when mouseUp then ↵
        put "The mouse button is UP." ↵
        into field "Button Status"
    when keyDown then ↵
        put "The last key typed was" && ↵
        QUOTE & the key & QUOTE & "." ↵
        into field "Key Status"
end eventActionsOn
```

The three `when...then` statements set event scripts that respond to system events: the user pressing the mouse button (`mouseDown`), releasing the mouse button (`mouseUp`), or pressing a key (`keyDown`).

These event scripts, once set up by `eventActionsOn`, remain in effect until they are turned off by `eventActionsOff`, which is called from the sprite scripts of the navigation buttons.

Notice that the `when keyDown` statement constructs the text of field “Key Status” by joining several strings together. Both the single ampersand (&) and the double ampersand (&&) are text operators that concatenate strings. The only difference between them is that the && operator adds a space character between the strings when it joins them.

The `key` function returns the last character typed by the user. The `QUOTE` character constant is used to display quotation marks around the character that was typed. It is necessary because the actual quotation mark character (") is used to indicate the beginning and end of a literal string, and thus can't appear inside one.

After the event scripts are established in frame 1, the movie loops on frame 2 because it has the following script in the Script channel:

```
checkCursor
go to marker(0) + 1
```

Each time the movie loops, this script checks the cursor position by calling the following handler in the Movie Script:

```
on checkCursor
  if rollOver(2) then
    put "The cursor is INSIDE the square." ↵
    into field "Cursor Status"
  else
    put "The cursor is OUTSIDE the square." ↵
    into field "Cursor Status"
  end if
end checkCursor
```

If `rollOver(2)` returns `TRUE`, the cursor is over sprite 2, the square. The `if...then...else` statement uses the `rollOver` result to display one of two messages in the “Cursor Status” field.

Simple Puppets

The Simple Puppets movie illustrates the use of puppet sprites and puppet sounds.

Puppet Sprites

In the first section of the movie, you can click the light switch to turn the light bulb on and off. You can also click the Shake It button to shake the bulb.

Both the switch and the bulb are puppet sprites. This means that they are controlled by Lingo scripts and not by the Score. For example, the bulb's random shaking is not a frame-by-frame animation in the Score. In fact, the entire section occupies just two frames of the Score.

The first one, labeled "sprites," sets up sprite 1 as the dark bulb and sprite 2 as the "down" switch. Its sprite script—`setPuppets TRUE`—turns these two sprites into puppets under Lingo control by calling the `setPuppets` handler (defined in the Movie Script) to issue the appropriate `puppetSprite` commands:

```
on setPuppets state
    puppetSprite 1, state
    puppetSprite 2, state
end setPuppets
```

The movie then loops on the following frame since the frame script is:

```
go to marker(0) + 1
```

When you first click sprite 2, the `mouseUp` handler in the Cast Script of the "Down Switch" castmember is executed:

```
on mouseUp
    set the castNum of sprite 1 to
        the number of cast "Lighted Bulb"
    set the castNum of sprite 2 to
        the number of cast "Up Switch"
end mouseUp
```


This handler changes the castmember displayed by each sprite so that the bulb is now lighted and switch is up.

The next click on the switch sprite executes the `mouseUp` handler in the Cast Script of the "Up Switch" castmember, since this is now the sprite's current castmember:

```
on mouseUp
    set the castNum of sprite 1 to
        the number of cast "Dark Bulb"
    set the castNum of sprite 2 to
        the number of cast "Down Switch"
end mouseUp
```

This just sets things back the way they were.

The cast script for the Shake It button defines four handlers: `mouseDown`, `shake`, `rnd`, and `breakFilament`.

It uses a `mouseDown` handler rather than `mouseUp` so that the bulb starts shaking as soon as the button is pressed. This handler loops using `repeat while the stillDown` to continue shaking as long as the button remains down:

```
on mouseDown
    repeat while the stillDown
        if shake(1, 5) then
            breakFilament
            exit repeat
        end if
    end repeat
end setPuppets
```

The real work is done by the `shake` handler, which is defined as follows:


```

on shake whichSprite, howFar
    put the locH of sprite whichSprite into h
    put the locV of sprite whichSprite into v
    put rnd(howFar) into deltaH
    put rnd(howFar) into deltaV
    set the locH of sprite whichSprite to h + deltaH
    set the locV of sprite whichSprite to v + deltaV
    updateStage

    set the locH of sprite whichSprite to h
    set the locV of sprite whichSprite to v
    updateStage

    if abs(deltaH) = howFar  $\neg$ 
        and abs(deltaV) = howFar then
        return TRUE
    else
        return FALSE
    end if
end shake

```

This handler performs one shake of a specified sprite (`whichSprite`) by a random displacement up to the specified amount (`howFar`).

To do this, it first stores the sprite's current position (`locH` and `locV` properties) in the local variables `h` and `v`. It then generates random horizontal and vertical displacements between `-howFar` and `+howFar`, storing them in the local variables `deltaH` and `deltaV`. (It uses our `rnd` function handler, which calls Lingo's random function.)

To make the sprite appear to shake, `shake` moves the sprite to its new, randomly displaced position, updates the Stage, moves it back to where it used to be, and updates the Stage once more.

Finally, it returns `TRUE` if both the horizontal and the vertical displacements were the maximum specified by `howFar`. In this case, the Shake It button's `mouseDown` handler calls a handler called `breakFilament`, which dims the bulb and castigates the user.

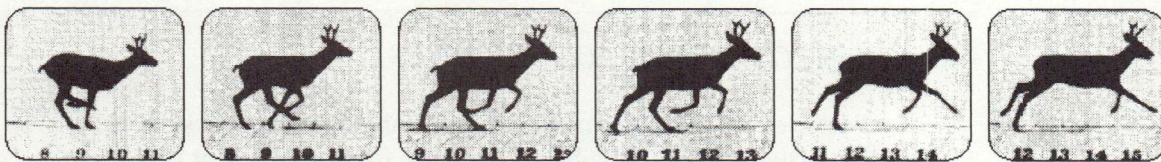
Remember to call `puppetSprite n, false` when you no longer want to use a sprite as a puppet. In this section, this happens when the appropriate navigation controls call `setPuppets FALSE` in their sprite scripts.

Puppet Sounds

The second section of *Simple Puppets* illustrates how you can play sounds under script control, without putting anything in the Sound channels.

The first two buttons have `mouseUp` handlers in their Cast Scripts which use the `puppetSound` command to play two different sound castmembers. The third button's `mouseUp` handler calls `puppetSound 0` to turn the sound off.

The first sound is a play-once sound which stops after playing through. The second one is a looping sound which continues to play until the command `puppetSound 0` stops it. (It even continues to play if you return to the •*Main Menu* movie without clicking Stop the Sound.)



Chapter 5: Factories

A **factory** is a particular kind of Lingo script in which you define your own **objects**. Once you've defined an object within the factory, you can subsequently call on the factory to create as many **instances** of the object as you want throughout a movie.

This chapter describes factories, their advantages and how to use them. The next chapter shows you two working factories, contained in the *Simple Factory* movie.

Introduction to Factories

Factories are useful for making more than one of an item such as bouncing balls or flying birds. Let's say, for example, that you are designing an interactive game. The game lets the user decide a level of difficulty from one to ten. A level of one means that the user will be able to zap one animated alien object. A level of ten lets the user do battle with ten animated alien objects.

If you didn't use a factory to create your aliens, you'd have to create ten different sequences in your movie, one for each level of difficulty, and place one alien sprite in the first segment, two in the second, and so on. When an alien is hit by user-fire, you want it to explode and disappear from the screen. Furthermore, you want all your aliens to move randomly around the screen and every once in awhile to fire phaser shots of their own at the user. In the ten-alien non-factory case, you'd have a lot of things to keep track of in your scripts: alien coordinates, hits or misses, how many aliens are left, and so on. Things get complicated quickly. The better way is to use a factory.

In our example, you would write an "alien factory" for the game. In the factory, you'd define what an alien is, how it behaves, what messages it can respond to, and any other characteristics required to specify the object and its overall behavior. When the movie is running, the factory will "build" aliens in response to the user's input, as many as needed for a given game. When an alien is "hit" the alien object is removed from the screen and from memory. This ability to create and destroy objects dynamically, "on the fly," is an important characteristic of factories.

Furthermore, each alien object can track its own movement and whether or not it has been hit. All the aliens can respond to the same set of messages. (In our alien game, for example, you might create a `move` message handler, or a `disintegrate` message handler.) Each alien is called an "instance" of the object. Each instance of our alien can have its own data (such as its current screen position).

➡ **Note** Even though our examples in this and the following chapter are associated with a graphic castmember that you can see on the screen, a factory object does not have to have a visual component. You can create

objects that reside in memory and perform other functions, such as controlling multiple videodisc players.

Factories, then, allow you to create more efficient, compact, and straightforward scripts. You can define as many different kinds of objects in a factory as you want.

Recall that in your Lingo scripts so far, you've used the `on` command to define handlers. The handler name defines the message to which the handler responds. In factories, you also define the messages to which an object can respond. These are called **methods**, and the rules for defining a method are slightly different from those you use to define a handler.

■ **Tips & Hints** You can also use a factory to create and manage an array, to control a series of related objects with the same methods and functions.

Factories are usually written in the Movie Script.

Objects and Messages

Factory objects communicate with other Lingo scripts through their messages. In a typical Lingo script, a message might be sent as follows:

```
put PioneerLaserdisc(mNew, 0) into videodisc  
videodisc(mStopAtFrame, 22500)
```

The first line of this script creates an object, here called `videodisc`, by sending the message `mNew` to the `PioneerLaserdisc` factory. In this example, the `0` argument indicates that the videodisc player is connected to the modem port of the Macintosh computer.

In the second line, the message `mStopAtFrame` is sent to the newly created object. The object's `mStopAtFrame` method will stop the videodisc player at a particular frame.

Assume that you attach a second videodisc player to your Macintosh. You can create a second object using the same factory:

```
put laserDisc(mNew, 1) into secondDisc  
secondDisc(mStopAtFrame, 22500)
```

At this point, you have two devices which are synchronized, and positioned at the same frame, 22500.

The syntax shown in this example is used by Lingo scripts to communicate with Lingo objects. Every object has a set of methods that it provides to Lingo scripts. In the line `videodisc (mStopAtFrame, 22500)`, the *object* is `videodisc` and the first argument, `mStopAtFrame`, is the *message* being sent (the same as the name of the specific method being called).

In addition, each method can have a set of arguments that it expects to receive. If the method expects arguments, these follow the the method name. In this case, the argument `22500` is passed to the method `mStopAtFrame` in object `videodisc`.

When you write your factory, you can specify how the objects created by the factory will receive messages, using methods and their arguments. The object can receive real-time input from a variety of sources: mouse or keyboard, a sequence of predefined data from a text castmember, or interactive input from one of the Macintosh serial ports.

How Factories Are Defined

You define a factory in a movie script. The definition always begins with the `factory` keyword, followed by the factory name:

```
factory factoryName
```

➡ **Tips & Hints** In previous versions of Director, factories could be created only in text castmembers. The current version continues to let you define a factory this way, but the recommended way is to put them in your Movie Script.

The factory name uses alphanumeric characters (no special characters or punctuation marks). A factory name can be only one word; no spaces are allowed.

The statement containing the `factory` keyword and its name is followed by a series of method definitions, defined by the `method` keyword:


```

method messageName1 [arg1, arg2...]
    statements
end messageName1

method nextMessageName [arg1, arg2...]
    statements
end nextMessageName

```

► **Note** In the preceding example, words in *typewriter* type are those elements that you enter exactly as shown. The words or phrases in *italics* are placeholders that describe the general parameter or argument for which you supply specifics. The square brackets [] enclose optional elements that you include if needed. (You don't type the square brackets, though.) Optional elements may or may not change what a statement does. For more about these conventions, see Chapter 10, and Appendix A.

Factories also make use of **instance variables**, defined by the *instance* keyword and discussed next.

Instance Variables

In Lingo scripts outside of factories there can be two kinds of variables: global and local. Global variables remain in existence for the duration of a movie. Local variables only exist while the handler or script that created it is being executed. Factories can use local and global variables, and can include a third kind of variable: instance variables. A factory can assign instance variables to specific objects. Instance variables contain a unique set of values specific to each individual object, even though the variables have the same name. The methods of a factory use the instance variables.

An instance variable is available only to the object with which it is associated. The value of an instance variable is established when the object is created, or when a method is used to change it. Each instance variable and its value persists as long as the object itself persists.

To define an instance variable, you must use the *instance* keyword, otherwise the factory will assume it is a local (temporary) variable.

You would typically define all your instance variables in the `mNew` method of a factory, the method that creates new objects. Subsequently, the values of instance variables can be changed by other methods.

For example:

```
method mNew parameter1, parameter2
  instance vName1, vName2
  set vName1 = parameter1
  set vName2 = parameter2
end mNew
```

Here, the first line defines two parameters that will be used to pass values to two variables. The second line defines two instance variables. The third line sets the initial values for the two variables.

The perFrameHook Property

The `perFrameHook` property, when attached to an object, causes an interrupt to occur at every frame when the playback head advances. When this interrupt occurs, the specified object calls a special message called `mAtFrame`. You define in a factory what actions the `mAtFrame` method performs. This is a much simpler way of calling a script that needs to be executed every frame (`perFrameHook` is especially useful when recording to videotape frame-per-frame). For specific information about `perFrameHook`, see Chapter 11.

Creating Objects From Factories

After you've defined a factory, you can use it to "build" as many instances of the factory's objects as you want.

An instance of an object is created by calling the `mNew` handler of the factory. The object can then use any of the factory's handlers for sending messages and determining new values.

Objects are created with the name of the factory and the `mNew` method:

```
put myfactory (mNew, arg1, arg2,...) into myobject
```


Special Methods in Factories

Three special predefined methods are available to every object, and do not need to be defined within your factories. These are: `mPut`, `mGet`, and `mDispose`. These are described next.

Creating and Using Object Arrays

The `mPut` method places values in an array. Every instance of a factory object automatically has an array associated with it. In fact, you might create a factory just so you can use its built-in arrays as containers. Arrays are useful for containing a “variable number of variables”—a number of values at various locations within the array.

Here is the syntax for `mPut`:

```
objectname(mPut, n, value)
```

The `mPut` method places value at location *n* in an array. The value *n* must be an integer that is equal to or greater than zero. Subsequently, you can use `mGet` to retrieve the value. *Value* can be any value, a number, a string, or another object.

`mGet` retrieves a value from the array. The syntax is:

```
put objectname(mGet, n) into variable
```

The `mGet` method returns the value at location *n* in an array created with `mPut`. Once again, *n* must be an integer that is equal to or greater than zero.

Removing an Object From Memory

Another method that is automatically available for every object is `mDispose`. Like `mPut` and `mGet`, `mDispose` does not need to be defined in your factory script before it can be used.

`mDispose` deletes an object from memory. The syntax is:

```
objectname(mDispose)
```


The `mDispose` method removes the object `objectname` from memory. Use `mDispose` to free up memory when an object is no longer needed.

The `me` Keyword

Factory objects can also call their own methods by using the `me` keyword. `me` is equivalent to the name of the object whose method is being called. In the example here, the `animateBird` method is called from within the `lakeScene` factory:

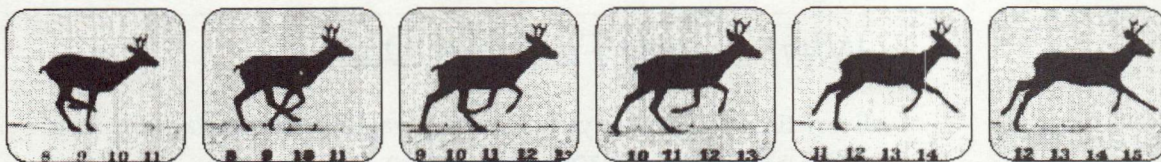
```
factory lakeScene
  method mNew ...
  ...
end mNew

  method animateBird startV, startH, speed
  ...
end animateBird

...

  method fly
  ...
  me(animateBird, 100, 100, 2)
  ...
end fly
```

The `me` keyword is useful when you want to call the same method with different objects. This way, you don't have to specify the individual object's name each time you call that method. See the *Simple Puppets* movie in the *Apartment* folder to see how the `me` keyword is used.



Chapter 6: The Simple Factories Movie

This chapter guides you through an examination of two examples of factories, contained in the *Simple Factories* document. These examples will give you a better idea of how factories are put together; how the factories are then used to make multiple objects; and how methods and instance variables are used. You will find it helpful to follow along with the *Simple Factories* movie open and running as you read this chapter. You'll find *Simple Factories* in the *Factories* folder. Take a moment to start the example before you continue.

What Simple Factories Does

Every time you click anywhere on the graphic of the car factory, a car object is created and placed on the screen.

Every time you click anywhere on the graphic of the house, it creates a jogger object and places it on the screen.

You can then set the car objects and jogger objects moving by clicking them. You can even change the speed on the car objects.

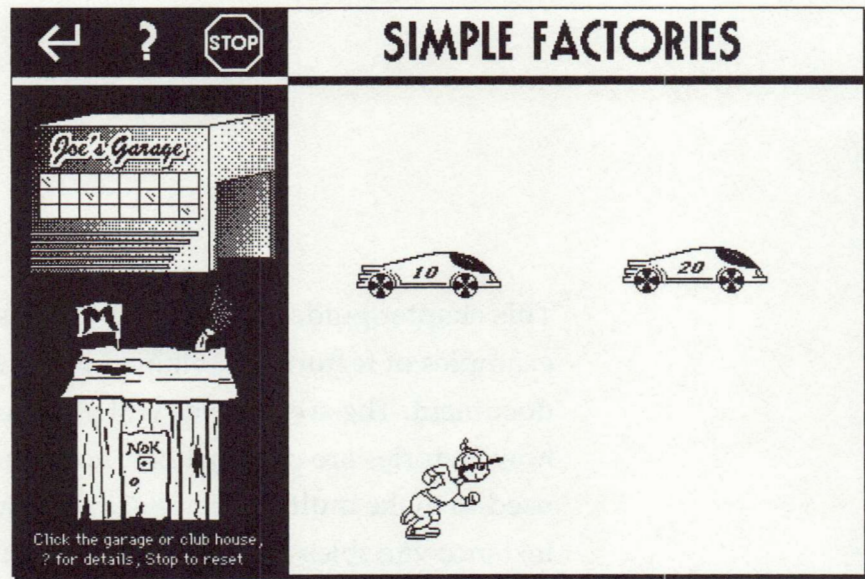


Figure 6.1 The Simple Factories Movie

The Elements of Simple Factories

The *Simple Factories* example is made up of the following elements:

- ◆ a control button to start the action of the example, and an associated initialization routine in the Movie Script
- ◆ two clickable sprites which use factories to create new objects
- ◆ the objects themselves which are created each time the user clicks one of the aforementioned sprites
- ◆ a control button to stop the movie, and an associated clean up routine in the Movie Script

The following sections examine each of these elements in more detail. The two factory castmembers are similar to one another. Only the car factory is discussed in this chapter. You can examine the other one on your own.

Establishing the Environment

The scripts that establish the visual elements and instructions for movement include the Go button and a handler that initializes the environment.

The Go Button

The Go button is at the upper left of the Stage. The Go button's Cast Script sets up the action of the factory example with the statements in its `mouseUp` handler:

```
on mouseUp
    cleanup
    initialize
    go to frame "start"
end mouseUp
```


This handler includes two messages — `cleanup` and `initialize` — which call custom handlers in the Movie Script.

The initialize Handler

The `initialize` handler sets up the environment for the demonstration:

```
on initialize
    global maxSprites, objectCount, theObjects
    put 10 into maxSprites
    put 0 into objectCount
    put Array(mNew) into theObjects
end initialize
```

The `initialize` handler sets up three global variables that are used in other handlers and methods throughout the movie.

The variable `maxSprites` is set to 10 because Score channels 1 through 10 are available for displaying cars and joggers. It doesn't change and simply serves as a constant.

The variable `objectCount` is used to keep track of the total number of objects—either cars or joggers—that the user creates. It is initialized to 0 since the movie starts off with none. Each time the user creates an object, `objectCount` will be incremented by 1. The variable `theObjects` is used as an array to hold each car or jogger object that the user creates. The first object created will go into slot 1 of this array, the next into slot 2, and so on. Note that we can't know in advance what combination of cars and joggers the user will choose to create—or in what order. Fortunately, each slot of a Lingo array can hold any Lingo data type—integer, floating-point number, string, symbol, or object.

The array `theObjects` is itself an object, created by sending the message `mNew` to another factory called `Array`. The `Array` factory is defined later in the Movie Script:

```
factory Array
```

This factory does not need any new methods of its own, because it uses only the predefined methods, `mNew`, `mPut`, `mGet`, and `mDispose`.

Creating the Objects

Once the `initialize` handler has established the proper environment, you can create new objects by clicking one of the two graphic sprites at the left. The top sprite creates car objects; the bottom sprite creates jogger objects. Each of these objects has its own Cast Script with a `mouseUp` handler and a factory definition.

The car factory is castmember A31. Select this castmember in the Cast window, then open its Cast Script window. (Choose Cast Info from the Cast menu, then click Script in the Cast Info dialog box.)

The mouseUp Handler

The car factory's Cast Script has two sections. The first section is the `mouseUp` handler:

```
on mouseUp
    global maxSprites, objectCount, theObjects
    if objectCount < maxSprites then
        put objectCount + 1 into objectCount
        theObjects(mPut, objectCount, Car(mNew))
    end if
end mouseUp
```

The `global` declaration lets the handler access the `maxSprites` parameter, the `objectCount` counter, and the `theObjects` object array which were defined in the Movie Script's `initialize` handler.

This is followed by an `if` condition test. If there are still channels available for new sprites, the handler increments `objectCount` and puts a new car object into the `theObjects` array. This is accomplished with an `mNew` call to the Car factory. The Car factory is defined in the rest of the Cast Script, as follows.

The mNew Method

The Car factory is made up of five methods: `mNew`, `mNextFrame`, `mClick`, `mAccelerate`, and `mDispose`. The first method, `mNew`, is the one where the new car objects are actually created:


```

method mNew
    global objectCount
    instance mySpeed, mySprite
    put 0 into mySpeed
    put objectCount into mySprite

    set the puppet of sprite mySprite to TRUE
    set the castNum of sprite mySprite to
        to the number of cast "StoppedCar"
    set the locH of sprite mySprite to 250
    set the locV of sprite mySprite to
        to randomInRange(125, 175)
end mNew

```

As with the `on mouseUp` handler example, the `global` declaration lets the method access variables that were initialized in the Movie Script. This is followed by the declaration of two instance variables:

```
instance mySpeed, mySprite
```

These instance variables will be available to each car object created using the `Car` factory. The variable names `mySpeed` and `mySprite` serve to remind you that these instance variables belong to the objects themselves. The next two lines initialize the instance variables.

Each new car's `mySpeed` variable is initialized to 0, meaning that it starts out parked. We'll see shortly how it can change to be 10, 20, 30, 40, 50, or 60. It's important to understand that each car object that the user creates has a variable called `mySpeed`. If you create two cars and start them moving, one car could have `mySpeed` equal to 20 while the other has `mySpeed` equal to 50.

The `mySprite` variable stores the number of the channel sprite the car should control as a puppet. For example, if it is the fourth object that has been created, `objectCount` will be 4, so sprite channel 4 will be used to display this car.

The next line in the `mNew` method sets the `puppet` property of the sprite representing the car object so that it can be controlled by Lingo. The last three lines initialize the car sprite to use the cast member named "StoppedCar" and position the car on the screen.

Movement From Frame to Frame

In addition to handlers that you name, define, and call whenever you wish from other scripts, the Movie Script can contain four special handlers—`startMovie`, `stepMovie`, `stopMovie`, and `idle`—that Director itself calls at specific times.

When a movie starts, Director looks for a `startMovie` handler in the Movie Script and executes it if one exists. It calls the `stepMovie` handler each time the playback head moves to a new frame. When the movie stops, it executes the `stopMovie` handler. Finally, it constantly calls the `idle` handler when there is nothing else for it to do.

The stepMovie Handler

In this movie the `stepMovie` handler is used to animate the objects that the user creates and to detect when the user clicks them.

```
on stepMovie
    global objectCount, theObjects
    repeat with i = 1 to objectCount
        put theObjects(mGet, i) into ithObject
        if objectP(ithObject) then ↵
            ithObject(mNextFrame)
        end repeat
    end stepMovie
```

This handler doesn't actually do much. The `repeat with` loop simply uses `mGet` to retrieve, one by one, each object that has been stored in the `theObjects` array. (Each object is retrieved into the same local variable, `ithObject`.) Then, after making sure that it is really an object, `stepMovie` tells it to execute the `mNextFrame` method in the factory that created it. The real work is done by `mNextFrame`.

It is important to realize that the `stepMovie` handler doesn't care whether the objects are actually cars or joggers. Nor does it know how *their* `mNextFrame` method will respond. This is up to the method definition in the factory that created the object.

The mNextFrame Method

Since we are concentrating on the Car factory, let's look at its `mNextFrame` method. If you investigate the Jogger factory on your own, you'll see that it also has an `mNextFrame` method. It's not exactly the same, however, because the joggers need to behave differently than the cars from frame to frame.

```
method mNextFrame
    moveSprite(mySprite, mySpeed)
    if the mouseDown and the clickOn = mySprite then
        me(mClick)
    end mNextFrame
```

The first statement calls the `moveSprite` handler in the Movie Script. The instance variables `mySprite` and `mySpeed` are passed as arguments to the handler to tell it which sprite to move and how far to move it. If `mySpeed` is 20, for example, the car will move 20 pixels to the right each frame.

The definition of `moveSprite` is:

```
on moveSprite spriteNum, howFar
    put the locH of sprite spriteNum into h
    put h + howFar into h
    if h > (the stageRight - the stageLeft) then
        put 150 into h
    set the locH of sprite spriteNum to h
end moveSprite
```

This handler simply moves the sprite that is specified by `spriteNum` to the right, by adding the `howFar` argument to the sprite's horizontal coordinate `locH`. If the sprite reaches the right edge of the stage, its horizontal coordinate is reset to 150. This makes it reappear on the left side of the Stage.

The `moveSprite` handler is in the Movie Script—rather than in the Cast Script of the car factory (castmember A31)—because it has enough general usefulness that the jogger factory also calls it. A Cast Script can only call a handler contained in itself or in the Movie Script; it can't call one contained in another Cast Script.

Manipulating the Objects

In addition to animating the objects from frame to frame, we need to detect when the user clicks them. The second statement of the `mNextFrame` method checks if the mouse button is down over the sprite being controlled by this object:

```
if the mouseDown and the clickOn = mySprite →  
    then me(mClick)
```

If the mouse button is down, the statement calls the object's `mClick` method using the `me` keyword to refer to itself.

```
method mClick  
    if the optionDown then  
        me(mAccelerate, -10)  
    else  
        me(mAccelerate, 10)  
    end if  
end mClick
```

The `mClick` method defines how the car responds to clicks: it accelerates by 10 when you click it, and decelerates by 10 when you Option-click it. The `mAccelerate` method, called using the `me` keyword, is also defined in the `Car` factory:

```
method mAccelerate speedChange  
    put mySpeed + speedChange into mySpeed  
    if mySpeed < 0 then put 0 into mySpeed  
    else if mySpeed > 60 then put 60 into mySpeed  
    set the castNum of sprite mySprite →  
        to the number of cast "StoppedCar" →  
        + mySpeed/10  
end mAccelerate
```

The `mAccelerate` method changes the speed of the car as determined by the argument `speedChange`, constraining it between 0 and 60 inclusive. This method then changes the `castNum` property of the car to indicate the speed on the car itself. (Look at the cast members in the range from A51 through A57.)

Cleaning Up and Exiting

Finally, when the user has finished with the movie and decides to stop, a set of clean-up-and-exit scripts lets the user leave gracefully.

The Stop Button

When the movie is running, one of the control buttons at the upper left of the stage is the Stop button. This button sets up the action of the factory example with its `mouseUp` handler.

```
on mouseUp
    cleanUp
    go to frame "intro"
end mouseUp
```

The `mouseUp` handler has two lines. The first one calls the `cleanUp` handler, the second sends the playback head back to the frame labeled "intro."

The cleanUp Handler

The `cleanUp` handler is defined in the Movie Script:

```
on cleanUp
    global objectCount, theObjects
    if objectP(theObjects) = FALSE then exit
    repeat with i = 1 to objectCount
        put theObjects(mGet, i) into ithObject
        if objectP(ithObject) then
            ithObject(mRelease)
            ithObject(mDispose)
        end if
    end repeat
    put 0 into objectCount
    theObjects(mDispose)
end cleanUp
```


This handler gets rid of the objects which were created by the `Car` and `Jogger` factories and placed into the `theObjects` array. The second line in the handler uses the `objectP` function. If the `theObjects` array has not yet been created, then no cleanup is needed, so the `exit` command is used to leave the method.

The `repeat with` loop then goes through the array of objects. For each object, it uses the predefined `mGet` method to assign it to the local variable `ithObject`. Next it makes sure that the object really exists. If so, it tells the object to execute its factory's `mRelease` method:

```
method mRelease
    set the puppet of sprite mySprite to false
end mRelease
```

This method relinquishes Lingo control of its sprite by setting the `puppet` property to `FALSE`. Finally, it disposes of the object itself using the predefined `mDispose` method.

► **Tips & Hints** If you define a method called `mDispose` in a factory, it will be called instead of the predefined one. This isn't usually what you want, since the object will not actually get disposed. If an object needs to do some housekeeping before it gets disposed, you should use a different name for the cleanup method (such as `mRelease` above). After the objects have been cleared, the object count is reset, and the `theObjects` array is also cleared using the `mDispose` method.

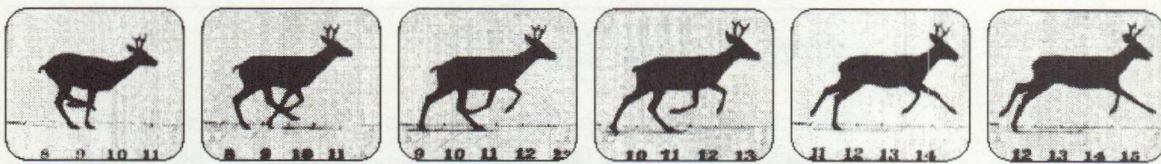
When the `cleanUp` handler has finished its work, the status of the movie is returned to the condition it was in before `Go` was clicked, with no objects in existence.

This matter goes on a... which was... by the... and...
... and... the...
... the... of... in... and...
... the... of... in... and...
... the... of... in... and...

The... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...

The... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...

The... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...
... the... of... the... was... by... and...



Chapter 7: About XObjects

The main use for XObjects (external objects) in Lingo is to allow third party developers and advanced users to add their own code objects to the language. There are XObjects available for controlling various kinds of videodisk players, for example. Internal math and database XObjects are also available. This chapter describes XObjects and provides information that you can use to create your own.

What Are XObjects?

XObjects are compiled code modules that Lingo can access. These modules are like Lingo factories (see Chapter 5), in that they create instances of their objects in memory at the time they are called, and there can be more than one copy (instance) of an XObject in memory at the same time.

➡ **Tips & Hints** XObjects are stored in the resource fork of a movie document. You can see them if you open the document with a utility such as ResEDIT. XObject code modules are stored as XCOD resources.

➡ **HyperTalk Users** XCOD resources in Lingo are similar to XFCN and XCMD resources in HyperTalk insofar as they are compiled and stored with the file's resources. An XObject called `XCMDGlue` is provided with Director (in the *movieName* document). `XCMDGlue` allows you to use appropriate HyperCard XCMDs and XFCNs from within Lingo scripts. They must be XCMDs and XFCNs that control non-HyperCard-specific operations (which is to say, they aren't oriented toward card manipulation, stack sorting, and so on). Any XCMD or XFCN that is highly HyperCard specific might not be appropriate to use with `XCMDGlue`. To learn how to use `XCMDGlue`, see the next chapter.

XObjects are capable of creating multiple copies (instances) of themselves in memory (XCMDs and XFCNs are not). This approach lets you maintain only one copy of an XObject, thus saving a lot of disk space as well as avoiding the confusion that could result from having to refer to multiple individual disk copies of an XObject.

XObjects **drivers** are designed to let the Macintosh communicate with external devices and **NuBus** cards (videodisks, CD-ROM drives, video-in-window cards). XObjects can also extend Lingo's control over the Macintosh Toolbox (windows and pop-up menus, and the like).

XObject messages can automatically pass back any information needed by Lingo or external devices.

➡ **HyperTalk Users** Note that Lingo's XObjects enable you to send messages to Lingo, without requiring the explicit callback routines that are often needed by HyperCard XCMDs and XFCNs. XCMDs and

XFCNs are procedurally based and require explicit routines to handle communicating with HyperTalk or external devices.

An XObject's methods are built into the XObject's compiled code. This means that you can't actually see the scripts that comprise each method. This is different from a factory's methods: you can see the method definitions in your Lingo script. Even so, you can "open up" an XObject and view a list of its methods by typing the following in the Message window and pressing Return:

```
XObjectName (mDescribe)
```

Here, *XObjectName* is a place holder. You type in the actual name of the XObject whose methods you want to list.

`mDescribe` is a built-in message that you can send to any XObject. Note that the XObject must be open before you can send the `mDescribe` message. (See `openXLib` and `showXLib` in the Lingo Reference as well as "Viewing XObjects" and "Opening XObjects" in this chapter.)

You can write an XObject to address a specific external device or a generic group of devices. For example, you could use an XObject to control only a specific brand of videodisk player. The limitation of a specific XObject is that you have to be absolutely certain that it will only be used with a particular external device. But, if you wanted to control a variety of videodisk players, you would need a more generic XObject, capable of using a variety of standard communications signals. The drawback of a generic XObject is that you might not be able to use features unique to a particular external device.

Characteristics of XObjects

After you have created an object that understands one set of messages, you can easily create yet another object that not only understands the original set of messages, but also adds additional messages (or commands) to Lingo's vocabulary.

The syntax you use to access XObjects is the same that you use to control an internal factory-generated object (see Chapters 5 and 6 for more about factories), for example:

```
put ballGenerator(mNew) into ball12  
ball12(mBounce, 3, 28)
```

This example starts a routine that causes an animated ball to bounce. The XObject `ballGenerator` first uses the `mNew` method to create the object `ball12`. The `ball12` object then uses the `mBounce` method with two arguments (3 and 28) to start the ball bouncing.

This basic syntax shown in the above sample statements is the way all Lingo objects communicate with Lingo scripts. Every object method has a different set of arguments that it expects, and each argument of the message is plugged directly into that method. Note, however, that you can also write methods that just perform some action and don't require arguments.

The arguments are the "ports" through which the object communicates with the rest of Lingo. You can establish a variable for any argument and define that variable by means of real-time input from the mouse, a sequence of predefined data, or other kinds of interactive input.

Writing Your Own XObjects

If you want to use XObjects that others have written, you don't need to know how to use a high-level programming language. But, if you want to define your own XObjects, you should know a programming language such as Pascal or C.

If you are interested in writing XObjects, contact MacroMind Customer Service to order an *XObject Developer Kit*.

Where XObjects are Located

XObjects are stored as XCOD resources and can be placed in three places:

- ◆ in the MacroMind Director 3.0 program and Player
- ◆ in a MacroMind Director 3.0 document (including a *Shared Cast* file)
- ◆ in a separate file (such as a resource file or stack)

To add or remove XObjects, you have to use a resource editor like ResEdit or a similar tool.

A group of XObjects stored together in a file is called an **XLibrary**.

Several standard XObjects are already built into the MacroMind Director program in an XLibrary called **Standard.xlib*. These are *FileIO*, *Panel*, *SerialPort*, *Window*, and *XCMDGlue*.

If a document that comes with MacroMind Director uses an XObject that is not built into the program, it is stored in a resource file located in the document's folder.

Viewing XObjects

XObjects that are installed in the resource fork of a document, in the MacroMind Director program, or in the MacroMind Player do not need to be explicitly opened. They are automatically loaded whenever that file is opened. To see which XLibraries are currently open, type the following script into the Message window and press Return:

```
showXlib
```

This causes a list of all open XLibraries to display. To see which XObjects are contained in a particular XLibrary, use this script:

```
showXlib XLibraryname
```

This causes a list of all the XObjects stored in the specified XLibrary to display in the Message window. If you are using the *XCMDGlue* XObject, the *showXlib* command displays all XCMDs and XFCNs that are in a stack (or other file) you have opened with *openXlib*.

Opening XObjects

XObjects stored in separate files are not automatically opened. You must use the following statement to explicitly open the file:

```
openXlib XObjectName
```

You can use `openXlib` to open XCMDs and XFCNs that you want to use with the `XCMDGlue` XObject.

Closing XLibraries

To remove an XObject from memory, you must close the XLibrary it came from. To remove an XLibrary from memory, use this statement:

```
closeXlib XLibraryname
```

If you don't specify a name, all open XLibraries are closed.

Looking at an XObject's Methods

You can display a summary of the methods in each XObject, including the method names and the arguments (if any) that each method requires.

To display an XObject's summary, use the following statement:

```
XObjectName (mDescribe)
```

The `mDescribe` method is automatically included for each XObject. The result displays all methods that the given XObject has available to it.

The `mDescribe` method is useful as a quick reference to an XObject's methods and arguments. For more complete XObject documentation, see the document that is associated with the XObject. The scripts used with each XObject are fully commented, and the `About` text castmember provides further documentation.

Common XObject Methods

In addition to the `mDescribe` method, most XObjects usually include these three methods: `mNew`, `mDispose`, and `mName`.

- ◆ `mNew` causes the XObject to create a new object with its own set of messages (methods).

- ◆ `mDispose` causes an object created by an XObject to be removed from memory.
- ◆ `mName` returns the name of the XObject that created the object.

The following section describes how to use these methods.

Creating an Instance of an XObject

The specific messages for using an XObject vary from one XObject to another. All XObjects, however, share a common message passing syntax.

To create an instance of an XObject in memory, you must first load in the XLibrary that defines the XObject, (this assumes that it is not one of the XObjects built into the MacroMind Director application):

```
openXlib XLibraryname
```

Once you have opened the XLibrary that contains the XObject, you use `mNew` to create a new instance of an object, like this:

```
put XObjectName (mNew, arg1, arg2,...) into objectName
```

objectName is a variable that refers to the instance of the object you have created. Both *arg1* and *arg2* are optional arguments that are only needed if `mNew` specifically requires them to pass parameter information to the object. To create different objects, use `mNew` message with a different *objectName*.

Once you have used the `mNew` message to create an instance of an object, you manipulate it by sending it messages from other methods in the XObject. To send an object a message (command it to do something), you use a script with these components:

```
objectName (messagename, arg1, arg2,...)
```

Once again, depending on the method, additional arguments may or may not be required.

For example, the following script would display a string that contains the name of the XObject that created the object *objectname*:

```
put objectname (mName)
```

The `mName` method is included in most XObjects.

Removing an XObject From Memory

When you have finished using a particular object, it's a good idea to remove it from memory. For example, if you have been controlling a videodisk player with an XObject and you no longer will be using the player, you should remove the object that was controlling the player. Usually a method called `mDispose` removes an object from memory:

```
objectname (mDispose)
```

Like `mPut` and `mGet`, `mDispose` does not need to be defined in order to be used. It is a built-in, always available command.

`mDispose` deletes the object from memory. The syntax for `mDispose` is as follows:

```
objectname (mDispose)
```

The `mDispose` method removes the object *objectname* from memory. Use `mDispose` to free up memory when the object is no longer needed.

XObjects Ready to Go

MacroMind Director comes with a wide range of XObjects that are ready for you to use. These can be found in *The Apartment* folder examples. You don't need to know all the specifics of creating the scripts to use these XObjects, because the examples supplied provide the basic features you might need. To use these XObjects, you need to:

1. **Copy the scripts from the example document into your document.**

Most of the XObject scripts have been placed in one text castmember of the document, so that they are easy to copy and paste into your own document.

2. **Copy the XObject file that is in the same folder as the example document into the folder where the document you are working with is located.**

This resource file contains the actual XObject. The XObject is usually stored in a ResEdit™ file and contains *XObj* in its name. You then:

3. Copy the handler definitions to the appropriate scripts in your movie document.

For example, if you want to build videodisk player control into your document, you would open the *Videodisk* example in *The Apartment* and copy the castmember that contains all of the videodisk player handlers. After pasting the text castmember into your document, assign the `vPlay` handler to your Play button (or to a frame in the Script channel). Thereafter, this button will start your videodisk player. It's not necessary that you fully understand the `vPlay` handler. All you need to know is that it causes the videodisk player to play. Similar handlers are supplied for initializing, stopping, searching and rewinding a videodisk player.

Each XObject has its own set of capabilities that is controlled by specific methods. All of the information you need to know about each XObject example can be found in the About text castmember in each document.

Note that most of the XObject examples require an `init` handler that sets up the conditions necessary for proper operation. Also, most of the XObjects display an error message if the XObject is not properly set up or if the external device (if any) is not properly connected. Usually the `init` handler is placed in the Movie Script.

If you want to learn more about the logic of each script, or even to create your own scripts, you should study the example XObjects.

XObjects Included with MacroMind Director

The following are brief descriptions of each example document that uses one or more XObjects. For specific documentation regarding a particular XObject, see the commented script and the About text castmember in the example document.

Note that *The Apartment* movies can provide detailed information describing the basic functionality of an XObject. Most of the documents are examples of actual applications that an XObject (or several XObjects) could be used for.

Many of these examples require that specific hardware be connected to your Macintosh computer and will not function properly otherwise.

XObject Examples in The Apartment

There are a number of useful XObjects contained in *The Apartment* movies supplied with MacroMind Director. Some of the more useful ones are described here.

CD-ROM Audio

CD-ROM Audio is an example of how you can use the AppleCD SC™ CD-ROM player to play disks containing CD-Audio music. The playback is controlled by the AppleAudioCD XObject.

The AppleAudioCD XObject lets you to control the sound stored on an audio compact disk that is played on the AppleCD SC CD-ROM player. Since the playing of music from this device is independent from the Macintosh's normal processing, you can use the CD-ROM player to play continuous sound even when movies are being loaded into memory. You can even play standard sampled sounds and sound effects from MacroMind Director's Sound channel while the CD-ROM player is simultaneously playing.

File I/O

File I/O demonstrates the basic functionality of the `fileIO` XObject: reading, writing, and appending text files. *File I/O TextFile* is used as an example text file which can be read and appended to.

The `fileIO` XObject lets you read and write text files to and from disk. Notice in particular that this XObject lets you open and create text files using the standard directory dialog box. You can also specify that a particular file be opened or written to, thereby bypassing the standard directory dialog box. See "Saving Text" below.

PopUp Menu

PopUp Menu uses the *PopMenu* XObject to show how you can create several different kinds of Macintosh-style pop-up menus.

Saving Text

Saving Text illustrates how you can read, edit, and write data to and from an external text file (using the *fileIO* XObject). It also shows several ways to view and print data. “Saving Text Data” is an example text file which you can read and edit. *Saving Text Test* is a text file which illustrates which castmembers are actually used for each field and record. See “File I/O” above for more.

Videodisk

Videodisk shows how you can control a videodisk player using the *Laserdisk* XObject. The *Laserdisk* XObject is a generic videodisk XObject; that is, it is designed to be used with a range of Pioneer and Sony videodisk players. This means, however, that it does not take full advantage of all the features each videodisk offers. See “Video Playback” below for an XObject that specifically addresses the features of particular Pioneer videodisk players.

Windows

Windows demonstrates how to create Macintosh-style windows. The *Window* and *Panel* XObjects let you create fully functional Macintosh windows. The *Window* and *Panel* XObjects work together. *Window* defines the size and type of the window, while *Panel* specifies the contents of the window (buttons, radio buttons, check boxes, graphics, and so on).

When you use the *Window* and *Panel* XObjects it is helpful to use a factory as a window handler to manage user interaction with the window. It is also helpful to use the special methods *mPut* and *mGet* (always available to any factory) to create an array that keeps track of which windows are open. This simplifies the process of closing any open windows when quitting from your interactive application.

XCMDGlue

XCMDGlue demonstrates how you can use the *XCMDGlue* XObject to use HyperCard XCMDs or XFCNs within MacroMind Director. The “*XCMDGlue*” stack contains the XFCN that is used in this example.

XCMDGlue is meant to work transparently. Usually, once you open an XCMD or XFCN, you can use the syntax of the XCMD or XFCN in your Lingo scripts just as you would in your HyperTalk scripts. Sometimes, you will need to supply the XCMD or XFCN with additional information for it to work properly. This is accomplished by setting up a callback handler.

CD-Audio Sound

CD-Audio Sound uses the Apple AudioCD XObject. This example demonstrates how you can cause an audio compact disk to start playing from a script in the Script channel (using the AppleCD SC CD-ROM player). This is useful in self-running presentations where you don’t want the user to have to start the CD-ROM player. Notice that animation can occur while the CD-ROM player is playing.

You should especially note that the audio continues playing even while another document is loading. This is not possible with sounds played from the Sound channel of MacroMind Director.

FramePerFrame

FramePerFrame is an application that you can use to record movies frame-per-frame to certain intelligent videotape recorders (8mm, S-VHS, 1/2”, 3/4”, 1”, and so on). *FramePerFrame* uses several XObjects.

The PopMenu XObject is used to choose the particular XObject (DiaQuestXObj or ARTI) that controls frame-per-frame recording. These XObjects let you record images by controlling hardware devices (supplied by DiaQuest® and ARTI®) that in turn are capable of controlling a variety of videotape recorders. Frame-per-frame recording is a very useful technique if you want to animate high-quality 32-bit images.

The FileIO XObject is used in a clever way to choose which movies are recorded to videotape. FileIO lets you choose files with the standard directory dialog box. Once you have selected a file, you can return the name of the file with one of FileIO's methods. This name is then passed to the frame-per-frame XObject to be recorded.

Also notice in particular that each XObject uses the `perFrameHook` property to cause the `mAtFrame` method to execute each time the playback head advances to a new frame or subframe (a subframe is one step of a transition). The ability to detect subframes allows you to record MacroMind Director transitions to videotape.

Video Playback

Video Playback uses multiple XObjects. This application uses a pop-up menu (created with the PopMenu XObject) that lets you choose a video device. The video devices supported in this example are videotape and videodisk players.

The PioneerLaserdisk XObject (in conjunction with the SerialPort XObject) lets you control the features of Pioneer laserdisk players. The `vtrXObj` XObject controls several consumer videotape decks (8mm and S-VHS). The ARTI XObject controls a wide range of videotape devices.

Once you choose a video device, controls specific to the device are displayed (play, stop, pause, fast forward, rewind, etc.). You also can think of this example as a library of video playback controls.

Video-in-Window

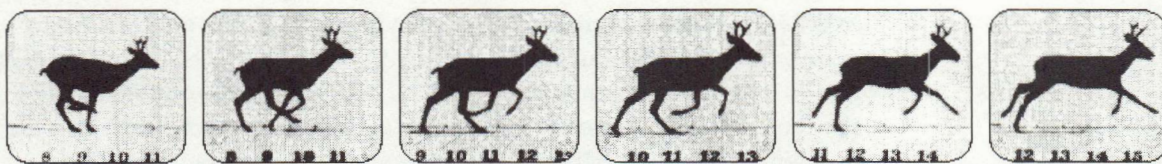
Video-in-window boards display images from a video source (such as a videodisk or video camera) within a variable-sized window on a Macintosh computer display. Depending on the specific board, the video image can be manipulated to produce different effects (such as flipping or scrolling).

This example uses a videodisk (controlled by the PioneerLaserdisk XObject) as a video source for the MASS Microsystems™ ColorSpaceFX™

board. ColorSpaceFX is a video-in-window and special effects board that allows a video source to be displayed in real-time within various "windows" on a Macintosh display. In this example, the windows are defined by QuickDraw rectangles set to a certain key color. The ColorSpaceFX board is controlled by the XCMDGlue XObject which reads the ColorSpaceFX XCMD.

Multiple Devices

This document is an example of MacroMind Director's power to combine multiple media and external devices. Start by playing an audio compact disk on a CD-ROM player by means of the AppleAudioCD XObject. You can then control a videodisk and a videotape player with the PioneerLaserdisk or vtrXObj XObjects. Finally, the video images are displayed on the Macintosh screen using the ColorSpaceFX board controlled by the XCMDGlue XObject (using the ColorSpaceFX XCMD).



Chapter 8: Using XCMDs and XFCNs

Lingo lets you use HyperCard XCMDs and XFCNs in your movies. You can use any existing XCMD or XFCN in scripts, as long as the XCMD's or XFCN's primary purpose is not to perform a HyperCard-specific action: for example, having to do with cards, HyperTalk scripts, or other parts of the HyperCard interface. You can thus extend the capabilities of Lingo by making use of the huge number of available XCMDs and XFCNs.

XCMDGlue is the Lingo word that lets you add XCMDs and XFCNs to your scripts. XCMDGlue works differently from XObjects. Many XCMDs and XFCNs work automatically with XCMDGlue, while others may not. For the latter case, a special mechanism is provided which may solve the problem for some XCMDs that don't seem to run the first time. This chapter describes how to use XCMDGlue.

Opening XCMD and XFCN Resources

There are two places XCMDs and XFCNs can be located. If an XCMD or XFCN resource is stored in the resource fork of the current MacroMind Director movie, it is automatically opened when the movie is opened (just like the **Standard.xlib* is automatically opened when you launch Director).

If an XCMD or XFCN resource is stored in an external file (such as a resource file or stack), you need to open it with the *openXlib* command:

```
openXlib XCMDresourcefile
```

All XCMDs and XFCNs stored in *XCMDresourcefile* are opened. *XCMDresourcefile* can be a HyperCard stack. Notice that this is the same command which is used to open regular XObjects.

Viewing XCMDs and XFCNs

The *showXlib* command displays all open resource files that contain XCMDs and XFCNs (as well as XObjects):

```
showXlib
```

To display the contents of a specific XCMD or XFCN resource file, use the *showXlib* command with the name of the resource file:

```
showXlib XCMDresourcefile
```

Closing XCMD and XFCN Resources

Use the `closeXlib` command to close all open resource files that contain XCMDs and XFCNs (as well as XObjects):

```
closeXlib
```

To close a specific resource file that contains XCMDs and XFCNs, specify that resource file's name with `closeXlib`:

```
closeXlib XCMDresourcefile
```

Using an XCMD or XFCN

In many cases, once you open an XCMD or XFCN (see “Opening XCMD and XFCN Resources”, above) all you need to do is use the XCMD or XFCN in your Lingo scripts just as you would use it in a HyperTalk script. That's all there is to it. `XCMDGlue` does all the work of converting the XCMD or XFCN for you. For example, here is how you would use the `Flash XCMD` that comes with HyperCard:

```
openXlib "Flash stack"
```

Flash Stack is a file containing the `Flash XCMD`. Once the file is open, you can use the XCMD:

```
flash
```

This is the HyperTalk word used to activate the command and flash (invert) the screen.

Occasionally, however, `XCMDGlue` is unable to properly convert the XCMD or XFCN. When you attempt to use the syntax of an XCMD or XFCN in a script, an error message is displayed.

Not all XCMDs and XFCNs can be used with `XCMDGlue` in a completely transparent manner. Certain XCMDs and XFCNs may call on HyperCard to internally perform some tasks while it is executing. These calls are known as **callback** requests. HyperCard provides twenty-nine callbacks. Most of these are conversion routines and are used to

conveniently convert information to and from different formats. The remaining callbacks either deal with the HyperTalk interpreter or access information stored in HyperCard-specific entities (such as fields), or both. See Table 8.1 later in this chapter for specific technical information regarding these callbacks.

Lingo automatically supports all callbacks that are not overly HyperCard-specific. Even some HyperCard-specific callbacks are supported where Lingo provides a direct equivalent. The remaining callbacks that are not automatically supported (a total of nine) are so HyperCard-specific that they cannot be resolved automatically unless the application calling the XCMD is virtually a HyperCard clone. Even in such cases, it is still possible to use an XCMD or XFCN by means of a user-defined mechanism called a **callback handler**. A callback handler is an instance of a factory object which accepts and responds to messages that correspond to HyperCard callback requests.

Essentially, a callback handler provides a mechanism that some XCMDs and XFCNs already expect to be available. The XCMD or XFCN expects that when it sends or receives a callback message, something will be there to receive it and possibly return another message (usually HyperCard does this). A callback handler defined in Lingo simply intercepts and returns these messages when appropriate. Whether you choose to use this information depends on your understanding of the purpose of the callback.

Using a Callback Handler

Fortunately, when `XCMDGlue` does not understand a callback request, it indicates the name of the callback in the error message.

Once you know which callback your XCMD or XFCN needs to deal with, you can create a callback handler for it. There are three basic steps to creating a callback handler:

- ◆ Defining a callback factory.
- ◆ Creating the callback object.
- ◆ Specifying the XCMD or XFCN to be used with the callback object (with the `setCallBack` command that is part of `XCMDGlue`).

The Callback Factory

The first step is to define the callback factory. Below is an example factory that includes methods for all the callbacks that are not supported by XCMDGlue. This factory does not attempt to do anything with the callback requests other than provide a record of them in the Message window for your own information:

```
--  
  
factory callBackFactory  
  
method mNew  
me(mPut, 1, "SendCardMessage")  
me(mPut, 2, "EvalExpr")  
me(mPut, 3, "StringLength")  
me(mPut, 4, "StringMatch")  
me(mPut, 5, "SendHCMessage")  
me(mPut, 6, "ZeroBytes")  
me(mPut, 7, "PasToZero")  
me(mPut, 8, "ZeroToPas")  
me(mPut, 9, "StrToLong")  
me(mPut, 10, "StrToNum")  
me(mPut, 11, "StrToBool")  
me(mPut, 12, "StrToExt")  
me(mPut, 13, "LongToStr")  
me(mPut, 14, "NumToStr")  
me(mPut, 15, "NumToHex")  
me(mPut, 16, "BoolToStr")  
me(mPut, 17, "ExtToStr")  
me(mPut, 18, "GetGlobal")  
me(mPut, 19, "SetGlobal")  
me(mPut, 20, "GetFieldByName")  
me(mPut, 21, "GetFieldByNum")  
me(mPut, 22, "GetFieldByID")  
me(mPut, 23, "SetFieldByName")  
me(mPut, 24, "SetFieldByNum")  
me(mPut, 25, "SetFieldByID")  
me(mPut, 26, "StringEqual")  
me(mPut, 27, "ReturnToPas")  
me(mPut, 28, "ScanToReturn")  
me(mPut, 31, "FormatScript")  
me(mPut, 32, "ZeroTermHandle")  
me(mPut, 33, "PrintTEHandle")  
me(mPut, 34, "SendHCEvent")  
me(mPut, 35, "HCWordBreakProc")  
me(mPut, 36, "BeginXSound")
```



```

me(mPut, 37, "EndXSound")
me(mPut, 38, "RunHandler")
me(mPut, 39, "ScanToZero")
me(mPut, 40, "GetXResInfo")
me(mPut, 41, "GetFilePath")
me(mPut, 42, "FrontDocWindow")
me(mPut, 43, "PointToStr")
me(mPut, 44, "RectToStr")
me(mPut, 45, "StrToPoint")
me(mPut, 46, "StrToRect")
me(mPut, 47, "GetFieldTE")
me(mPut, 48, "SetFieldTE")
me(mPut, 49, "GetObjectName")
me(mPut, 50, "GetObjectScript")
me(mPut, 51, "SetObjectScript")
me(mPut, 52, "StackNameToNum")
me(mPut, 53, "Notify")
me(mPut, 54, "ShowHCAalert")
me(mPut, 100, "NewXWindow/GetNewXWindow")
me(mPut, 101, "CloseXWindow")
me(mPut, 102, "SetXWIdleTime")
me(mPut, 103, "XWHasInterruptCode")
me(mPut, 104, "RegisterXWMenu")
me(mPut, 105, "BeginXWEdit/EndXWEdit")
me(mPut, 106, "SaveXWScript")
me(mPut, 107, "GetCheckPoints")
me(mPut, 108, "SetCheckPoints")
me(mPut, 109, "XWallowReEntrancy")
me(mPut, 110, "SendWindowMessage")
me(mPut, 111, "HideHCPalettes")
me(mPut, 112, "ShowHCPalettes")
me(mPut, 113, "XWAlwaysMoveHigh")
me(mPut, 200, "GoScript")
me(mPut, 201, "StepScript")
me(mPut, 202, "AbortScript")
me(mPut, 203, "CountHandlers")
me(mPut, 204, "GetHandlerInfo")
me(mPut, 205, "GetVarInfo")
me(mPut, 206, "SetVarValue")
me(mPut, 207, "GetStackCrawl")
me(mPut, 208, "TraceScript")

method mEvalExpr x
  put "mEvalExpr:" && x

method mSendHCMMessage x
  put "mSendHCMMessage:" && x

```



```

method mSendCardMessage x
  put "mSendCardMessage:" && x

method mGetFieldByName card, name
  put "mGetFieldByName:" && card && name

method mGetFieldByNum card, num
  put "mGetFieldByNum:" && card && num

method mGetFieldByID card, id
  put "mGetFieldByID:" && card && id

method mSetFieldByName card, name, value
  put "mSetFieldByName:" && card && name && value

method mSetFieldByNum card, num, value
  put "mSetFieldByNum:" && card && num && value

method mSetFieldByID card, id, value
  put "mSetFieldByID:" && card && id && value

method mUnknown which
  put me(mGet, value(which)) into callbackName
  put "mUnknown:" && which && "(" & callbackName & ")"

```

Note that you do not need to specify every callback handled in this factory. You are only required to define methods for the callbacks that are indicated in the error dialogs. For example, the `mEvalExpr` callback may often be the only callback you'll need to account for.

As indicated above, the `put` scripts in each method are optional. They exist to let you know what the XCMD or XFCN is attempting to tell HyperCard. You can use this information in any way you deem necessary. Sometimes, a callback requires a value (message) to be sent back to HyperCard. If you know what that value should be, use the `return` keyword at the end of the specific callback method's script. For example, if a callback required HyperCard to return `TRUE` or `FALSE` you could use a method similar to the following:

```

method callbackMethod
  if test then return TRUE
  else return FALSE
end callbackMethod

```

Some XCMDs and XFCNs are processor intensive (such as the `interFACE™ RAVE™` driver). In this situation, using a `put` script

slows down whatever the XCMD or XFCN does (because `put` has to be evaluated and written into the Message window). To optimize the callback factory in a case like this, just remove the `put` scripts. You don't have to enter the text of callback factories into your own documents. The above factory (both with and without `put` scripts) can be found in the file *Callback Factory*, in the *Factories* folder, which is located in the *Device Control* folder.

➡ **Note** When a callback error occurs, after you click OK in the error dialog box, the XCMD or XFCN usually stops running. However, because of the design of certain XCMDs and XFCNs, sometimes the XCMD or XFCN continues to execute. You still need to create a callback handler for these XCMDs and XFCNs. Otherwise, unexpected results could occur.

Creating the Callback Object

Once you have defined a callback factory, you can create the callback object with this script:

```
put callbackFactoryName(mNew) into callbackObjectName
```

Specifying the Callback Handler

Finally you specify the callback handler with the `setCallBack` command (this command is part of the *XCMDGlue* XObject):

```
setCallBack XCMD/XFCNname, callbackObjectName
```

The XCMD or XFCN should now function properly. Note that if you later use other elements of the XCMD's or XFCN's syntax, you may still need to deal with other callbacks. This is easily accomplished by adding the appropriate method to your callback factory.

For an example of the *XCMDGlue* XObject, see *XCMDGlue* in "The Apartment" sample movie.

XCMD and XFCN Callback Requests

The following table lists HyperCard's callback requests. The symbol in the rightmost column identifies which level of support is provided for each callback.

Table 8.1 HyperCard's callback requests.

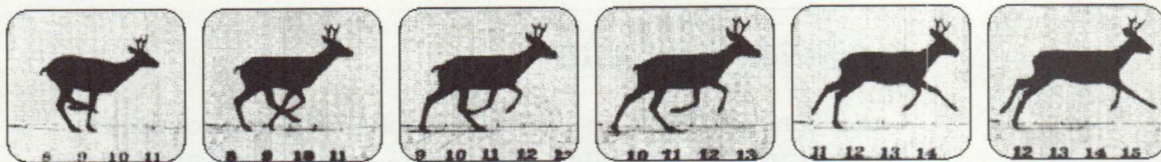
Number	HyperCard callback	Type
1	SendCardMessage	✎
2	EvalExpr	✎
3	StringLength	✓
4	StringMatch	✓
5	SendHCMMessage	✎
6	ZeroBytes	✓
7	PasToZero	✓
8	ZeroToPas	✓
9	StrToLong	✓
10	StrToNum	✓
11	StrToBool	✓
12	StrToExt	✓
13	LongToStr	✓
14	NumToStr	✓
15	NumToHex	✓
16	BoolToStr	✓
17	ExtToStr	✓
18	GetGlobal	✓
19	SetGlobal	✓
20	GetFieldByName	✎
21	GetFieldByNum	✎
22	GetFieldByID	✎
23	SetFieldByName	✎
24	SetFieldByNum	✎
25	SetFieldByID	✎
26	StringEqual	✓
27	ReturnToPas	✓
28	ScanToReturn	✓
31	FormatScript	✎
32	ZeroTermHandle	✎
33	PrintTEHandle	✎
34	SendHCEvent	✎
35	HCWordBreakProc	✎
36	BeginXSound	✎
37	EndXSound	✎
38	RunHandler	✎
39	ScanToZero	✓
40	GetXResInfo	✎
41	GetFilePath	✎
42	FrontDocWindow	✎
43	PointToStr	✎
44	RectToStr	✎
45	StrToPoint	✎

Table 8.1 (continued) HyperCard's callback requests.

Number	HyperCard callback	Type
46	StrToRect	✎
47	GetFieldTE	✎
48	SetFieldTE	✎
49	GetObjectName	✎
50	GetObjectScript	✎
51	SetObjectScript	✎
52	StackNameToNum	✎
53	Notify	✎
54	ShowHCAAlert	✎
100	NewXWindow/GetNewXWindow	✎
101	CloseXWindow	✎
102	SetXWIdleTime	✎
103	XWHasInterruptCode	✎
104	RegisterXWMenu	✎
105	BeginXWEdit/EndXWEdit	✎
106	SaveXWScript	✎
107	GetCheckPoints	✎
108	SetCheckPoints	✎
109	XWAllowReEntrancy	✎
110	SendWindowMessage	✎
111	HideHCPalettes	✎
112	ShowHCPalettes	✎
113	XWAlwaysMoveHigh	✎
200	GoScript	✎
201	StepScript	✎
202	AbortScript	✎
203	CountHandlers	✎
204	GetHandlerInfo	✎
205	GetVarInfo	✎
206	SetVarValue	✎
207	GetStackCrawl	✎
208	TraceScript	✎

✓: Automatically supported by Lingo.

✎: Requires a callback handler. Some messages and expressions (such as `EvalExpr`) may be evaluated by `XCMDGlue` in a manner compatible with HyperTalk. Other messages and expressions (such as `GetFieldByName`) always assume HyperCard entities for which there are no counterparts in MacroMind Director.



Chapter 9: *Advanced Topics*

This chapter provides supplemental information you can use for planning your movies. The sections on “Managing Color”, “Planning for Video Output”, and “The Virtual Cast Facility” will be useful to you in those special situations when you need to know more about color, video, and how cast information is managed. The section on the “Working With Coordinates” gives summary information and overviews that can help you understand how to use Lingo commands to control visual objects on the Stage.

Managing Color

You can use Lingo to manage and control the Macintosh computer's color capabilities, including 8-bit palettes and 24/32-bit environments. The following sections give you an overview of things to keep in mind about color management when you are designing your movies for a color environment.

Color Depth and Pixels

If you've used a color Macintosh, you've probably also used the Color Picker to choose and set colors for various applications. The Color Picker is a wheel and an associated dialog box, and is one way to control color on the Macintosh. The Color Picker employs three numbers that specify the Red, Green, and Blue components of the signal, and a slider bar that lets you set the brightness of the color you've picked.

With a color depth of 1-bit, each pixel on the screen can show one of two values: on or off (black or white). With a color depth of 16 bits, each pixel on the screen can display one of 32,768 values. With a color depth of 24 bits or (in the future) 48 bits, each pixel can be one of millions or even trillions of colors.

32-bit QuickDraw

The 32-bit QuickDraw Macintosh standard is an implementation of graphic routines that displays color information on the Macintosh. You can consider it state of the art on the Macintosh computer, circa 1990. This standard is supported by Director.

When you are creating color animations using Director, whether you will be using these in combination with live video or not, you are limited to a pixel depth of 32: 24 bits of color, plus 8 bits of so-called "alpha channel" information (described later in this chapter).

To access and use as many as 32 bits per pixel, you need to have a recent monitor card or a video card that supports the Apple Macintosh 32-bit QuickDraw standard.

32-bit Display Mode

The 32-bit display mode uses 24-bit direct color, with 8 bits each for the red, blue, and green components of the video. (8 additional bits are reserved for other uses, such as alpha channel information.) In comparison to 8-bit display mode, the drawback to having this greater range of colors available is that redraw and processing time is slower. This can have implications for your animations and movies.

16-bit Display Mode

The 16-bit display mode is an alternative to 32-bit display. This offers a compromise between the slow redraw time of 32-bit color versus the smaller range of colors available in 8-bit mode. The 16-bit scheme gives you 15 bits of color information (5 per color component) and 1 bit of alpha overlay information.

Direct Color

Both the display modes described above are called *direct color* schemes because each pixel within an image defines the color value directly, assigning a value to each of the colors (red, blue, and green). The alternative to direct color is indexed color, and is the method used in 8-bit, 4-bit and 2-bit display modes.

Indexed Color

On the Macintosh, 8-bit images do not contain information about the red, green, and blue components of the signal the way 16- and 24-bit color schemes do. Instead, indexed color uses a set of 256 colors that are defined as a subset of 16.777 million possibilities. This subset is called a *palette*. These are the colors or shades of gray that you can see on screen at any one time in the 8-bit display mode.

Each 8-bit image contains, as a part of the information stored with it, a pointer to the particular palette that was used *at the time the image was created*. This palette is called the *color lookup table* and is abbreviated CLUT. The image is said to be mapped to its CLUT.

Thus, for each pixel, Director assigns an index number between 1 and 256. The number corresponds to the position of the color in the related position on the CLUT. The image pixel does not contain any information about the RGB components themselves. These are derived from the palette.

The System Palette

Assuming that you have the appropriate monitor display card, you can choose display modes of 2-bit, 4-bit, and 8-bit color using the Monitors CDEV in your Macintosh control panel. For these modes, the operating system uses a pre-defined color lookup table called the System Palette. This is a representative set of hues and shades distributed across the spectrum. Unless you use applications that allow you to create your own custom palettes, you'll always be working with the colors as they are indexed in the System Palette.

Custom Palettes

Color scanners, sophisticated paint programs, and video capture boards all create and allow you to manipulate custom palettes. If you were forced, with these applications, to always use the system palette, the effects could be somewhat garish. Custom palettes and their manipulation lets you get the most pleasing visual effects possible from a limited range of colors.

For example, if you intend your movies to be recorded in NTSC video, then you'll want to use the NTSC palette supplied with Director.

Color and the Alpha Channel

The concept of an alpha channel for digital video has its roots in the film industry. It is an extension of the idea of travelling mattes to put multiple film images on the same frame. The alpha component for each pixel in the image is an 8-bit value that specifies the relative degree of transparency for the accompanying 24-bit true-color image. With a single alpha bit a value of 0 indicates complete transparency, while a value of 1 indicates complete opacity. An 8-bit value, then, allows 256-step

transitions, from opaque, through various degrees of translucency, to complete transparency. A value of 128 represents a 50-50 mix between the background and foreground image.

The preceding describes how the alpha channel works to mask edges or blend images in 32-bit paint programs or 3D rendering programs. Video overlay cards, however use the alpha channel in a different way: the video is displayed using the main 24-bits, spread across the R, G, and B channels. The alpha channel of the video card contains whatever graphic image would normally be displayed on the computer monitor, depending on the application program you are running, up to a limit of 8-bits.

The graphic image in the alpha channel replaces the video on a pixel-for-pixel basis wherever a particular chromakey color is found in the main RGB channels. The RGB value of the chromakey color is specified manually using a Control Panel device or a desk accessory, or in Lingo using an XObject to control the overlay card.

As a result of this limitation of 32-bit QuickDraw, whenever you overlay Director animations over a full color 24-bit video, you will be limited to a movie with 8-bit castmembers or less.

This basic idea of extending the amount of data stored with an image pixel to include alpha channel information is now the backbone of modern video overlay techniques involving computer-generated images. The most commonly-used pixel specification is 32 bits: 24 bits for the image (8 bits each for R, G, and B channels), and 8 bits for the alpha channel.

Furthermore, alpha channel information can be used further down in the video production stream, by other digital video image processors, to provide advanced image compositing capabilities. Alpha information can be used by video switchers or downstream keyers. Many of the video packages and add-on boards for the Macintosh computer support alpha channel information, as do some paint programs (TrueVision, NuVista, Adobe Photoshop, Oasis).

► **Note** The word “channel” as used here in the phrase “alpha channel” should not be confused with the channels in the Score window. The meanings are distinctly different.

Most video overlay cards put an 8-bit signal from the Macintosh into the alpha channel over a 24-bit video image. The result is that you can't overlay a 24-bit movie on top of 24-bit video.

The Palette Channel

In Director's Score window, the Palette channel is provided so that you can use a particular palette to match 8-bit castmembers that you may have created and/or imported from applications that let you use custom palettes, starting at a particular frame.

As long as your system is in 8-bit display mode, you can import a custom palette at the same time as you import your castmember image. When you do so, both the palette and the image will become individual castmembers in your movie.

For images that you create within Director, you can use the palette window to select or create a custom palette for a castmember.

A new palette becomes the *active palette* when the playback head reaches the frame containing the custom palette, or when you declare it to be a puppet palette from within a Lingo script. When that happens, you'll see the colors in the active palette on the display change to reflect the new mapped values.

It is important to note that the substitution of color values on the display happens only on the screen. No change is made to the basic image information itself. In Director, the only way to permanently change the colors in a bitmapped castmember, for example, is to use the Remap Cast command from the Cast menu, or the pop-up menu in the Cast Info dialog box. The selected castmember will then be remapped to reflect the currently-active palette.

Planning for Video Output

If you are using Director and Lingo to create movies that you want to transfer for later playback from a videodisc, videotape, or other video storage device, you need to keep in mind all the factors that will

contribute to the best possible results, before you send your movie out to tape. This section provides you with the minimum tips to remember.

The Recorder Format

The better your video recorder, the better your results will be. Of course, better also correlates with more expensive. Here are the formats in increasing order of quality.

Composite Analog

VHS cassettes

Sony Betamax cassettes (1/2" and 3/4" formats)

Sony U-Matic cassettes (3/4" format)

Type B (helical scan, 1" reels)

Type C (helical scan, 1" reels)

8mm cassettes

Component Analog

Component video is so named because it separates the video signal into two components: the luminance or brightness component, and the color or chrominance component. The "S" in S-VHS stands for "separation".

S-VHS cassettes

Hi-8 (8mm) cassettes

U-Matic (Sony Betacam SP) cassettes

MII ("M-two") cassettes

Composite and Component Digital

Digital recorders convert video information to digits that then have all the beneficial qualities of other kinds of digital signal. They also can be manipulated by computer methods. This is currently the highest quality video recording system available, and can be found in many commercial environments, such as television stations and movie production studios.

Video Cards

Again, the higher quality Macintosh video display cards will give you better results with Director movie output to video.

Cards that support S (separated) video are better. (See the preceding section for description of S video.)

A scan converter is useful and probably essential if you are going to send Macintosh computer output to a commercial digital tape recorder. Digital recorders require a color difference signal. A scan converter will change the RGB component video display of your Macintosh computer to the difference signals needed by the higher quality digital recorders.

The Video Transfer Method

There are two ways to send a signal to a recorder: frame-by-frame or in real time. In general, frame-by-frame is more accurate, gives you better results, and can be controlled more closely, an important consideration when you are designing movies where split-second timing is required.

For frame-by-frame output, the video recorder must be capable of frame-accurate control. Furthermore, the software that you use to control the recording device must also support this feature. In Director, this means that you must use the appropriate XObject in conjunction with the recorder.

Even when you use a frame-by-frame transfer method, however, you'll probably want to record transition effect in real time to get the best results. If you don't, the recorded transition may take a longer time to playback than you expected.

Pixel Point Sizes

If you intend your movies to be seen on an ordinary even high-quality video monitor, you should make sure that all of the lines composing your images is at least 2 pixels wide. See Appendix C of the *MacroMind Director Studio Manual* for more about this recommendation.

Font Sizes

Larger font sizes are better for video. You'll notice this if you watch your local commercials and tv graphics more closely. The recommended size is 24 points or greater.

Color Saturation

Use the NTSC palette to create or re-map your images in Director when you intend movies to be output to video. (Some scan converters may perform the appropriate conversion for you.) The intense colors available on the Macintosh computer screen may result in bleeding and other undesirable artifacts when you try to transfer them to video.

Frame Cropping

Your Macintosh display screen contains a limited number of pixels (width times height of the image). However, the edges of the image will probably be cropped off when you transfer it to a video recording device. The way in which this is done will depend on the output card you're using.

Therefore, when you are designing your presentation, animation, or other movie, keep in mind the effect that such cropping will have on your images. If you use a video monitor to preview your work, you'll know for sure and won't be surprised when it comes time to put down your final take.

Monitor Used in Development

You should always test your movies on a monitor similar to the one that will be used to playback your tapes. The recommended practice is to use a good monitor connected to the VCR or other recording device as you develop your movies. That way, you'll be able to make the important technical and visual design decisions right away, and will reduce the number of potentially nasty surprises later, when it's too late to change them.

Virtual Cast Facility

MacroMind Director 3.0 provides a new way of managing memory and cast resources in the form of a *virtual cast* facility. Director automatically loads cast members as they are needed, and removes them from memory when they're not needed in order to make room for new castmembers. If you are working with a small movie, you probably won't notice a difference between version 2.0 and version 3.0, but if you are composing a movie with a lot of color castmembers, you'll notice that Director will occasionally read from the disk drive while running the movie.

In general, you can allow Director to decide when to load castmembers into memory, and when to clear them from memory. Lingo, however, provides three commands that let you manage the cast yourself. These commands are described in this section.

If you are running Director under Macintosh System 7, you will find that the Macintosh computer's built-in virtual memory management does not interfere with Director's virtual cast facility. On the other hand, you can expect reduced performance speed, because two different algorithms are being used to move data between the computer's memory and the disk drive. For improved performance, you should switch off System 7's virtual memory while running Director. For more information on managing castmembers, refer to your *MacroMind Director Studio Manual*.

Managing the Virtual Cast

Lingo provides limited direct control of the virtual cast facility, using the Lingo commands `preLoad` and `preLoadCast`.

The preLoad Command

The `preLoad` command lets you select a range of frames, and load all of their castmembers in advance. As with the `lockCast` command, the `preLoad` command lets you look ahead during a movie.

For example, if you know you will have a pause followed by a rapid sequence of frames, you can use the `preLoad` command to load castmembers during the pause. This will keep Director from slowing down to load castmembers as they are needed during the rapid sequence.

The `preLoad` command can be used by itself, or it can be followed by one or two arguments. If you use `preLoad` alone, Director will load all of the castmembers needed from the current frame through the end of the movie. If you follow `preLoad` by a single argument, Director accepts this as the *toFrame* argument, and loads all of the castmembers used in the range of frames from the current frame through to the frame you specify as the argument. If you follow `preLoad` by two arguments, Director loads all of the castmembers used in the range of frames specified by the arguments.

If Director finds that there is not enough memory to load all of the castmembers you specify with the `preLoad` command, it stops the `preLoad` procedure and returns to normal way of using the virtual cast facility.

The preLoadCast Command

The `preLoadCast` command works like the `preLoad` command, but it lets you load specific castmembers, instead of the castmembers for a range of frames.

Like the `preLoad` command, `preLoadCast` can be used by itself, or followed by one or two arguments. If you use `preLoadCast` by itself, all of the castmembers in the movie will be preloaded. If you provide a single argument for `preLoadCast`, the castmember you specify will be loaded. If you provide two arguments, `preLoadCast` loads the range of castmembers specified by the arguments.

If Director finds that there is not enough memory to load all of the castmembers you specify, it stops the `preLoadCast` procedure and returns to normal virtual cast management.

Working With Coordinates

Director uses two coordinate systems. You use the display coordinate system to specify the position of the Stage on monitor screens. You use the Stage coordinate system to specify the location of sprites and the mouse pointer relative to the Stage. If the Stage is the same size as a single monitor screen, and it completely fills the screen, the two coordinate systems resolve into one.

The Stage Coordinates

The *MacroMind Director Studio Manual* describes how you set the size and position of the Stage using the Preferences dialog box. (Choose Preferences from the Control menu.) You cannot set the size and position of the Stage using Lingo, but Lingo does provide four functions that let you retrieve that information during a movie:

```
the stageLeft  
the stageRight  
the stageTop  
the stageBottom.
```

The width of the Stage is equal to

```
the stageRight - the stageLeft
```

and the height of the Stage is equal to

```
the stageBottom - the stageTop
```

The position of the Stage is defined with respect to the display coordinate system. The upper left corner of the monitor is defined as the origin (0, 0) of the display coordinate system. The horizontal coordinate value increases as you move to the right, and the vertical coordinate value increases as you move down. If you have a single screen, the position of the Stage, and the values returned by the `stageLeft`, `stageRight`, `stageTop`, and `stageBottom` functions are always positive.

If you are using more than one monitor screen, the primary monitor (the monitor with the menu bar) is the one that establishes the origin of the display coordinate system. Thus, if the Stage is located on a monitor to the left of the primary monitor, the `stageLeft` and the `stageRight` functions will return negative values. Similarly, if the Stage is located on a monitor above the primary monitor, the `stageTop` and the `stageBottom` functions will return negative values.

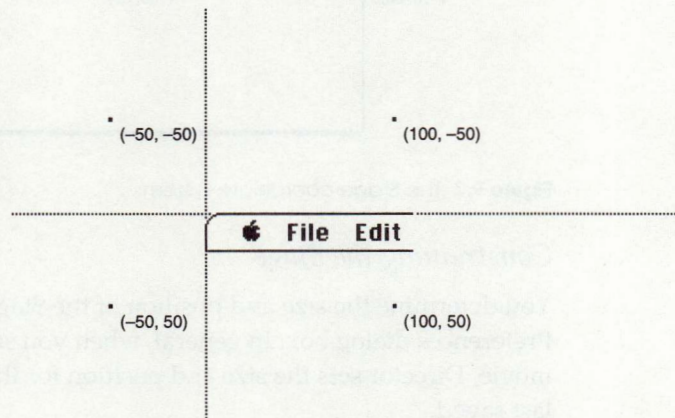


Figure 9.1 The screen coordinate system

Mouse and Sprite Coordinates

The position of sprites and the mouse pointer are defined with respect to the Stage. The upper left corner of the Stage is defined as the origin $(0, 0)$ of the coordinate system. As with the screen coordinate system, the horizontal coordinate increases as you move to the right, and the vertical coordinate increases as you move down. Within the Stage boundaries, all of the coordinate numbers are positive. Thus, if a sprite is visible on the Stage, its coordinates will have a positive value.

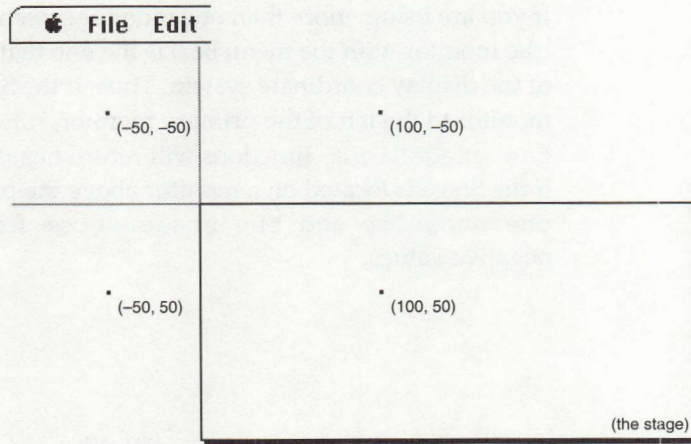


Figure 9.2 The Stage coordinate system

Constraining the Stage

You determine the size and position of the Stage on the screen with the Preferences dialog box. In general, when you start playing a different movie, Director sets the size and position for that movie the way it was last saved.

Lingo provides you with two commands to override this default operation. If you set the `centerStage` property to `TRUE`, Director will automatically center the Stage on the primary screen, without regard to the position that was last saved with the movie.

Similarly, if you set the `fixStageSize` property to `TRUE`, Director will maintain the size of the current Stage when it loads a new movie, without regard to the size setting that was last saved with the movie.

The Mouse Pointer Position

The position of the mouse pointer is determined by the user, so it cannot be set directly from Lingo. Lingo provides two functions to give you the current position of the mouse pointer:

```
the mouseH
```

```
the mouseV
```


Lingo will return values for these functions based on the Stage coordinate system. These functions track the position of the mouse pointer continuously, even when it is outside of the Stage, returning negative values when the mouse pointer is above or to the left of the Stage.

Sprite Positions

Lingo provides two sets of sprite properties to indicate the current position of a sprite on the Stage. The first set uses the rectangular borders of the sprite, and the second set uses the sprite's registration point (see the *MacroMind Director Studio Manual* for information about registration points). For QuickDraw sprites, the default registration point is at the upper left corner of the sprite, but for bitmapped sprites it is at the center of the sprite.

Sprite Border Properties

Four sprite properties indicate the rectangular borders of a sprite in the Stage coordinate system:

```
the left
the top
the right
the bottom
```

As with the mouse pointer functions, these properties can be read, but they cannot be directly set.

If the sprite is a puppet, Lingo provides the `spriteBox` command you can use to indirectly set these properties. This example moves the sprite down and to the right:

```
set lCoord = the left of sprite 1 + 10
set tCoord = the top of sprite 1 + 10
set rCoord = the right of sprite 1 + 10
set bCoord = the bottom of sprite 1 + 10

if the puppet of sprite 1 then ↵
    spriteBox 1, lCoord, tCoord, rCoord, bCoord
```


The Registration Point Properties

Two additional sprite properties determine the horizontal and vertical position of the sprite on the Stage:

```
the locH
```

```
the locV
```

Like the properties described above (left, top, right, and bottom) the `locH` and `locV` are based on the Stage coordinate system. Nevertheless, these properties are *not* equivalent to `left` and `top` of the sprite. The `left`, `right`, `top`, and `bottom` sprite properties indicate the respective edges of the sprite itself. The `locH` and `locV` sprite properties determine the registration point of the sprite on the Stage.

Constraining the Position of a Sprite

If you have given a sprite the `moveableSprite` command, the sprite can be moved around the monitor screen, even off the Stage. The `left`, `right`, `top`, `bottom`, `locH`, and `locV` properties will return the values of the current sprite position, even if it is a negative number, or greater than the dimensions of the Stage.

Lingo provides a sprite property to limit the range of movement of a sprite:

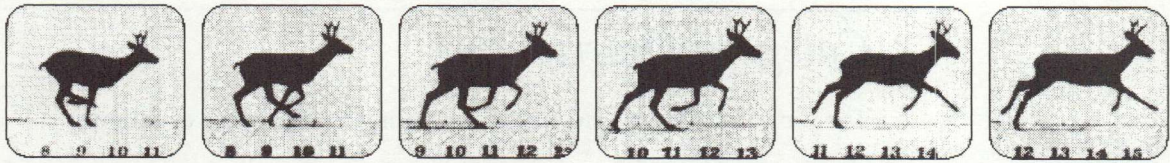
```
the constraint
```

The `constraint` lets you confine the movement of a sprite within the borders of a second sprite:

```
set the constraint of sprite 2 to 4
```

This example uses sprite 4 to establish the boundaries of movement of sprite 2.

The `constraint` property uses the `left`, `right`, `top`, and `bottom` of sprite 4 as the boundaries of movement. The position of sprite 2, however, is determined by its `locH`, and `locV` properties. Therefore, the registration point of sprite 2 is limited to the edges of sprite 4. In this example, if sprite 2 is a bitmap, it can be allowed to overlap the edge of sprite 4.



Chapter 10: Lingo Summary

This chapter provides a summary of the major Lingo word groups and guidelines for script placement. Each of the words is described individually in Chapter 11. There is a quick summary of all the words, identified by category, in Appendix A.

Category Descriptions

The Lingo words are categorized according to their type, as follows:

- ◆ Commands
- ◆ Functions
- ◆ Keywords
- ◆ Properties
- ◆ Constants
- ◆ Operators
- ◆ Predefined Methods and Special Messages common to Lingo Objects

The following sections describe each category. In Chapter 11, each entry shows the vocabulary element in the title.

Commands

Commands instruct MacroMind Director to perform some kind of action. Actions include such things as moving the playback head (*go*, *play*, *play done*), *stop* or *resume* the interactive movie (*pause*, *continue*), *create a menu* (*installMenu*), and *making a sprite moveable or editable* (*moveableSprite*, *editableText*).

Command names cannot be used as variables.

Functions

Functions return a value. The value returned is based on the arguments passed in, or on the current state of the system. Functions do not change the system in any way.

Function names cannot be used as variables.

Keywords

Keywords are Lingo words that have specific meanings. They are used within text castmembers when defining multiple-line macros or factories.

You cannot use keywords as variable names.

Properties

Properties are named characteristics or attributes of the current system, of specific sprites, and of text castmembers. The characteristics of system properties include the location of the Stage, whether a `mouseDown` event will activate a script, and whether 32-bit color is active.

Properties are super-global in their scope, which means they are available within scripts, handlers, and methods without an explicit global declaration. Like global variables, properties are available between different movies in the same presentation (unless changed by system events.)

Sprite properties are named characteristics of sprites. A sprite is an instance of a castmember or a QuickDraw element (primitive or text) on the Stage, and consequently in the Score. Text properties are named characteristics of text castmembers. Sprite and text properties change when a new movie is loaded.

A property can be generally set and changed with the language. In addition, the current property setting can usually be tested by the language. Some properties can only be set, and others can only be tested. Whether a property can be set or only tested is specified in each case in Chapter 11, "The Lingo Dictionary."

Constants

Constants are named values that do not change. You can use a constant in place of certain characters or numbers when necessary. You would use the `QUOTE` constant, for example, if you wanted to include quotation marks in a literal, as in:

```
set the text of cast "wanda" to ¬
    QUOTE & "What?" & QUOTE
-- enters "What?" in Cast named "wanda"
```

Constants do not require quotation marks. You cannot use a constant as a variable name.

Operators

Operators are used to calculate values and compose expressions. The operators that are Lingo words are listed in alphabetic order. The operators that are symbols (mathematical, logical) are listed at the end of Chapter 11, in their order of precedence.

Precedence determines the order of the calculations in an expression. Calculations that use an operator with a higher order number are performed before those with a lower order number. In expressions that use operators with the same precedence order, the operators are evaluated from left to right. Parentheses change the order of evaluation.

Arithmetic operators work on numbers and result in numbers. Comparison operators work on sprites, numbers, strings, and boolean values (TRUE, FALSE), and result in boolean values. Logical operators work on boolean values and result in boolean values. Text operators work on text strings (or numbers treated as text) and result in text strings.

Predefined Methods and Special Messages

Lingo has some built-in, predefined methods that can be used at any time. This means that they are available to every Lingo factory object and/or external XObjects. Since these predefined methods are so commonly used in scripting, they are included in Chapter 11.

Methods are defined as part of Factories and XObjects. Methods make possible the creation and use of object instances. This saves having to keep track of a lot of pointers and variables when you need multiple items with similar functionality.

XObjects are external XCOD resources which extend the functionality of Lingo to include access to system routines, or external devices. XObjects are created in the languages of C or Pascal, using the syntax and structure defined in the *XObject Developer's Kit* available from MacroMind Customer Service.

By convention, the placeholder for specifying Lingo methods is *messageName*. In practice, method names begin with a lowercase m.

Scripting Guidelines

Here is a summary of the script writing principles and techniques presented in this manual. The guidelines cover general recommendations about where to place your scripts for maximum effectiveness, and a set of stylistic recommendations you can use when writing your own scripts.

Movie Scripts

The Movie script is the place to define what happens when the movie is idling, started and stopped, and when the playback head moves to a new frame. It is also the place where you put all the handlers that you want to be available to Score or Cast scripts.

The following messages are automatically generated by Lingo and sent to the Movie script, as follows:

`startMovie`

This message is sent to the Movie script when the movie starts playing, after Director displays the first frame.

`stopMovie`

This message is sent to the Movie script if the movie stops for any reason.

`stepMovie`

This message is sent to the Movie script when a new frame is displayed.

`idle`

This message is sent to the Movie script when nothing else is going on.

Score Scripts

There are two places where you can attach scripts in the Score: in the Script channel and attached to a sprite. Use Script channel scripts to handle simple frame-specific flow control, like pausing and looping. Script channel scripts are also the place to set up event scripts for actions

that occur outside of a castmember (for example, when `keyDown` events), and to control properties (such as `set the text of cast "name" to "Endiva"`).

If you want the same script to play in every frame, use an `on stepMovie` handler in the movie script.

If you need the same castmember to behave slightly differently in different frames, use the Cast script to define a reasonable default behavior. Then use a sprite script in the Score to override the Cast scripts when necessary. For example, the left arrow button in the Apartment movies has a Cast script that reads:

```
go to marker (-1)
```

This line moves back to the previous section of the movie. For the help section, however, a sprite script overrides this action to make the left arrow go back to the first section of the movie.

Sprite scripts respond to the `mouseUp` message generated when the mouse pointer is positioned over the sprite on the stage. You can not define other handlers in a sprite script.

Cast Scripts

Use `on mouseUp` and `on mouseDown` handlers in cast scripts to define what happens when the castmember is clicked. These are the only events to which a Cast script may respond. Other handlers that may be called by the mouse handlers and related specifically to that castmember can also be placed in the castmember script. (For example, see the Shake button script in the “Simple Puppets” sample movie.)

If you use several castmembers to animate an object, try to limit the scripts of each to simple one-line handlers, such as:

```
on mouseUp
  doSomething
end mouseUp
```

Here, `doSomething` is a handler that you’ve defined in a movie script. This is not only faster, but you won’t have to revise several copies of the handler, one for each castmember in the series.

Handlers

You can define handlers only in Movie Scripts or Cast Scripts.

Factories

You can define factories only in Movie Scripts, Cast Scripts, or in the Text cast window. Factory definitions will not work in a Score Script.

Macros

Text castmember macro definitions were used in previous versions of Lingo. Macros have been displaced by handler definitions and Movie Scripts. However, this version of Lingo will play movies that still have macros in them. Macros can only be defined in the Text cast window.

Frame Markers and Labels

Always use frame markers and labels in your scripts, rather than referring to frames by number. This allows you to insert, delete, and revise frames without having to update your scripts.

Naming Castmembers

Assign names to those castmembers that you want to refer to in other scripts. This allows you to rearrange the castmembers in the Cast window without having to go back and revise all your scripts.

However: you still have to refer to a castmember by number when changing the castmember associated with a sprite, as in the following line.

```
set the castNum of sprite 5 to C11
```

When to Use stepMovie

Use an `on stepMovie` handler in the Movie Script when you want the same thing to happen at every frame. This is simpler than using `perFrameHook`.

Script Writing Style

Lingo is not case-sensitive. You can use any combination of uppercase and lowercase characters that you want. Generally, lowercase characters are used because you can enter them easily, and can then put uppercase letters to better use, therefore:

- ◆ Lingo commands, functions, keywords, variable names and method names are shown in lowercase, unless they are compound words. For readability, compound words begin with lowercase, and the second part (and subsequent parts, if any) begins with an uppercase letter. If you use the Lingo menu to insert Lingo text, you'll automatically be using this style, and will reduce typing errors, a common source of bugs in code.

```
beepOn  
tripleCompoundWord
```

- ◆ Lingo constants are shown in all capital letters:

```
TRUE  
EMPTY
```

- ◆ The Lingo editor automatically indents two character spaces and provides one level of indentation. The editor indents all of the lines following a handler definition, or a control structure (such as `if` or `repeat`) and outdents a line beginning with the word `end` (such as `end if` or `end repeat` or `end handlerName`).
- ◆ Precede your handler definition by a blank line. When the script is very long, or contains a series of control structures, or use two or more lines. If you need to print out your scripts, you'll thank yourself, and the extra lines don't cost you any disk or memory space.
- ◆ Precede control structures (`if` and `repeat`) with a blank line.

Commenting and Naming Conventions

Lingo has been made English-like for easy reading and understanding. You extend Lingo by writing handlers, factories, and XObjects. If you are careful about how you name and then comment your scripts, you'll find

it a lot easier to edit and debug them. A variable or a short, simple handler probably won't need any additional commenting if you've named it well. Here are some more tips.

- ◆ Avoid short and cryptic names for variables and handlers (for example, `v1` is not as self-descriptive as a name such as `usersAnswer`). Choose names that describe in some way what the variable or method does (for example, `leftEyeBlink`).
- ◆ Precede method names in factories with a lowercase "m":

`mNew`

`mDispose`
- ◆ Provide comments that describe what a specific handler immediately after the `on handlerName` definition line.
- ◆ A variable or a short, simple handler probably doesn't need any additional commenting if you've named it well.
- ◆ Like handlers, control structures (those initialized by `if`, `repeat`, for example) should have comments located immediately *after* the end of the structure. This automatically indents the comment, emphasizing the control structure itself.
- ◆ In-line comments can quickly obscure the structure of the script code if they are not used with great care. In general, try to avoid these in favor of dedicated comment lines.
- ◆ In-line comments *are* appropriate where the script code is made up of a series of short lines, such as a series of `else if` lines. When in-line comments are used, the comment block should be lined up (all comment lines should start in the same vertical column), and should be at least eight or twelve character spaces to the right of the longest script code line.
- ◆ Refer to Chapter 11 for information about the syntax conventions used in this manual.



Chapter 11: *The Lingo Dictionary*

This chapter presents an alphabetical listing of Lingo words, followed by a listing of non-alphabetical symbols. Every Lingo language element appears in this section.

Typographic Conventions

Here are the conventions used in this dictionary.

In the *Syntax* and *Example* sections, words or phrases in `typewriter type` are Lingo words or elements that you type literally, exactly as shown. Words or phrases in *italic type* are placeholders that describe something general that you type, not the actual thing. For example:

`open fileName`

In this statement, the typefaces show you that you type `open`, but that you provide an actual filename for the word *fileName*.

Square brackets ([]) enclose optional elements, which may be included if you need them. Don't type the square brackets. An optional element may or may not change what the statement does. Some commands have optional arguments that you use in particular circumstances. Often an optional word is available so that a statement is easier to read (`go to frame 23` is the same as `go 23`).

In the *Description* and *See Also* sections, language categories are shown in *italic type*.

Throughout this book, Lingo words, handler names and variable names are shown in small letters with a capital letter in the middle of compound words (for example, `mouseDown`) to make these words easier to read.

A

A11...H88

cast identifiers

Syntax A11

Description Cast identifiers A11 through H88 are reserved as numerical constants for referring to castmembers. As a result, you cannot use these cast identifiers as variables.

There are three ways to refer to castmembers:

- 1) by cast name (for example, "Red Balloon");
- 2) by cast identifier (for example, A21);
- 3) by cast number (for example, 9).

Using cast names (which you assign in the Cast Info dialog box) is best because the names don't change if you rearrange the Cast window.

Cast identifiers and cast numbers are equivalent ways of describing the actual position of a castmember in the Cast window. In fact, a cast identifier is simply a symbolic constant for the corresponding cast number. For example, since castmember A21 is the 9th one in the Cast window, the constant A21 has the value 9.

This means that you can perform integer arithmetic with cast identifiers. For example, the castmember which is 5 locations beyond A21 can be referred to as "cast (A21 + 5)".

Examples put A21
-- 9

- ◆ put A21 + 5
-- 14
- ◆ set the hilite of cast B13 to TRUE
- ◆ put C11 into firstField

See also the name of cast **and** the number of cast **cast properties**

abbreviated

See the `date` and `time` functions.

abs

function

Syntax `abs (numericExpression)`

Description This function returns the absolute value of a numerical expression. If *numericExpression* evaluates to an integer, its absolute value is also an integer. If *numericExpression* evaluates to a floating point number, its absolute value is also a floating point number.

The `abs` function is useful for tracking mouse and sprite movement. Use it to convert coordinate differences (which can be either positive or negative) into distances (which are always positive).

Examples `put abs(-67)`
 `-- 67`

◆ `put abs(2.2)`
 `-- 2.2`

◆ `if abs(the mouseV - startV) > 30 then ↵`
 `put TRUE into draggedEnough`

after

See `put...after` command.

alert

command

Syntax `alert message`

Description This command causes a system beep and displays an alert dialog box containing the evaluated string expression *message* and an OK button. This command is useful for error messages.

Examples alert "There is no CD-ROM drive connected."

- ◆ alert "The file" && QUOTE & filename & QUOTE ¬
&& "was not found."

and

logical operator

Syntax *logicalExpression1* and *logicalExpression2*

Description This operator performs a logical AND operation on two logical expressions. When both *logicalExpression1* and *logicalExpression2* evaluate to TRUE, the result is TRUE (1). When either or both expressions evaluate to FALSE, the result is FALSE (0).

This is a logical operator with a precedence level of 4.

Examples put 1 < 2 and 2 < 3
-- 1

- ◆ put 1 < 2 and 2 < 1
-- 0

- ◆ if field "City" = "San Francisco" ¬
and field "State" = "CA" then ¬
put "Beautiful city" into field "Description"

See also or and not logical operators

B

the backColor of sprite

sprite property

Syntax the backColor of sprite *whichSprite*

Description This sprite property determines the background color of the sprite specified by the integer expression *whichSprite*. The background color applies only to 1-bit bitmap and shape castmembers. It does not affect

the display of a text or button castmember. An 8-bit bitmap is affected, but generally not in a useful way. Setting the `backColor` in a Lingo script is equivalent to choosing the background color from the Tool window when the sprite is selected on the Stage.

The value of a sprite's `backColor` ranges from 0 to 255 for 8-bit color, or from 0 to 15 for 4-bit color. This number is actually the index number of the background color in the current palette. (You can click any color in the Palette window to see its index number in the window's lower left corner.)

The `backColor` sprite property can be tested and set, although in order to set it with Lingo the sprite must be a puppet.

Examples `put the backColor of sprite 5 into oldColor`

- ◆ `set the backColor of sprite (10 + random(3)) to 36`
-- color 36 in System palette is pale blue

Note If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

Tip One use of the `backColor` property which does work consistently with 8-bit bitmap sprites is to specify which color will be made transparent using the Score window ink effect *Background Transparent*. This is a particularly useful ability when creating or importing anti-aliased graphics or objects from a 3-D rendering program for use over video.

Using a black stage color which is defined as the overlay color by the video card, and having images which were anti-aliased against a black background usually works best. The result will be that a dark gray shadow will be seen behind the graphic when used over a video source. This is the least objectionable shadow color.

Assign the Score window *Background Transparent* ink effect to all sprites which need this treatment. While they are still selected, set the `backColor` to black using the Tool window Background Color chip.

See also `foreColor` sprite property; `stageColor` property

BACKSPACE

character constant

Syntax BACKSPACE

Description This character constant represents the backspace key, marked “delete” on the Macintosh keyboard.

Example when keyDown then \neg
if the key = BACKSPACE then clearField

beep

command

Syntax beep [*numberOfTimes*]

Description This command causes the Macintosh computer’s speaker to beep the number of times specified by the integer expression *numberOfTimes*. If *numberOfTimes* is missing, the beep occurs once.

Examples if field "Zip" = EMPTY then beep 2

- ◆ when keyDown then beep
- ◆ beep random(maxBeeps)

Note The “beep” sound is actually whatever Alert Sound is currently selected in the Sound control panel. If the Speaker Volume in the Sound control panel is set to 0, the menu bar flashes instead. Lingo’s `soundEnabled` and `soundLevel` properties have no effect on the alert sound.

See also beepOn property

the beepOn

property

Syntax the beepOn

Description This property determines whether the Macintosh speaker beeps when the user clicks outside an active sprite. If `TRUE`, clicking outside an active sprite results in a beep. An active sprite is a sprite that has a script associated with it.

The `beepOn` property can be tested and set, and the default value is `FALSE`.

Examples if the beepOn = TRUE then \neg
alert "Click one of the sprites."

- ◆ set the beepOn to TRUE
- ◆ set the beepOn to (not the beepOn)

Note The "beep" sound is actually whatever Alert Sound is currently selected in the Sound control panel. If the Speaker Volume in the Sound control panel is set to 0, the menu bar flashes instead. Lingo's `soundEnabled` and `soundLevel` properties have no effect on the alert sound.

See also `beep` command

before

See `put...before` command.

the bottom of sprite

sprite property

Syntax the bottom of sprite *whichSprite*

Description This sprite property indicates the bottom vertical coordinate of the bounding rectangle of the sprite specified by the integer expression *whichSprite*.

The `bottom` sprite property can be tested, but not set directly. The bottom vertical coordinate of a sprite can be set with the `spriteBox` command.

Examples if the bottom of sprite 3 > (the stageBottom \neg
- the stageTop) then `offBottomEdge`

- ◆ `put` the bottom of sprite (i + 1) into `lowest`

Note Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

See also `top`, `left`, `right`, `height`, `width`, `locH`, and `locV` sprite properties; `spriteBox` command

the `buttonStyle`

property

Syntax `the buttonStyle`

Description This property determines what happens when a user presses the mouse button while the pointer is over a button, but then moves the pointer off the first button and onto other buttons. When the user places the pointer over a button and presses the mouse button, the button inverts until the user moves the pointer off the button. The `buttonStyle` property controls the visual action of buttons that the pointer subsequently touches, while the mouse button is continually being pressed.

The values you can assign to the `buttonStyle` property are:

- 0 list style
- 1 dialog style

When the `buttonStyle` property is set to 0 (list style), subsequent buttons are highlighted. If the user releases the mouse button while the pointer is over such a button, the script associated with that button is activated.

When the `buttonStyle` property is set to 1 (dialog style) only the first button is highlighted. Subsequent buttons are not highlighted. If the user releases the mouse button while the pointer is over a button other than the original button clicked, the script associated with that button is not activated.

The `buttonStyle` property can be tested and set, and the default value is 0 (list style).

Examples `put the buttonStyle into previousButtonStyle`

◆ `set the buttonStyle to 1`

See also `checkBoxAccess` and `checkBoxType` properties

C

cast

keyword

Syntax the *property* of cast *whichCastmember*

Description This keyword specifies to Lingo that the next expression refers to a castmember.

If *whichCastmember* evaluates to a string, it is used as the cast name.
If *whichCastmember* evaluates to an integer, it is used as the cast number.
(The cast identifiers A11 through H88 evaluate to integer cast numbers.)

Examples

- set the hilite of cast "Power On" to TRUE
- ◆ put the name of cast B13 into buttonName
- ◆ set the picture of cast ("Picture" && n) →
to frameGrabber(mRecordToPicHandle)
- ◆ if the name of cast (i + 1) = EMPTY then exit

See also A11...H88 cast identifiers

castmembers

See the number of castmembers property.

the castNum of sprite

sprite property

Syntax the castNum of sprite *whichSprite*

Description This sprite property determines the number of the castmember associated with the sprite specified by the integer expression *whichSprite*. It is used with puppet sprites to change from one castmember to another.

The `castNum` sprite property can be tested and set.

Examples if the castNum of sprite 3 = A12 then play "home run"

- ◆ set the castNum of sprite (firstCar + i) to \neg the number of cast "Car"

Note If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command after making all of the changes.

See also number of cast `cast` property

the centerStage

property

Syntax the centerStage

Description This property determines if the Stage is centered on the monitor when a movie is loaded. If this property is `TRUE`, the Stage is centered. It is intended primarily for use with the MacroMind Player.

The `centerStage` property can be tested and set, and the default value is `TRUE`.

Examples if the centerStage = `FALSE` then \neg
go to frame "off center"

- ◆ set the centerStage to `FALSE`
- ◆ set the centerStage to (not the centerStage)

See also `fixStageSize` property

char...of

chunk expression keyword

Syntax char *whichCharacter* of *chunkExpression*

- ◆ char *firstCharacter* to *lastCharacter* of *chunkExpression*

Description This keyword is used to specify a character or a range of characters in a chunk expression.

The expressions *whichCharacter*, *firstCharacter*, and *lastCharacter* must evaluate to integers which specify a character in the chunk.

Chunk expressions are used to refer to any character, word, item, or line in any source of text. Sources of text include fields (text castmembers) and variables that hold strings.

Examples

```
put char 6 of "MacroMind"
-- "M"

♦ put char 6 to 9 of "MacroMind"
-- "Mind"

♦ put char 6 to 60 of "MacroMind"
-- "Mind"

♦ if char 60 of "MacroMind" = EMPTY then put "true"
-- "true"

♦ put "zzzzz" into char 1 to 5 of word 2 of line 3 ↵
  of field "Sleepy"

♦ put char offset(".", s) + 1 to length(s) of s ↵
  into fractionalPart
```

Note Characters include letters, numbers, punctuation marks, spaces, and control characters like TAB and RETURN.

See also word...of, item...of, and line...of chunk expression keywords; the number of chars in chunk function; chars, length, and offset functions

chars

See the number of chars in chunk function.

chars

function

Syntax chars (*stringExpression*, *firstCharacter*, *lastCharacter*)

Description This function returns a substring of characters from *stringExpression*, starting at *firstCharacter* and ending at *lastCharacter*. The expressions

firstCharacter, and *lastCharacter* must evaluate to integers which specify the position in the string.

If *firstCharacter* and *lastCharacter* are equal, then a single character is returned from the string. If *lastCharacter* is greater than the string length, only a substring up to the length of the string is returned. If *lastCharacter* is before *firstCharacter*, the function returns `EMPTY`.

Examples `put chars("MacroMind", 6, 6)`
 `-- "M"`

◆ `put chars("MacroMind", 6, 9)`
 `-- "Mind"`

◆ `put chars("MacroMind", 6, 60)`
 `-- "Mind"`

◆ `if chars("MacroMind", 60, 60) = EMPTY then put "true"`
 `-- "true"`

◆ `put chars(s, offset(".", s) + 1, length(s)) →`
 `into fractionalPart`

See also `char...of` chunk expression keyword; `length` and `offset` functions

charToNum

function

Syntax `charToNum(stringExpression)`

Description This function returns the ASCII code number corresponding to the first character of *stringExpression*.

Examples `put charToNum("A")`
 `-- 65`

◆ `if charToNum(nextChar) = 0 then foundNUL`

See also `numToChar` function

the checkBoxAccess

property

Syntax `the checkBoxAccess`

Description This property determines what happens when the user clicks a check box or radio button (created with the corresponding tools in the Tool window). There are three styles of interaction possible:

- 0 user can set check boxes and radio buttons *on* and *off* (this is the default)
- 1 user can set check boxes and radio buttons *on* but not *off*
- 2 user cannot set check boxes and radio buttons at all; they can only be set by scripts

The `checkBoxAccess` property can be tested and set, and the default value is 0.

Examples put the `checkBoxAccess` into `oldAccess`

◆ set the `checkBoxAccess` to 2

See also `hilite button` property; `checkBoxType` property

the `checkBoxType`

property

Syntax the `checkBoxType`

Description This property determines how check boxes indicate that they are selected. There are three styles:

- 0 an "x" (this is the default)
- 1 inset black rectangle
- 2 filled black rectangle

The `checkBoxType` property can be tested and set, and the default value is 0.

Examples put the `checkBoxType` into `oldType`

◆ set the `checkBoxType` to 1

See also `hilite button` property; `checkBoxAccess` property

the `checkMark of menuItem`

property

Syntax the `checkMark of menuItem` *whichItem* of menu *whichMenu*

Description This menu item property determines whether the specified menu item is displayed with a check mark. If it is `TRUE`, a check mark will appear next to the menu item; if it is `FALSE`, there will be no check mark. The *whichItem* expression can evaluate to either a menu item name or a menu item number; the *whichMenu* expression can evaluate to either a menu name or a menu number.

The `checkMark` property can be tested and set, and the default value is `FALSE`.

Example The following handler unchecks any items which are checked in the specified menu. For example, `unCheck("Format")` unchecks all the items in the Format menu.

```
on unCheck theMenu
    put the number of menuItems of menu theMenu into n
    repeat with i = 1 to n
        set the checkMark of menuItem i of menu theMenu
            to FALSE
    end repeat
end unCheck
```

See also name, number, enabled and script menu item properties;
name and number menu properties

the clickOn

function

Syntax the clickOn

Description This function returns the last active sprite clicked by the user. An active sprite is a sprite that has a sprite script associated with it.

If you want to detect when the user clicks a sprite with no script, you must assign a dummy script to it ("—" for example) so that it can be detected by the `clickOn`.

To detect a click on the Stage, test for `the clickOn = 0`.

Examples

- ◆ if the clickOn = 7 then alert "Sorry – wrong choice."
- ◆ put the clickOn into currentSprite
- ◆ set the foreColor of sprite (the clickOn) to brightRed

See also mouseDown, mouseUp, and doubleClick functions

closeDA

command

Syntax `closeDA`

Description This command closes all open desk accessories.

See also `openDA` command

closeResFile

command

Syntax `closeResFile [whichFile]`

Description This command closes the resource file specified by the string expression *whichFile*. If the resource file is in another folder than the current movie, *whichFile* must specify a pathname. If no file is specified, all open resource files are closed.

Examples `closeResFile "Special Fonts"`

- ◆ `closeResFile myDrive & "Hot Stuff:Cool Icons"`
- ◆ `closeResFile myResFile`
- ◆ `closeResFile -- close all open resource files`

Note It is good practice to close any file you have opened as soon as you are finished using it.

See also `openResFile` and `showResFile` commands

closeXlib

command

Syntax `closeXlib [whichFile]`

Description This command closes the Xlibrary file specified by the string expression *whichFile*. If the Xlibrary is in another folder than the current movie, *whichFile* must specify a pathname. If no file is specified, all open Xlibraries are closed.

XObjects, which are extensions to the Lingo language, are stored in Xlibrary files. Xlibrary files are resource files that contain XCOD (XObjects) resources. In addition, HyperCard XCMDs and XFCNs can be stored in Xlibrary files.

Examples `closeXlib "VideoDisc Xlibrary"`

- ◆ `closeXlib myDrive & "New Stuff:Transporter XObjects"`
- ◆ `closeXlib myXlib`
- ◆ `closeXlib -- close all open Xlibraries`

Note It is good practice to close any file you have opened as soon as you are finished using it.

See also `openXlib` and `showXlib` commands

the `colorDepth`

property

Syntax `the colorDepth`

Description This property is determines the color depth of the monitor on a color Macintosh. The values it can use are the following:

- 1 black-and-white
- 2 4 colors
- 4 16 colors
- 8 256 color
- 16 32,768 colors
- 32 16,777,216 colors

The `colorDepth` property can be tested and set, and the default value is the value set in the Monitors control panel.

Examples `if the colorDepth = 8 then play movie "Blue Moon"`

- ◆ `if the colorQD = TRUE then set the colorDepth to 8`

Note If you have multiple monitors, the `colorDepth` property refers to the monitor that the Stage is on. If the Stage spans more than one monitor, `colorDepth` indicates the greatest depth of those monitors; setting `colorDepth` will attempt to put all those monitors into the specified depth.

See also `colorQD` function; `switchColorDepth` property

the colorQD

function

Syntax the colorQD

Description This function returns a value indicating whether the Color QuickDraw software is available. It returns TRUE (1) for a color-capable Macintosh and FALSE (0) for a black-and-white-only machine.

Examples if the colorQD = TRUE then play movie "Color Movie"

♦ if the colorQD = TRUE then set the colorDepth to 8

Note A machine capable of displaying color may not have it switched on. This command is best used in conjunction with colorDepth.

See also colorDepth and switchColorDepth properties

the commandDown

function

Syntax the commandDown

Description This function returns TRUE if the Command key is being pressed.

Example when keyDown then →
if the commandDown then doCommandKey(the key)

See also key, optionDown, controlDown, and shiftDown functions

constrainH

function

Syntax constrainH(*whichSprite*, *integerExpression*)

Description This function returns the evaluated *integerExpression* constrained to the horizontal span of the sprite specified by the integer expression *whichSprite*.

For example, if the left and right coordinates of sprite 1 are 40 and 60, then constrainH would return the following values:


```

put constrainH(1, 20)
-- 40

put constrainH(1, 55)
-- 55

put constrainH(1, 100)
-- 60

```

That is, any number within the span of the sprite is left as is, while any number outside the span is forced to the nearest edge of the span.

The `constrainH` function returns the result of a comparison between *integerExpression* and the left and right coordinates of *whichSprite*. The `constrainH` and `constrainV` functions only constrain one axis each; the `constraint` sprite property limits both.

Example set the `locH` of `sprite horizontalSlider` to `constrainH(sliderGuide, the mouseH)`

See also `constrainV` function; left, right, and `constraint` sprite properties

the constraint of sprite

sprite property

Syntax the constraint of sprite *whichSprite*

Description This sprite property determines the constraints on the position of the sprite specified by the integer expression *whichSprite*. The constraint property itself is an integer which specifies another sprite whose bounding rectangle is used to constrain *whichSprite*.

The `constraint` property affects moveable sprites, and the `locH` and `locV` sprite properties. The constraint point of a moveable sprite that has a `constraint` set cannot be moved outside the bounding rectangle of the constraining sprite. (The constraint point is the registration point for a bitmap sprite and the top-left corner for a shape sprite.) For any sprite that has a `constraint` set, the constraint limits override any conflicting setting of the `locH` and `locV` sprite properties.

To remove a constraint property, set it to 0:

set the constraint of sprite *whichSprite* to 0

The constraint sprite property can be tested and set, and the default value is 0.

Examples if the constraint of sprite 3 \neq 0 then showConstraint

◆ set the constraint of sprite (i + 1) to boundarySprite

See also constrainH and constrainV functions; locH and locV sprite properties

constrainV

function

Syntax constrainV(*whichSprite*, *integerExpression*)

Description This function returns the evaluated *integerExpression* constrained to the vertical span of the sprite specified by the integer expression *whichSprite*.

For example, if the top and bottom coordinates of sprite 1 are 40 and 60, then constrainV would return the following:

```
put constrainV(1, 20)
-- 40

put constrainV(1, 55)
-- 55

put constrainV(1, 100)
-- 60
```

That is, any number within the span of the sprite is left as is, while any number outside the span is forced to the nearest edge of the span.

The constrainV function returns the result of a comparison between *integerExpression* and the top and bottom coordinates of *whichSprite*. The constrainH and constrainV functions only constrain one axis each; the constraint sprite property limits both.

Example set the locV of sprite verticalSlider →
to constrainV(sliderGuide, the mouseV)

See also constrainH function; top, bottom, and constraint sprite properties

contains

comparison operator

Syntax `stringExpression1 contains stringExpression2`

Description This operator compares two strings. When *stringExpression1* contains *stringExpression2*, the condition is TRUE (1); when *stringExpression1* does not contain *stringExpression2*, the condition is FALSE (0).

This is a comparison operator with a precedence level of 1.

Examples `put "MacroMind" contains "mind"`
`-- 1`

◆ `put "MacroMind" contains "micro"`
`-- 0`

◆ `if field "Interests" contains "animation" ↵`
`or field "Interests" contains "multimedia" then ↵`
`put "Buy a copy of MacroMind Director." ↵`
`into field "Advice"`

Note The string comparison is not sensitive to case or diacritical marks; "a" and "Å" are considered the same.

See also `starts` `comparison operator`; `offset` `function`

continue

command

Syntax `continue`

Description The `continue` command resumes animation after a pause.

Example `when keyDown then continue`

See also `pause` and `delay` commands; `pauseState` `function`

the controlDown

function

Syntax `the controlDown`

Description This function returns TRUE if the Control key is being pressed.

You can also test for the ASCII value of Control characters created with the combination of the Control key and one other alphanumeric key, using the Lingo `charToNum` function. Such combinations have a unique value in the ASCII table.

Example `when keyDown then ↵`
 `if the controlDown then doControlKey(the key)`

See also `key`, `optionDown`, `commandDown`, `shiftDown` and `charToNum` functions

cursor

command

Syntax `cursor whichCursor`

Description This command changes the cursor resource that is used over the entire Stage. The `cursor` command stays in effect until you turn it off by setting the cursor to zero.

The parameter *whichCursor* should evaluate to an integer which specifies the resource ID number of the cursor. The following cursors are always available:

- 0 no cursor set
- 1 arrow (pointer) cursor
- 1 I-beam cursor
- 2 crosshair cursor
- 3 crossbar cursor
- 4 watch cursor
- 200 blank cursor

To hide the cursor, set the `cursor` to 200 (a blank cursor resource).

Example `if status = "Please wait" then cursor 4`

Notes See `openResFile` for information about loading more cursor resources.

- ◆ During system events such as loading a file, the operating system may put up the watch cursor, and then change to the pointer cursor when returning control to the application. This will override the cursor command settings from the previous movie. Therefore, in a presentation

using a custom cursor with multiple movies, store any special cursor resource number as a global variable. Global Lingo variables stay resident in memory between movies. This will allow you to use the `cursor` command at the beginning of any new movie that is loaded.

See also `cursor` `sprite` property; `openResFile` command

the cursor of sprite

sprite property

Syntax the cursor of sprite *whichSprite*

Description This sprite property determines the cursor resource that is used when the pointer is over the sprite specified by the integer expression *whichSprite*. The `cursor` property stays in effect until you turn it off by setting the cursor to zero.

The `cursor` property is an integer which specifies the resource ID number of the cursor. The following cursors are always available:

- 0 no cursor set
- 1 arrow (pointer) cursor
- 1 I-beam cursor
- 2 crosshair cursor
- 3 crossbar cursor
- 4 watch cursor
- 200 blank cursor

To hide the cursor, set the `cursor` to 200 (a blank cursor resource).

The `cursor` sprite property can be tested and set.

Examples if the cursor of sprite 3 = 0 then setCursor

- ◆ set the cursor of sprite (i + 1) to grabber

Note See `openResFile` for information about loading more cursor resources.

See also `cursor` and `openResFile` commands

D

the date

function

Syntax the date
the short date
the long date
the abbreviated date
the abbrev date
the abbr date

Description This function returns the current date in the system clock as a string in one of three formats: short, long, or abbreviated. If no format is specified, the default is short. The abbreviated format can also be referred to as abbrev and abbr.

Examples put the short date
-- "9/7/91"

◆ put the long date
-- "Saturday, September 7, 1991"

◆ put the abbreviated date
-- "Sat, Sep 7, 1991"

◆ if char 1 to 4 of the date = "1/1/" →
then alert "Happy New Year!"

Note The three date formats vary, depending on the country for which your System file was designed. The examples above are for the United States.

See also time function

delay

command

Syntax delay numberOfTicks

Description This command causes the animation to halt for the given amount of time. The integer expression *numberOfTicks* specifies the number of ticks to wait. (There are 60 ticks per second.) No interactivity is possible during this time except for halting Lingo entirely with Command-Period (e.g., mouse clicks are ignored).

Examples `delay 2 * 60`

- ◆ `if the key = RETURN then delay random(180)`

Note The `delay` command does not function when the playback head is not moving. To set a halt in a handler, when the playback head is not moving, (in a `repeat...while` statement for example) use the command `startTimer` and test for the `timer` property.

See also `startTimer` command; `timer` property

delete

command

Syntax `delete chunkExpression`

Description This command deletes the specified chunk (character, word, item, or line) in any source of text. Sources of text include fields (text castmembers) and variables that hold strings.

Examples `delete word 1 of line 3 of field "Address"`

- ◆ `if char 1 of s = "$" then delete char 1 of s`

See also `field`, `char...of`, `word...of`, `item...of`, `line...of` chunk expression keywords; `hilite` text property

do

command

Syntax `do stringExpression`

Description This command evaluates *stringExpression* and executes the resulting string as a Lingo statement. This command is useful for evaluating expressions that the user has typed, and for executing commands stored in string variables, fields, arrays, and files.

Examples `do "beep 2"`

- ◆ `do "put" && userExpression && "into myVariable"`
- ◆ `do commandList(mGet, 3)`

done

See `play done` command.

dontPassEvent

command

Syntax `dontPassEvent`

Description This command is used only within a `when` event script to prevent passing an event to normal sprite handling.

Normal processing of a system mouse or keyboard event does the following:

For `mouseDown` events, Lingo looks for the sprite being clicked, and executes its cast or sprite script (the sprite script is given preference). It then executes any highlighting that is set for that sprite. For `keyDown` events, Lingo passes the character into the text area being edited (if any).

Example Consider a `mouseDown` event script

```
when mouseDown then myMouseHandler
```

in which `myMouseHandler` beeps if the Option key is down:

```
on myMouseHandler
  if the optionDown then beep
end myMouseHandler
```

Given this event script, when you Option-click a button, Lingo does the following: first any sprite or cast script is executed, then your event script makes the beep sound, and then any highlighting set on a sprite under the cursor is performed. You can prevent Lingo from executing the highlighting with the `dontPassEvent` command, as used in this version of `myMouseHandler`:

```
on myMouseHandler
  if the optionDown then
    beep
    dontPassEvent
  end if
end myMouseHandler
```


Now when you Option-click a button, the result is a beep and the sprite button is not highlighted.

Notes The `dontPassEvent` command only applies within event scripts (those starting with the word “when...”), or handlers called by an event script as in the example above. It does not have any effect elsewhere.

- ◆ The `dontPassEvent` command only applies to the current event being handled. It does not affect future events.

See also when `keyDown` and when `mouseDown` commands

the doubleClick

function

Syntax the doubleClick

Description This function returns `TRUE` if the last mouse event was a double-click and `FALSE` if it wasn't.

Example if the doubleClick then go to frame "open"

See also clickOn function

E

editableText

command

Syntax editableText

Description When this command is part of the Score script of a text sprite, that text can be edited as a text field while the movie is playing. The user can select, type, and delete text.

Notes The `editableText` command can only be used in the Score script of a text sprite. It has no effect elsewhere.

- ◆ You can also check Editable Text in a text castmember's Cast Info dialog box to make it editable wherever it appears in the movie.

See also text, selEnd, and selStart text properties

else

See `if...then` keywords.

EMPTY

character constant

Syntax `EMPTY`

Description This character constant represents the empty string, "", a string with no characters.

Examples `put length(EMPTY)`
 `-- 0`

- ◆ `set the text of field "Name" to EMPTY`

the enabled of menuItem

menu item property

Syntax `the enabled of menuItem whichItem of menu whichMenu`

Description This menu item property determines whether the specified menu item is displayed in plain text or dimmed, and whether it is selectable. If it is `TRUE` the menu item will appear in plain text and is selectable; if it is `FALSE` the menu item will appear dimmed and is not selectable. The *whichItem* expression can evaluate to either a menu item name or a menu item number; the *whichMenu* expression can evaluate to either a menu name or a menu number.

The `enabled` property can be tested and set, and the default value is `TRUE`.

Example The following handler enables or disables all the items in the specified menu. For example, `ableMenu("Special", FALSE)` disables all the items in the Special menu.


```

on ableMenu theMenu, flag
    put the number of menuItems of menu theMenu into n
    repeat with i = 1 to n
        set the enabled of menuItem i of menu theMenu to
            to flag
    end repeat
end ableMenu

```

See also name, number, checkMark and script menu item properties;
name and number menu properties

end

keyword

This keyword marks the end of handlers, methods, and multi-line control structures.

See if...then, repeat while, and repeat with keywords; on and method keywords; on mouseDown, on mouseUp, on keyDown, on startMovie, on stepMovie, on stopMovie, and on idle event handlers.

ENTER

character constant

Syntax ENTER

Description This character constant represents the Enter key, marked "enter" on the Macintosh keyboard.

Example when keyDown then ↵
if the key = ENTER then go to frame "do it"

exit

keyword

Syntax exit

Description This keyword exits from a handler, factory method, or macro and returns to the place from where the handler, method, or macro was called.

Example on setColors

```
    if the colorDepth = 0 then exit
    set the foreColor of sprite 1 to 35
    set the backColor of sprite 1 to 249
    set the foreColor of sprite 2 to 35
    set the backColor of sprite 2 to 249

end setColors
```

See also on, method, and macro keywords

exit repeat

keyword

Syntax exit repeat

Description This set of keywords jumps out of a repeat loop, to the statement following the end repeat statement. Control remains within the current handler, method, or macro.

Example The following handler finds the position of the first vowel in the string s:

```
on findVowel s
    repeat with i = 1 to the number of chars in s
        put char i of s into letter
        if letter = "a" then exit repeat
        if letter = "e" then exit repeat
        if letter = "i" then exit repeat
        if letter = "o" then exit repeat
        if letter = "u" then exit repeat
    end repeat
    return i
end findVowel
```

See also repeat while and repeat with keywords

the exitLock

property

Syntax the exitLock

Description This property determines whether the user can quit to the Finder when running the MacroMind Player. When the `exitLock` is `TRUE`, pressing Command-period or Command-w does not quit to the Finder from the MacroMind Player.

The `exitLock` property can be tested and set, and the default value is `FALSE`.

Examples if the commandDown and \neg
(the key = "." or the key = "w") and \neg
`exitLock` = `TRUE` then go to frame "quit sequence"

◆ set the `exitLock` to `TRUE`

F

factory

keyword

Syntax `factory` *factoryName*

```
method methodName1  
  [statements]  
end methodName1
```

```
method methodName2  
  [statements]  
end methodName2
```

[*more methods*]

Description This keyword defines a factory. A factory is composed of a related group of procedures, called methods, that can be used to create objects, send messages to other objects, and process messages. A factory is used to create internal objects that adhere to the same messaging syntax as XObjects.

Factories are also used to produce array objects, which can store an unlimited number of integers, floating point numbers, strings, symbols, and other objects, in any combination.

See Chapter 5 for more about factories, objects, methods, and instance variables.

Example See Chapter 6 for a detailed example of how to use factories.

See also `factory` **function**; `method`, `instance`, and `me` keywords; `mNew`, `mDispose`, `mPut`, `mGet`, `mPerform`, `mRespondsTo`, and `mInstanceRespondsTo` **predefined methods**

factory

function

Syntax `factory (factoryName)`

Description This function returns the factory or XObject specified by the string expression *factoryName*. If no factory or XObject is found with the given name, the `factory` function returns 0.

Example The three statements

```
put "AppleCD" into playerName
put factory(playerName) into playerFactory
put playerFactory(mNew) into cdPlayer
```

are equivalent to

```
put AppleCD(mNew) into cdPlayer
```

but allow the XObject's name to be easily changed under Lingo control.

See also `factory` **keyword**

fadeIn

See `sound fadeIn` command.

fadeOut

See `sound fadeOut` command.

FALSE

logical constant

- Syntax* FALSE
- Description* This logical constant represents the value of a logically false expression, such as `2 > 3`. It has a numerical value of 0.
- Example* set the `soundEnabled` to FALSE
- See also* TRUE logical constant

field

keyword

- Syntax* field *whichField*
- Description* This keyword refers to the text contained in the specified text castmember. A field can be either a source of text or a text destination.
- If *whichField* evaluates to a string, it is used as the cast name. If *whichField* evaluates to an integer, it is used as the cast number. (The cast identifiers A11 through H88 evaluate to integer cast numbers.)
- Examples*
- put field "President" into answer
 - ◆ put "George Washington" into field "President"
 - ◆ put field leftField into field rightField
 - ◆ if word 2 of field A13 = "Bush" then go to "correct"
 - ◆ put ":" after char 3 of field (status + 1)
- See also* cast keyword; char...of, word...of, item...of, and line...of chunk expression keywords

the fixStageSize

property

- Syntax* the fixStageSize
- Description* This property determines whether the Stage size remains the same when you load a new movie, regardless of the Stage size saved with that movie.

When the `fixStageSize` property is `TRUE`, the Stage size remains the same when you load a new movie. This property is primarily used with the MacroMind Player.

The `fixStageSize` property can be tested and set, and the default value is `TRUE`.

Examples `if the fixStageSize = FALSE then ↵`
 `go to frame "proper size"`

- ◆ `set the fixStageSize to FALSE`
- ◆ `set the fixStageSize to (not the fixStageSize)`

See also `centerStage` property

floatP

function

Syntax `floatP(expression)`

Description This function returns `TRUE` (1) if *expression* evaluates to a floating point number and `FALSE` (0) if it doesn't.

Examples `put floatP(3.0)`
 `-- 1`

- ◆ `put floatP(3)`
 `-- 0`

- ◆ `put floatP("3.0")`
 `-- 0`

- ◆ `if floatP(value(field "Entry")) = FALSE then ↵`
 `alert "Please enter a floating point number."`

Notes The "P" in `floatP` stands for "predicate."

- ◆ To force a literal value to be a floating point number, use a decimal point. To force a numerical variable to be floating point, add 0.0 to it.

See also `integerP`, `stringP`, `objectP`, and `symbolP` functions

the floatPrecision

property

- Syntax** the floatPrecision
- Description** This system property determines the number of total digits (after the decimal point) which are displayed for floating point numbers. The maximum is 19 significant digits.
- The floatPrecision property can be tested and set, and the default value is 4.
- Example**
- ```
set the floatPrecision to 4
put sqrt(3.0) into x
put x
-- 1.7321
set the floatPrecision to 8
put x
-- 1.73205081
```
- Note** As the example above shows, the floatPrecision determines only the number of digits with which floating point numbers are displayed, not the number of digits with which floating point calculations are performed.

## the foreColor of sprite

sprite property

- Syntax** the foreColor of sprite *whichSprite*
- Description** This sprite property determines the foreground color of the sprite specified by the integer expression *whichSprite*. The foreground color applies only to 1-bit bitmap and shape castmembers. It does not affect the display of a text or button castmember. An 8-bit bitmap is affected, but generally not in a useful way. Setting the foreColor in a Lingo script is equivalent to choosing the foreground color from the Tool window when the sprite is selected on the Stage.
- The value of a sprite's foreColor ranges from 0 to 255 for 8-bit color, or from 0 to 15 for 4-bit color. This number is actually the index number of the foreground color in the current palette. (You can click any color in the Palette window to see its index number in the window's lower left corner.)

The `foreColor` `sprite` property can be tested and set, although in order to set it with Lingo the `sprite` must be a puppet.

*Examples*    `put the foreColor of sprite 5 into oldColor`

- ◆ `set the foreColor of sprite (10 + random(3)) to 35`  
`-- color 35 in System palette is bright red`

*Note*    If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several `sprite` properties—or several `sprites`—you only have to use one `updateStage` command at the end of all the changes.

*See also*    `backColor` `sprite` property; `stageColor` property

## frame

---

See `go` and `play` commands.

## the frame

---

function

*Syntax*    `the frame`

*Description*    This function returns the number of the current frame being displayed in the current movie.

*Example*    `go to the frame - 1`

*See also*    `label` and `marker` functions

## framesToHMS

---

function

*Syntax*    `framesToHMS (frames, tempo, dropFrame, fractionalSeconds)`

*Description*    This function converts the specified frames to their equivalent in hours-minutes-seconds.

The integer expression *frames* specifies the number of frames.



The integer expression *tempo* specifies the tempo in frames per second.

The *dropFrame* argument is a logical expression, where `TRUE` means that this is a drop-frame, `FALSE` means that it is not. This argument is only meaningful if *fps* is set to 30 frames per second.

The *fractionalSeconds* argument is also a logical expression, which determines the handling of the residual frames. If it is set to `TRUE`, the residual frames are converted to a fraction of a second, to the nearest hundredth of a second; if it is set to `FALSE`, the residual frames are returned as an integer number of frames.

The resulting string uses the form: "`sHH:MM:SS.FFD`", where:

- s    "-" if the time is less than zero, or  
     space if the time is greater than or equal to zero
- HH   hours
- MM   minutes
- SS   seconds
- FF   fraction of a second if *fractionalSeconds* is `TRUE`, or  
     frames if *fractionalSeconds* is `FALSE`
- D    "d" if *dropFrame* is `TRUE`, or  
     space if *dropFrame* is `FALSE`

*Example*    `put framesToHMS(2710, 30, FALSE, FALSE)`  
              `-- " 00:01:30.10 "`

*See also*    `HMStoFrames` function

## the freeBlock

function

*Syntax*     `the freeBlock`

*Description*    This function returns the size of the largest free contiguous block of memory, in bytes. A kilobyte (KB) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

*Example*     `if the freeBlock < 10 * 1024 then ↵`  
                  `alert "Not enough memory!"`

*See also*     `freeBytes` and `memorySize` functions

## the freeBytes

function

*Syntax*    the freeBytes

*Description*    This function returns the total number of bytes of free memory. A kilobyte (KB) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

*Example*    if the freeBytes > 200 \* 1024 then ↵  
              play movie "colorMovie"

*See also*    freeBlock and memorySize functions

## G

### global

keyword

*Syntax*    global variable1 [, variable2] [, variable3]...

*Description*    This keyword declares a variable to be global.

A global variable can be declared by a script, a handler, a macro, or a method, and its value can be used by other scripts, handlers, macros, and methods. Global variables declared by factories can be reset by scripts, macros, and other objects that are created with the same factory, or by objects created with other factories.

To use a global variable inside a macro or factory method, you must declare it to be a global variable, otherwise the macro or factory method assumes it is a local variable.

This is why the macros used in the *Apartment* examples often include a second line that declares global variables, even though they have been set elsewhere. You must declare all the global variables the handler, macro, or method uses. Otherwise any variable the macro or method uses is a local variable, even if it has been declared a global variable by another script, handler, macro, or method.



For further information, see *Variables in the Setting and Communicating Values* section of Chapter 3.

**Example**    `global startingPoint`  
              `set startingPoint = whichMenu`

**See also**    `showGlobals`    **command**

## go

command

**Syntax**    `go [to] [frame] whichFrame`

- ◆ `go [to] movie whichMovie`
- ◆ `go [to] [frame] whichFrame of movie whichMovie`

**Description**    This command causes the playback head to jump to the specified frame of the specified movie. The expression *whichFrame* can evaluate either to a string marker label or to an integer frame number. The expression *whichMovie* must evaluate to a string which specifies a movie file. (If the movie is in another folder, *whichMovie* must specify the pathname.)

**Examples**    `go to "start"`

- ◆ `go to marker(0)`
- ◆ `go to the frame - 1`
- ◆ `go to movie "My Drive:More Movies:" & newMovie`
- ◆ `go to frame "elevated" of movie "Chicago"`

**Notes**    It's better to refer to marker labels instead of frame numbers, because the process of editing a movie (adding or deleting frames) can cause frame numbers to change. Thus a command like `go to frame 35` can become incorrect. It's also easier to read your script if you use marker labels. You can create markers by dragging them out of the triangular marker well in the upper left portion of the Score window (see the *MacroMind Director Studio Manual*).

- ◆ The `go to movie` command loads frame 1 of the movie. If the command is called from within a handler (or factory), the handler

in which it is placed continues executing. If you want the handler suspended during the playing of the movie, use the `play` command.

- ◆ The following are reset when loading a movie: the `keyDownScript`, the `mouseUpScript`, the `mouseDownScript`, the `beepOn`, and the constraint properties, the cursor and immediate sprite properties, the `puppetSprite` and cursor commands, and custom menus. Note, however, that the `timeoutScript` is not reset when loading a movie.

*See also* `marker` function; `play` command

## H

### the height of sprite

sprite property

*Syntax* the height of sprite *whichSprite*

*Description* This sprite property determines the vertical size in pixels of the sprite specified by the integer expression *whichSprite*. The `height` applies only to bitmap and shape castmembers. It does not affect text or button castmembers.

The `height` sprite property can be tested and set.

*Examples* set the height of sprite 10 to 26

- ◆ put the height of sprite (i + 1) into h

*Notes* Setting this property does not have any effect for bitmap sprites unless the sprite's `stretch` property is set to `TRUE`.

- ◆ If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

*See also* `width` and `stretch` sprite properties; `spriteBox` command



## hilite

command

- Syntax*    `hilite chunkExpression`
- Description*    This command highlights (selects) the specified chunk (character, word, item, or line) in a field on the Stage.
- Example*    `hilite word 2 of field "Thought For The Day"`
- See also*    `field`, `char...of`, `word...of`, `item...of`, `line...of` `chunk expression keywords`; `delete text property`

## the hilite of cast

button property

- Syntax*    `the hilite of cast whichCastmember`
- Description*    This button property determines the check state of a check box or radio button on the Stage. If it is `TRUE`, the check box or radio button is selected; if it is `FALSE`, the check box or radio button is not selected.
- If *whichCastmember* evaluates to a string, it is used as the cast name. If *whichCastmember* evaluates to an integer, it is used as the cast number. (The cast identifiers A11 through H88 evaluate to integer cast numbers.)
- The `hilite` text property can be tested and set, and the default value is `FALSE`.
- Examples*    `if the hilite of cast "2400 baud" = TRUE then ↵`  
                  `setBaudRate(2400)`
- ◆    `set the hilite of cast powerSwitch to TRUE`
- See also*    `checkBoxAccess` and `checkBoxType` properties

## HMStoFrames

function

- Syntax*    `HMStoFrames (hms, tempo, dropFrame, fractionalSeconds)`
- Description*    This function converts from hours-minutes-seconds to the equivalent number of frames.

The string expression *hms* specifies the time in the form "sHH:MM:SS.FFD", where:

- s    "-" if the time is less than zero, or  
space if the time is greater than or equal to zero
- HH   hours
- MM   minutes
- SS   seconds
- FF   fraction of a second if *fractionalSeconds* is TRUE, or  
frames if *fractionalSeconds* is FALSE
- D    "d" if *dropFrame* is TRUE, or  
space if *dropFrame* is FALSE

The integer expression *tempo* specifies the tempo in frames per second.

The *dropFrame* argument is a logical expression, where TRUE means that this is a drop-frame, FALSE means that it is not. If the string *hms* ends in a "d", the time is treated as a drop-frame, regardless of the value of *dropFrame*.

The *fractionalSeconds* argument is also a logical expression, which determines the meaning of the numbers following the seconds. If it is set to TRUE, the numbers specify a fraction of a second, to the nearest hundredth of a second; if it is set to FALSE, the numbers specify a residual number of frames.

*Example*    `put HMStoFrames(" 00:01:30.10 ", 30, FALSE, FALSE)`  
              `-- 2710`

*See also*    `framesToHMS` function

# I

## idle

---

See on `idle` event handler.



## if...then

keyword

**Syntax** `if logicalExpression then then-statement`

- ◆ `if logicalExpression then then-statement  
else else-statement`

- ◆ `if logicalExpression then  
then-statement1  
then-statement2  
...  
end if`

- ◆ `if logicalExpression then  
then-statement1  
then-statement2  
...  
else  
else-statement1  
else-statement2  
...  
end if`

**Description** The `if...then` statement evaluates the *logicalExpression*. If the condition is TRUE, it executes the *then-statement(s)*; if it is FALSE, it executes the *else-statement(s)*.

The `else` portion of the statement is optional. If you need to have more than one *then-statement* or *else-statement*, you must use the form ending with `end if`.

**Examples** `if the key = RETURN then continue`

- ◆ `if the colorQD = TRUE then play "Color Movie"  
else play "Black & White Movie"`
- ◆ `if the commandDown and the key = "q" then  
go to frame "exit"  
cleanUp  
quit  
end if`

- ◆ if the mouseDown then
  - set the foreColor of sprite s to 35
  - set the backColor of sprite s to 36
- else
  - set the foreColor of sprite s to 36
  - set the backColor of sprite s to 35
- end if

*Note* Note that when you use `else`, it always corresponds to the previous `if` statement. This means that sometimes a preceding `else` nothing statement may be required to associate an `else` keyword with the proper `if` keyword.

## the immediate of sprite

sprite property

*Syntax* the immediate of sprite *whichSprite*

*Description* This sprite property determines whether the Score script of the sprite specified by the integer expression *whichSprite* will execute on `mouseDown`. Usually sprite scripts don't execute until the user releases the mouse button over the sprite (for example, a `mouseUp` event). This gives the user a chance to move the pointer off the sprite (button) before letting up on the mouse button. In some cases, however, it is desirable to have the sprite's script execute as soon as the mouse button is pressed over the sprite. Setting this property for a given sprite does this.

Be sure to turn off this state when it is no longer appropriate. For example:

```
set the immediate of sprite 7 to FALSE
```

The `immediate` sprite property can be tested and set, and the default value is `FALSE`.

*Examples* put the immediate of sprite 7 into `soonerOrLater`

- ◆ set the immediate of sprite (i + 1) to `TRUE`

*Note* It is usually easier to use a `mouseDown` handler in a cast script.

*See also* on `mouseDown` event handler



## in

---

See the number of chars in, the number of items in, the number of lines in, **and** the number of words in functions.

## the ink of sprite

---

sprite property

**Syntax** the ink of sprite *whichSprite*

**Description** This sprite property determines the ink effect that the sprite specified by the integer expression *whichSprite* uses to display its castmember on the Stage.

The following ink effects are available:

- 0 Copy
- 1 Transparent
- 2 Reverse
- 3 Ghost
- 4 Not Copy
- 5 Not Transparent
- 6 Not Reverse
- 7 Not Ghost
- 8 Matte
- 9 Mask
- 32 Blend
- 33 Add Pin
- 34 Add
- 35 Subtract Pin
- 36 Background Transparent
- 37 Lightest
- 38 Subtract
- 39 Darkest

In the case of ink effect 36 (Background Transparent), you set the color which is defined as transparent with the Background Color Chip in the Tool window while the sprite is selected in the Score window. Another way to accomplish the same thing is to set the `backColor` property

of the sprite using Lingo, but this is unpredictable if the sprite has more than 1-bit color.

For further information on using ink effects, see the *MacroMind Director Studio Manual*.

The `ink` sprite property can be tested and set.

*Examples*    `put the ink of sprite 3 into currentInk`

◆    `set the ink of sprite (i + 1) to 8`

*Note*    If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

*See also*    `backColor` sprite property

## installMenu

command

*Syntax*    `installMenu castNumber`

*Description*    This command reads the definition of a menu bar from the given text castmember, and installs the menus defined there. These custom menus appear only while the movie is playing. To remove the custom menus, use the `installMenu` command with no argument, or 0 as the argument.

The menu definition in the text castmember uses the following syntax:

```
menu: menuName
 itemName ≈ script
 itemName ≈ script
...
menu: menuName
 itemName ≈ script
 itemName ≈ script
...
```

For an example of a custom menu bar definition and a table of special menu definition characters, refer to the `menu:` keyword.



*Examples*    `installMenu All`

- ◆ `installMenu (the number of cast "Menubar")`
- ◆ `installMenu 0    -- disable custom menus`

*See also*    `menu:` keyword

## instance

keyword

*Syntax*    `instance variable1 [ , variable2 ] [ , variable3 ]...`

*Description*    An instance variable is a special kind of variable used with factories. A factory can assign instance variables to specific objects. Instance variables contain a unique set of values for each individual object. The methods of a factory can use the instance variables.

An instance variable is available only to the factory object it is associated with. The value of an instance variable is established when an object is created, or when a method is used to change it. Each new object created by a factory has its own set of instance variable values which persist as long as the object itself persists.

To use an instance variable inside of a factory method, you must declare it with the `instance` keyword, otherwise the factory will assume it is a local (temporary) variable. Variables created inside a handler, macro, or a factory are assumed to be local, unless otherwise specified to be global or instance variables.

Instance variables need only be declared once in the factory—not in every method as is necessary with global variables.

An instance variable is usually defined in the `mNew` method of a factory. Subsequently, the values of instance variables can be changed by other methods.

To create instance variables in a factory, you use the `instance` keyword. For example, in the `mNew` method:

```
method mNew parameter1, parameter2
 instance variable1, variable2
 put parameter1 into variable1
 put parameter2 into variable2
end mNew
```

For further information, see the section on “Variables” Chapter 3, and Chapter 6.

*Example*    `method mNew`  
              `global counter`  
              `instance mySpeed, mySprite`  
              `put 0 into mySpeed`  
              `put counter into mySprite`  
  
              `end mNew`

*See also*    `factory`, `method`, and `global` keywords

## integer

function

*Syntax*    `integer (numericExpression)`

*Description*    This function returns the integer portion of *numericExpression*. The `integer` function rounds the value of *numericExpression* to the nearest whole integer.

*Examples*    `put integer(3.75)`  
              `-- 4`  
  
              ◆ `put integer(-3.25)`  
              `-- -3`  
  
              ◆ `set the locH of sprite 1 to`  
              `integer(0.333 * stageWidth)`

## integerP

function

*Syntax*    `integerP (expression)`

*Description*    This function returns TRUE (1) if *expression* evaluates to an integer and FALSE (0) if it doesn't.



*Examples*    `put integerP(3)`  
                   `-- 1`

◆ `put integerP(3.0)`  
                   `-- 0`

◆ `put integerP("3")`  
                   `-- 0`

◆ `if integerP(value(field "Entry")) = FALSE then ↵`  
                   `alert "Please enter an integer."`

*Note*    The “P” in `integerP` stands for “predicate”.

*See also*    `floatP`, `stringP`, `objectP`, and `symbolP` functions

## intersects

---

See `sprite...intersects` comparison operator.

## into

---

See `put...into` command.

## item...of

---

chunk expression keyword

*Syntax*    `item whichItem of chunkExpression`

◆ `item firstItem to lastItem of chunkExpression`

*Description*    This keyword is used to specify an item or a range of items in a chunk expression. An item chunk is any sequence of characters delimited by commas.

The expressions *whichItem*, *firstItem*, and *lastItem* must evaluate to integers which specify an item in the chunk.

Chunk expressions are used to refer to any character, word, item, or line in any source of text. Sources of text include fields (text castmembers) and variables that hold strings.

*Examples*    put item 3 of "red, yellow, blue green, orange"  
                   -- "blue green"

- ◆ put item 3 to 4 of "red, yellow, blue green, orange"  
       -- "blue green, orange"
- ◆ put item 3 to 10 of "red, yellow, blue green, orange"  
       -- " blue green, orange"
- ◆ put item 10 of "red, yellow, blue green, orange"  
       -- ""
- ◆ put "sharp cheddar" into item 4 of line 2 ↵  
       of field "Shopping List"
- ◆ if length(item i + 1 of list) > 31 then ↵  
       alert "Item too long"

*See also*    char...of, word...of, and line...of chunk expression keywords;  
               the number of items in chunk function

## items

---

See the number of items in chunk function.

## K

### key

---

function

*Syntax*    the key

*Description*    This function returns the last character typed.

*Example*    if the key = "q" then quit

*See also*    keyCode function



## the keyCode

function

- Syntax**    the keyCode
- Description**    This function returns a numerical code for the last key pressed. Useful for detecting when the user has pressed the arrow or function keys.
- Example**    if the keyCode = 123 then go to marker(-1)
- See also**    key function

## keyDown

See when keyDown then command.

## the keyDownScript

property

- Syntax**    the keyDownScript
- Description**    This property determines the string that is executed as a Lingo statement when a key is pressed.

Setting the `keyDownScript` property is equivalent to executing a `when keyDown then command`; both establish what Lingo will do when a `keyDown` event occurs. For example, the statements

```
set the keyDownScript to
 "if the key = RETURN then continue"
```

and

```
when keyDown then
 if the key = RETURN then continue
```

are equivalent.

When the event script you've assigned is no longer appropriate, turn it off with either of the following statements:

```
set the keyDownScript to EMPTY

when keyDown then nothing
```

The `keyDownScript` property can be tested and set, and the default value is `EMPTY`.

*Examples*    set the `keyDownScript` to  
              to "if the key = ENTER then beep"

◆ put the `keyDownScript` into `oldScript`

*See also*    when `keyDown` then **and** `dontPassEvent` commands

## L

### label

function

*Syntax*    `label (stringExpression)`

*Description*    This function returns the number of the frame associated with the given marker label. The *stringExpression* should evaluate to a label in the current movie.

*Examples*    go to `label("start") + 10`

◆ put `label(line n of the labelList)` into `whichFrame`

*See also*    marker function

### the labelList

function

*Syntax*    `the labelList`

*Description*    This function returns a string consisting of a list of the frame labels in the current movie, one label per line.

*Example*    put the `labelList` into field "Key Frames"

*See also*    label **and** marker functions



## the lastClick

function

*Syntax* the lastClick

*Description* This function returns the time in ticks (60ths of a second) since the mouse button was last pressed.

*Example* if the lastClick > 10 \* 60 then go to "no click"

*See also* lastEvent, lastKey, and lastRoll functions; startTimer command

## the lastEvent

function

*Syntax* the lastEvent

*Description* This function returns the time in ticks (60ths of a second) since the last mouse click, mouse roll, or key press.

*Example* if the lastEvent > 10 \* 60 then go to "help"

*See also* lastClick, lastKey, and lastRoll functions; startTimer command

## the lastKey

function

*Syntax* the lastKey

*Description* This function returns the time in ticks (60ths of a second) since the last key was pressed.

*Example* if the lastKey > 10 \* 60 then go to "no key"

*See also* lastClick, lastEvent, and lastRoll functions; startTimer command

## the lastRoll

function

*Syntax* the lastRoll

- Description** This function returns the time in ticks (60ths of a second) since the mouse was last moved.
- Example** if the lastRoll > 10 \* 60 then go to "no roll"
- See also** lastClick, lastEvent, and lastKey functions; startTimer command

## the left of sprite

sprite property

- Syntax** the left of sprite *whichSprite*
- Description** This sprite property indicates the left horizontal coordinate of the bounding rectangle of the sprite specified by the integer expression *whichSprite*.
- The left property can be tested, but it cannot be set directly. The left horizontal coordinate of a sprite can be set with the `spriteBox` command.
- Examples** if the left of sprite 3 < 0 then offLeftEdge
- ◆ put the left of sprite (i + 1) into leftMost
- Note** Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.
- See also** right, top, bottom, height, width, locH, and locV sprite properties; `spriteBox` command

## length

function

- Syntax** length(*stringExpression*)
- Description** This function returns the number of characters in the string that results from evaluating *stringExpression*.
- Examples** put length("Macro" & "Mind")  
-- 9



- ◆ if length(field "File Name") > 31 then →  
alert "That file name is too long."

**Note** Spaces count as characters. So do control characters like TAB and RETURN.

**See also** chars and offset functions

## line...of

chunk expression keyword

**Syntax** line *whichLine* of *chunkExpression*

- ◆ line *firstLine* to *lastLine* of *chunkExpression*

**Description** This keyword is used to specify a line or a range of lines in a chunk expression. A line chunk is any sequence of characters delimited by returns.

The expressions *whichLine*, *firstLine*, and *lastLine* must evaluate to integers which specify a line in the chunk.

Chunk expressions are used to refer to any character, word, item, or line in any source of text. Sources of text include fields (text castmembers) and variables that hold strings.

**Examples** put lines 1 to 4 of list into field "To Do"

- ◆ put "and" after word 2 of line 3 of s
- ◆ if line (i + 1) of field "Shopping List" →  
contains "cheese" then go to "dairy"

**See also** char...of, word...of, and item...of chunk expression keywords; the number of words in chunk function

## lines

See the number of lines in chunk function.



## the `lineSize` of sprite

sprite property

**Syntax** `the lineSize of sprite whichSprite`

**Description** This sprite property determines the thickness, in pixels, of the border of the sprite specified by the integer expression *whichSprite*. The `lineSize` applies only to shape sprites.

The `lineSize` sprite property can be tested and set.

**Examples** `put the lineSize of sprite 4 into thickness`

- ◆ `set the lineSize of sprite (border + 1) to 3`

## the `locH` of sprite

sprite property

**Syntax** `the locH of sprite whichSprite`

**Description** This sprite property determines the horizontal position on the Stage of the sprite specified by the integer expression *whichSprite*. It is the horizontal coordinate of the sprite's registration point. (See the *MacroMind Director Studio Manual* for information about registration points.)

The `locH` sprite property can be tested and set.

**Examples** `if the locH of sprite 9 > (the stageRight - the stageLeft) then set the locH of sprite 9 to 0`

- ◆ `set the locH of sprite (slider + 1) to the mouseH`

**Notes** Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

- ◆ If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

**See also** `locV`, `left`, `right`, `top`, `bottom`, `height` and `width` sprite properties; `updateStage` command



## the locV of sprite

sprite property

**Syntax** the locV of sprite *whichSprite*

**Description** This sprite property determines the vertical position on the Stage of the sprite specified by the integer expression *whichSprite*. It is the vertical coordinate of the sprite's registration point. (See the *MacroMind Director Studio Manual* for information about registration points.)

The locV sprite property can be tested and set.

**Examples** if the locV of sprite 9 > (the stageBottom -  
- the stageTop) then set the locV of sprite 9 to 0

- ◆ set the locV of sprite (slider + 1) to the mouseV

**Notes** Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

- ◆ If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

**See also** locH, left, right, top, bottom, height and width sprite properties; `updateStage` command

## long

See the date and time functions.

## M

## the machineType

function

**Syntax** the machineType

*Description* This function returns an integer that indicates the kind of Macintosh that is currently being used:

1 = Macintosh 512Ke  
2 = Macintosh Plus  
3 = Macintosh SE  
4 = Macintosh II  
5 = Macintosh IIfx  
6 = Macintosh IIcx  
7 = Macintosh SE/30  
8 = Macintosh Portable  
9 = Macintosh IIfx  
11 = Macintosh IIfx  
15 = Macintosh Classic  
16 = Macintosh IIsi  
17 = Macintosh LC  
256 = IBM PC-type machine

*Example* if the machineType = 15 then play "Classic Movie"

*See also* colorQD function; colorDepth property

## macro

keyword

*Special Note* The `macro` keyword is retained in Director 3.0 to maintain compatibility with scripts developed under Version 2.0. When writing new scripts, or editing old scripts, you should use handlers instead of macros. (Handlers are defined with the `on` keyword.)

*Syntax* -- [comment]  
macro *macroName* [*argument1*] [, *argument2*] [, *argument3*]...  
[*statement1*]  
[*statement2*]  
...

*Description* This keyword defines a macro. A macro is a multiple-line script defined in the Text window. Macros can accept arguments (inputs) and optionally return a result. Macros can call other macros and can be called from any other script or factory.



**Note** The first line of a castmember in the Text window that contains a macro must be a comment (--). You can define more than one macro in a given text castmember. The macro definition ends where the next macro (or factory) begins.

**See also** on keyword

## marker

function

**Syntax** marker(*integerExpression*)

**Description** This function returns the frame number of markers before or after the current frame. This can be useful for implementing a “next” or “previous” button, or for setting up an animation loop.

The *integerExpression* can evaluate to any positive or negative integer or zero. For example,

- marker(2) returns the frame number of the second marker after the current frame;
- marker(1) returns the frame number of the first marker after the current frame;
- marker(0) returns the frame number of the current frame, if the current frame is marked, or the frame number of the previous marker if the current frame is not marked;
- marker(-1) returns the frame number of the first marker before the current frame;
- marker(-2) returns the frame number of the second marker before the current frame.

**Examples** go to marker(0)

- ◆ go to marker(random(4))
- ◆ put marker(1) into nextMarkedFrame

**See also** frame and label functions

## mAtFrame

special message

**Syntax**    method mAtFrame frameNumber, subFrameNumber

          [statements]

end mAtFrame

**Description**    This special message is used by Lingo in conjunction with any XObject or factory-produced object which has been assigned to the `perFrameHook` property, as follows:

          set the `perFrameHook` to *objectName*

Subsequently, the `mAtFrame` message is automatically sent to the object every time the playback head reaches a new frame, or every time an internal subFrame is reached within a visual transition.

The functionality within `mAtFrame` must be supplied by the XObject or factory definition (as opposed to predefined methods). That is why `mAtFrame` is technically called a *message*, instead of a *predefined method*.

The `perFrameHook` property is primarily designed to be used with XObjects which need to be called at every subframe, such as frame-by-frame video recorders. Factory-produced objects should generally use an `on stepMovie` movie handler if they need to be called at every frame.

**See also**    factory and method keywords; `perFrameHook` property;  
          on `stepMovie` movie handler

## mDescribe

predefined method

**Syntax**    `XObjectName(mDescribe)`

**Description**    This predefined method is used only with XObjects (as opposed to factory-produced objects). The purpose of `mDescribe` is to create a list of methods in the Message window. This list contains the names of other methods of the XObject, plus any comments by the programmer of the XObject which document the functionality or syntax of these methods.



*Example*    fileIO(mDescribe)  
               -- Factory: SerialPort Id:200  
               -- SerialPort, Tool, Version 1.1, 9/24/90  
               -- © 1989, 1990 MacroMind, Inc.  
               -- by John Thompson and Jeff Tanner.  
               II     mNew, port  
               X     mDispose  
               I     mGetPortNum  
               IS    mWriteString, string  
               II    mWriteChar, charNum  
               S     mReadString  
               I     mReadChar  
               I     mReadCount  
               X     mReadFlush  
               III   mConfigChan, driverNum, serConfig  
               IIII  mHShakeChan, driverNum, CTSenable, CTScharNum  
               IIII  mSetUp, baudRate, stopBit, parityBit  
               ----

*Note*     You only use this method while you are authoring, typing the XObject name and the message `mDescribe` directly into the message window. Do not include this syntax as part of scripts within a movie.

*See also*   mMessageList   predefined method

## mDispose predefined method

*Syntax*    *object* (mDispose)

*Description*   This predefined method is used to destroy an object which was created earlier with the `mNew` method. It is used to dispose of both factory-produced objects and instances of XObjects. Use it to free up memory when an object is no longer needed.

*Examples*    on cleanUp  
                   global myObject  
                   if objectP(myObject) then myObject(mDispose)  
               end cleanUp

- ◆ on initialize

```
global myObject
if objectP(myObject) then myObject(mDispose)
put myFactory(mNew) into myObject

end initialize
```

*Notes* It is also recommended that before creating new instances of an object using `mNew`, you check for previous instances of an object with the same name, and `mDispose` it before you create a new one. (The initialize handler above is an example of this.) In this way, if the movie is aborted before the normal `mDispose`, you won't fill up memory by repeatedly creating new ones. This can happen during the development of a project, when you repeatedly stop it before the end, and play it again from the beginning.

- ◆ If you define an `mDispose` method in a factory, it will be executed instead of the predefined method. The result—which is seldom what you want—is that the object will not really get disposed. If you need to perform various housekeeping actions before disposing, put the routines in a method with another name, like `mRelease`.

*See also* `mNew` predefined method

## me

keyword

*Syntax* `me (messageName [, argument1 ][, argument2 ] ...)`

*Description* This keyword is used only inside methods. It evaluates to the current object that is processing the method. It is useful when a method needs to call other methods of the same object.

*Example* The following `mNew` method for an object stores some status strings inside the object's array, by calling its own `mPut` method:

```
method mNew

 me(mPut, 1, "Stopped")
 me(mPut, 2, "Playing")
 me(mPut, 3, "Paused")

end mNew
```

*See also* factory and method keywords



## memorySize

function

**Syntax**    the memorySize

**Description**    This function returns the total amount of memory (in bytes) allocated to the program, whether in use or free. It is useful for checking minimum memory requirements. A kilobyte (KB) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

**Example**    if the memorySize < 500 \* 1024 then alert ↵  
              "There is not enough memory to run this movie."

**See also**    freeBlock and freeBytes functions

## menu

See name menu property; name, number, checkMark, enabled, and script menu item properties.

## menu:

keyword

**Syntax**    menu: menuName  
              itemName ≈ script  
              itemName ≈ script  
              ...  
              menu: menuName  
              itemName ≈ script  
              itemName ≈ script  
              ...  
              [more menus]

**Description**    This keyword is used to define the actual content of custom menus, in conjunction with the installMenu command. Menu definitions are typed in the Text window. You refer to a particular menu definition by its cast number.

The menu: keyword specifies the name of the menu. In the subsequent lines you can specify the menu items for that menu. You can have a script execute when the user chooses that item by putting the script

after the "≈" symbol (press Option-x to create the symbol). A new menu is defined by the subsequent occurrence of the `menu:` keyword.

You can use special characters to define custom menus:

| Symbol | Example       | Description/Command key                                      |
|--------|---------------|--------------------------------------------------------------|
| ≈      | (see above)   | Associates a script with the menu item (Option-x)            |
| @      | menu: @       | Creates the Apple symbol when you define your own Apple menu |
| (      | Save(         | Disables the menu item                                       |
| (-     | (-            | Creates a disabled line in the menu                          |
| !√     | !√Easy Select | Checks the menu with a check mark (Option-v)                 |
| <B     | Bold<B        | Sets the menu item's style to Bold                           |
| <I     | Italic<I      | Sets the style to Italic                                     |
| <U     | Underline<U   | Sets the style to Underline                                  |
| <O     | Outline<O     | Sets the style to Outline                                    |
| <S     | Shadow<S      | Sets the style to Shadow                                     |
| /      | Quit/Q        | Defines a command-key equivalent                             |

Special symbols should follow the item name, and precede the "≈" symbol. You can also use more than one special character to define a menu item. Using `<B<U`, for example, sets the style to Bold and Underline.

*Example* menu: File

Open/O ≈ go to frame "Open"

Close/W ≈ go to frame "Close"

(-

Quit/Q ≈ go to frame "Quit"

menu: Edit

Undo/Z ≈ go to frame "Undo"

*See also* installMenu command



## menuItem

---

See `name`, `checkMark`, `enabled`, and `script menu item` properties.

## menuItems

---

See `number menu item` property.

## menus

---

See `number menu` property.

## method

---

keyword

**Syntax** `method methodName [argument1] [, argument2] ...`

**Description** This keyword is used to define a method. A method is a special kind of handler that exists inside a factory, with its own special syntax. It uses Lingo to create expressions that are commands or functions. A method is a single-line script, or series of scripts, that handle different messages (or processes) for objects created with a factory or an XObject.

There are two kinds of objects: internal (created by factories) and external (created by XObjects). Both factories and XObjects create objects and use methods. The difference is that you define a factory's methods in the movie script, a castmember script, or the Text window; an XObject's methods are predefined in the XObject itself. To see an XObject's methods, type `XObjectName (mdescribe)` in the Message window.

Each object has its own set of messages created by its methods. Messages are the way objects communicate with each other and with the rest of Lingo. Messages are sent by an object's methods and provide all of the necessary functionality for each particular object's task. Methods are associated with the objects created by their factory or XObject. Each object can use all the methods in its factory or XObject.



A method is defined using the `method` keyword:

```
method messagename
```

For ease of reference, it is a good convention to begin *messagename* with a lower-case *m*.

The `mNew` method is used to create a new object. It can be left as a predefined method, or it can be specifically defined in a factory to add object initialization. It can be followed immediately by additional methods.

If you want to perform additional initialization (setup) for an object, you can attach arguments to `mNew` that serve to accept values for the object it creates:

```
method mNew [argument 1] [, argument 2]...
```

Typically, you use the `mNew` method to assign an object's instance variables.

For complete information, see the `factory` keyword and *Chapter 6: Using XObjects and Factories*.

*Example*    `method mNew a,b,c`

*See also*    `exit, factory, instance, return` keywords

## mGet

predefined method

*Syntax*    `object (mGet , whichElement)`

*Description*    This predefined method (which can only be used with factory-produced objects) retrieves data from an object's internal array. Every object produced by a factory has an associated array capable of storing an arbitrary number of integers, floating point numbers, strings, objects, or symbols. The elements of the array are numbered 1, 2, 3, .... The `mPut` predefined method is used to assign values to a particular element.

The integer expression *whichElement* specifies which array element the `mGet` method returns. If you retrieve an element which has not been assigned a value with the `mPut` method, the element will have the numerical value 0.



**Examples**

```
myObject(mPut, 3, 2 + 2)
myObject(mPut, 7, sqrt(2.0))
myObject(mPut, 12, "hello" && "there")
put myObject(mGet, 3)
-- 4
put myObject(mGet, 7)
-- 1.4142
put myObject(mGet, 12)
-- "hello there"
```

- ◆ `put myObject(mGet, i + 1) ↪`  
into line `i` of field `"Data"`

**Note** Different types of data can be stored in various elements of the same array. You can use the functions `integerP`, `floatP`, `stringP`, `symbolP`, and `objectP` to determine the data type of a particular element.

**See also** `mPut` predefined method

## `mInstanceRespondsTo`

predefined method

**Syntax** `XObject(mInstanceRespondsTo, message)`

**Description** This predefined method (which can only be used with `XObjects`) returns a positive integer if an instance of the `XObject` responds to the specified message, which must be a string or symbol expression. In this case the integer returned is the number of arguments required by the message, plus 1. The method returns 0 if `XObject` does not respond to the specified message.

**Examples**

```
put SerialPort(mInstanceRespondsTo, "mWrite")
-- 3

◆ put factory("SerialPort") into theXObject
put theXObject(mInstanceRespondsTo, #mWrite)
-- 3
```

**See also** `mRespondsTo` predefined method

## mMessageList

predefined method

*Syntax*    *XObject* (mMessageList)

*Description*    This predefined method (which can only be used with XObjects) returns a string which describes the XObject and its methods. (It is the same string that the mDescribe method displays in the Message window.)

*Example*

```
put fileIO(mMessageList)
-- Factory: SerialPort Id:200
-- SerialPort, Tool, Version 1.1, 9/24/90
-- © 1989, 1990 MacroMind, Inc.
-- by John Thompson and Jeff Tanner.
II mNew, port
X mDispose
I mGetPortNum
IS mWriteString, string
II mWriteChar, charNum
S mReadString
I mReadChar
I mReadCount
X mReadFlush
III mConfigChan, driverNum, serConfig
IIII mHShakeChan, driverNum, CTSenable, CTScharNum
IIII mSetUp, baudRate, stopBit, parityBit

```

*See also*    mDescribe predefined method

## mName

predefined method

*Syntax*    *XObject* (mName)

◆ *XObjectInstance* (mName)

*Description*    This predefined method (which can be used only with XObjects and their instances) returns a string containing the name of the XObject which created the instance.

*Examples*

```
put SerialPort(mNew, 0) into modemPort
put modemPort(mName)
-- "SerialPort"
```



- ◆ set the text of cast All to laserDiscObj (mName)
- ◆ if laserDiscObj (mName) = "PioneerLaserDisc" then  
     go to movie "Pioneer Demo"  
   else  
     go to movie "Sony Demo"  
   end if

*See also*    factory function

## mNew

predefined method

*Syntax*    factory (mNew [, argument1] [, argument2] ...

- ◆ XObject (mNew [, argument1] [, argument2] ...

*Description*    This predefined method is used to create *factory objects* or instances of an *external XObject* in RAM. To create the instance of a particular class of objects, you assign an object variable to the particular factory or XObject name using the mNew method.

Arguments to the mNew method are optional. Of course, a particular XObject may have been written to require a certain number of arguments of a certain type. See its documentation, its example movie, or its mDescribe in the message window for this information.

There is no requirement for any particular number of arguments to the mNew method of factory objects. Typically the mNew method of a *factory* object is where you assign *instance* variables used throughout the factory methods.

In order to clear the object you create using mNew from RAM at the end of the presentation, it is a good idea to use the predefined mDispose method for both *factory* objects and external *XObjects*.

*Examples*    put myArrayFactory (mNew) into myArray

- ◆ put birdFac (mNew, wingCastNum, legCastNum) into bird
- ◆ put PioneerLaserDisc (mNew, 1, 9600, 0) into vDisc
- ◆ put FileIO (mNew, "read", "textFile2") into readFile



- ◆ on startMovie

```

 global myObject
 -- check for previous instances:
 if objectP(myObject) then myObject(mDispose)
 -- create a new instance of the object in RAM:
 put myArrayFactory(mNew) into myObject
end startMovie

```

**Note** It is also recommended that before creating new instances of an object using `mNew`, you check for previous instances of an object with the same name, and `mDispose` it before you create a new one. In this way, if the movie is aborted before the normal `mDispose`, you won't fill up RAM by repeatedly creating new ones. This can happen during the development of a project, when you repeatedly stop it before the end, and play it again from the beginning.

**See also** `factory`, `method`, `instance`, `mDescribe`, `mDispose`, and Chapter 6

## mod

arithmetic operator

**Syntax** `integerExpression1 mod integerExpression2`

**Description** This operator performs the arithmetic modulus operation on two integer expressions. In this operation, *integerExpression1* is divided by *integerExpression2*. The resulting value of the entire expression is the integer remainder of the division.

This is an arithmetic operator with a precedence level of 4.

**Examples** `put 7 mod 4`  
`-- 3`

- ◆ `if (mySprite mod 2) = 1 then`  
`set the ink of sprite mySprite to 0`  
`else`  
`set the ink of sprite mySprite to 8`  
`end if`



## the mouseCast

function

*Syntax* the mouseCast

*Description* This integer function returns the cast number of the castmember under the mouse pointer at the time the function is called. If the mouse is not over a castmember, it returns -1.

*Examples* if the mouseCast = the number of cast "Off Limits" →  
then alert "Stay away from there!"

◆ put the mouseCast into lastCast

*See also* mouseChar, mouseItem, mouseLine, and mouseWord functions;  
the number of cast **cast property**

## the mouseChar

function

*Syntax* the mouseChar

*Description* If the mouse is over a field, this integer function returns the number of the character under the pointer at the time the function is called, counting from the beginning of the field. If the mouse is not over a field, it returns -1.

*Examples* if the mouseChar = -1 then →  
put "Please point to a character." →  
into field "Instructions"

◆ put char (the mouseChar) of field (the mouseCast) →  
into currentChar

*See also* mouseItem, mouseLine and mouseWord functions; char...of  
chunk expression keyword; the number of chars in **chunk**  
function

## mouseDown

See on mouseDown event handler and when mouseDown then  
command.

## the mouseDown

function

*Syntax* the mouseDown

*Description* This function returns TRUE if the mouse button is currently being pressed.

*Example* if the mouseDown then exit

*See also* mouseUp, mouseH, and mouseV functions; on mouseDown and on mouseUp event handlers; when mouseDown then and when mouseUp then commands

## the mouseDownScript

property

*Syntax* the mouseDownScript

*Description* This property determines the string that is executed as a Lingo statement when the mouse button is pressed.

Setting the mouseDownScript property is equivalent to executing a when mouseDown then command; both establish what Lingo will do when a mouseDown event occurs. For example, the statements

```
set the mouseDownScript to
 "if the optionDown then continue"
```

and

```
when mouseDown then
 if the optionDown then continue
```

are equivalent.

When the event script you've assigned is no longer appropriate, turn it off with either of the following statements:

```
set the mouseDownScript to EMPTY

when mouseDown then nothing
```

The mouseDownScript property can be tested and set, and the default value is EMPTY.



*Examples*    set the mouseDownScript  $\rightarrow$   
                  to "if the clickOn = 0 then beep"

◆    put the mouseDownScript into oldScript

*See also*    mouseUpScript **property**; when mouseDown then, when mouseUp then, and dontPassEvent **commands**

## the mouseH

function

*Syntax*    the mouseH

*Description*    This function returns the horizontal position of the mouse pointer.

*Examples*    set the locH of sprite 2 to the mouseH

◆    if abs(the mouseH - startH) > 10 then  $\rightarrow$   
          put TRUE into draggedEnough

*Note*    Mouse coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

*See also*    mouseV **function**; locH and, locV **sprite properties**

## the mouseItem

function

*Syntax*    the mouseItem

*Description*    If the mouse is over a field, this integer function returns the number of the item under the pointer at the time the function is called, counting from the beginning of the field. (An item is any sequence of characters delimited by commas.) If the mouse is not over a field, it returns -1.

*Examples*    if the mouseItem = -1 then  $\rightarrow$   
                  put "Please point to an item."  $\rightarrow$   
                  into field "Instructions"

◆    put item (the mouseItem ) of field (the mouseCast)  $\rightarrow$   
          into currentItem

*See also* mouseChar, mouseLine, and mouseWord functions; item...of chunk expression keyword; the number of items in chunk function

## the mouseLine

function

*Syntax* the mouseLine

*Description* If the mouse is over a field, this integer function returns the number of the line under the pointer at the time the function is called, counting from the beginning of the field. If the mouse is not over a field, it returns -1.

*Examples* if the mouseLine = -1 then ↵  
put "Please point to a line." ↵  
into field "Instructions"

- ◆ put line (the mouseLine) of field (the mouseCast) ↵  
into currentLine

*See also* mouseChar, mouseItem, and mouseWord functions; line...of chunk expression keyword; the number of lines in chunk function

## mouseUp

See on mouseUp event handler and when mouseUp then command.

## the mouseUp

function

*Syntax* the mouseUp

*Description* This function returns TRUE if the mouse button is currently not pressed.

*Example* if the mouseUp then exit

*See also* mouseDown, mouseH, and mouseV functions; on mouseDown and on mouseUp event handlers; when mouseDown then and when mouseUp then commands



## the mouseUpScript

property

*Syntax* the mouseUpScript

*Description* This property determines the string that is executed as a Lingo statement when the mouse button is released.

Setting the `mouseUpScript` property is equivalent to executing a `when mouseUp then command`; both establish what Lingo will do when a `mouseUp` event occurs. For example, the statements

```
set the mouseUpScript to
 to "if the optionDown then continue"
```

and

```
when mouseUp then
 if the optionDown then continue
```

are equivalent.

When the event script you've assigned is no longer appropriate, turn it off with either of the following statements:

```
set the mouseUpScript to EMPTY
when mouseUp then nothing
```

The `mouseUpScript` property can be tested and set, and the default value is `EMPTY`.

*Examples* set the mouseUpScript to  
 to "if the clickOn = 0 then beep"

◆ put the mouseUpScript into oldScript

*See also* `mouseDownScript` property; when mouseUp then, when mouseDown then, and `dontPassEvent` commands

## the mouseV

function

*Syntax* the mouseV

*Description* This function returns the vertical position of the mouse pointer.

*Example* set the locV of sprite 15 to the mouseV

- ◆ if abs(the mouseV - startV) > 10 then ¬  
put TRUE into draggedEnough

*Note* Mouse coordinates are expressed relative to the upper-left corner of the Stage. See “Working with Coordinates” in Chapter 9.

*See also* mouseH function; locH and locV sprite properties

## the mouseWord

function

*Syntax* the mouseWord

*Description* If the mouse is over a field, this integer function returns the number of the word under the pointer at the time the function is called, counting from the beginning of the field. If the mouse is not over a field, it returns -1.

*Examples* if the mouseWord = -1 then ¬  
put "Please point to a word." ¬  
into field "Instructions"

- ◆ put word (the mouseWord) of field (the mouseCast) ¬  
into currentWord

*See also* mouseChar, mouseItem, and mouseLine functions; word...of chunk expression keyword; the number of words in chunk function

## moveableSprite

command

*Syntax* moveableSprite

*Description* This command allows the user to drag a sprite around on the Stage. It only functions when it appears in a sprite script and when the playback head is in that frame.

*Example* moveableSprite

*See also* constraint sprite property; sprite...intersects and sprite...within comparison operators



## movie

---

See `go` and `play` commands.

## the movie

---

function

*Syntax*    `the movie`

*Description*    This string function returns the name of the currently open movie.

*Example*    `put the movie into field "Movie Name"`

*See also*    `pathName` function

## mPerform

---

predefined method

*Syntax*    `object(mPerform, message [, argument1] [, argument2] ...)`

*Description*    This predefined method is used to send an arbitrary message to any Lingo object. It is similar to the Lingo `do` command which executes a Lingo statement stored as a string. However, `mPerform` invokes a particular method of the specified object by sending that message to the object indirectly.

The way this is accomplished is as follows: The first argument to `mPerform` is a required argument called a “message expression.” This expression can be either in the form of either a string or symbol. This *message* specifies the name of the method to be invoked by the `mPerform` message.

Following this required first argument are optional additional arguments, which can be any data type, constant, or property used in the method to be invoked.

Typically, the object name is specified by use of the `me` keyword, since the typical use of `mPerform` is within a factory method which invokes one of several other methods.

A powerful use for `mPerform` is to eliminate a lot of `if...then` conditional tests within methods which call other methods.

**Example** If we create an instance of the SerialPort XObject with

```
put SerialPort(mNew, 0) into modemPort
```

then the statements

```
modemPort(mPerform, "mWriteChar", charNum)
```

and

```
modemPort(mWriteChar, charNum)
```

both invoke the `mWriteChar` method with the argument `charNum`.

*See also* `factory`, `method`, and `me` keywords

## mPut

predefined method

**Syntax** `object(mPut, whichElement, expression)`

**Description** This predefined method (which can only be used with factory-produced objects) puts data into an object's internal array. Every object produced by a factory has an associated array capable of storing an arbitrary number of integers, floating point numbers, strings, objects, or symbols. The elements of the array are numbered 1, 2, 3, .... The `mGet` predefined method is used to retrieve values from a particular element.

The integer expression *whichElement* specifies which array element the `mPut` method assigns. The value of *expression* is assigned to the specified element.

**Examples**

```
myObject(mPut, 3, 2 + 2)
myObject(mPut, 7, sqrt(2.0))
myObject(mPut, 12, "hello" && "there")
put myObject(mGet, 3)
-- 4
put myObject(mGet, 7)
-- 1.4142
put myObject(mGet, 12)
-- "hello there"

◆ myObject(mPut, i + 1, line i of field "Data")


```

*See also* `mGet` predefined method



## mRespondsTo

predefined method

**Syntax** *XObjectInstance* (mRespondsTo, *message*)

**Description** This predefined method (which can only be used with instances of XObjects) returns a positive integer if *XObjectInstance* responds to the specified message, which must be a string or symbol expression. In this case the integer returned is the number of arguments required by the message, plus 1. The method returns 0 if *XObjectInstance* does not respond to the specified message.

**Example**

```
put SerialPort(mNew, 0) into modemPort
put modemPort(mRespondsTo, "mWrite")
-- 3
```

**See also** mInstanceRespondsTo predefined method

## N

## the name of cast

cast property

**Syntax** the name of cast *whichCastmember*

**Description** This cast property determines the name of the specified castmember. If *whichCastmember* evaluates to a string, it is used as the cast name. If *whichCastmember* evaluates to an integer, it is used as the cast number. (The cast identifiers A11 through H88 evaluate to integer cast numbers.)

The name is a descriptive string assigned by the user. Setting this property is equivalent to entering a name in the Cast Info dialog box.

The name cast property can be tested and set.

- Examples*    set the name of cast "On" to "Off"
- ◆ set the name of cast A15 to "foo"
  - ◆ put the name of cast (i + 1) into itsName
- See also*    the number of cast **cast** property

## the name of menu

menu property

- Syntax*    the name of menu *whichMenu*
- Description*    This menu property returns a string containing the name of the specified menu. The expression *whichMenu* can evaluate to either a menu number or a menu name, but it is not particularly useful to get the name if you already know it.
- The name **menu** property can be tested but cannot be set directly. Use the **installMenu** command to set up a custom menu bar.
- Examples*    put the name of menu 1 into firstMenu
- ◆ The following handler returns a list of the menu names, one per line:
 

```

on menuList
 put EMPTY into list
 repeat with i = 1 to the number of menus
 put the name of menu i & RETURN after list
 end repeat
 return list
end menuList

```
- See also*    number **menu** property; name **menu item** property

## the name of menuItem

menu item property

- Syntax*    the name of menuItem *whichItem* of menu *whichMenu*
- Description*    This menu item property determines the text that appears in the specified menu item. The *whichItem* expression can evaluate to either a menu item name or a menu item number; the *whichMenu* expression can evaluate to either a menu name or a menu number.



The name menu item property can be tested and set.

**Examples**    put the name of menuItem 8 of menu "Edit" ↵  
                 into itemName

- ◆ set the name of menuItem "Open" of menu fileMenu ↵  
to "Open" && fileName

**See also**    number menu item property; name menu property

## not

logical operator

**Syntax**    not *logicalExpression*

**Description**    This operator performs a logical negation on a logical expression. When *logicalExpression* evaluates to TRUE, the result is FALSE (0). When *logicalExpression* evaluates to FALSE, the result is TRUE (1).

This is a logical operator with a precedence level of 5.

**Examples**    put not (1 < 2)  
                 -- 0

- ◆ put not (1 > 2)  
-- 1

- ◆ set the checkMark of menuItem "Bold" ↵  
of menu "Style" to not (the checkMark ↵  
of menuItem "Bold" of menu "Style")

**See also**    and and or logical operators

## nothing

command

**Syntax**    nothing

**Description**    This command does nothing at all. It is useful when you want to cancel a previous when...then command. Also, a nested if...then...else statement may require else nothing, since an else clause always belongs to the preceding if. (See the second example below.)



*Examples* when mouseDown then nothing

- ◆ if test1 then  
    if test2 then something2  
    else nothing -- need this!  
else something1

*See also* when keyDown then, when mouseDown then, when mouseUp then, and when timeOut then **commands**; if...then **keywords**

## the number of cast

cast property

*Syntax* the number of cast *whichCastmember*

*Description* This cast property indicates the cast number of the castmember specified by *whichCastmember*. The cast number is the numerical index of a castmember in the Cast window. The first castmember, A11, has cast number 1; the second, A12, has cast number 2; and so on.

If *whichCastmember* evaluates to a string, it is used as the cast name. If *whichCastmember* evaluates to an integer, it is used as the cast number, although it is pointless to get the cast number if you already know it.

The number cast property can be tested, but it cannot be set.

*Examples* put the number of cast "Power Switch" into n

- ◆ set the castNum of sprite 1 to  
    to the number of cast "Red Balloon"

*See also* the number of castmembers **property**; castNum **sprite property**

## the number of castmembers

property

*Syntax* the number of castmembers

*Description* This property indicates the number of the last castmember in the current movie. Some of the castmember slots may be empty, so you may have fewer actual castmembers.

The number of castmembers property can be tested, but it cannot be set.



**Example** The following handler returns a string containing a list of all the castmember names, one per line:

```
on castList
 put EMPTY into list
 repeat with i = 1 to the number of castmembers
 put the name of cast i & RETURN after list
 end repeat
 return list
end castList
```

**See also** the number of cast cast property

## the number of chars in

chunk function

**Syntax** the number of chars in *chunkExpression*

**Description** This function returns a count of the characters in a chunk expression.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

**Examples** put the number of chars ↵  
in "MacroMind, the multimedia company"  
-- 33

◆ put the number of chars in word i of s into n

**Note** Spaces count as characters. So do control characters like TAB and RETURN.

**See also** length function; number of items in, number of lines in, number of words in chunk functions; char...in chunk expression  
keyword

## the number of items in

chunk function

**Syntax** the number of items in *chunkExpression*

**Description** This function returns a count of the items in a chunk expression. An item chunk is any sequence of characters delimited by commas.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*    `put the number of items ↵`  
                   `in "MacroMind, the multimedia company"`  
                   `-- 2`

◆ `put the number of items in field "Choices" into n`

*See also*    `number of chars in, number of lines in, number of words in chunk functions; item...in chunk expression keyword`

## the number of lines in

chunk function

*Syntax*    `the number of lines in chunkExpression`

*Description*    This function returns a count of the lines in a chunk expression.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*    `put the number of lines ↵`  
                   `in "MacroMind, the multimedia company"`  
                   `-- 1`

◆ `put the number of lines in field "Choices" into n`

*See also*    `number of chars in, number of items in, number of words in chunk functions; line...in chunk expression keyword`

## the number of menuItems

menu property

*Syntax*    `the number of menuItems of menu whichMenu`

*Description*    This menu property indicates the number of menu items in the specified custom menu. The *whichMenu* parameter can evaluate to either a menu name or a menu number.



The number of menuItems menu property can be read but cannot be set directly. Use the installMenu command to set up a custom menu bar.

**Examples** put the number of menuItems of menu "File" →  
into fileItems

◆ put the number of menuItems of menu i into n

**See also** installMenu command; number of menus menu property

## the number of menus

menu property

**Syntax** the number of menus

**Description** This menu property indicates the number of menus installed in the current movie.

The number of menus menu property can be read but cannot be set directly. Use the installMenu command to set up a custom menu bar.

**Examples** if the number of menus = 0 then →  
installMenu (the number of cast "Menubar")

◆ put the number of menus into n

**See also** installMenu command; number of menuItems menu property

## the number of words in

chunk function

**Syntax** the number of words in chunkExpression

**Description** This function returns a count of the words in a chunk expression.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*    put the number of words →  
                   in "MacroMind, the multimedia company"  
                   -- 4

- ◆ The following handler reverses the words in a specified string:

```
on reverse wordList

 put EMPTY into list
 repeat with i = 1 to the number of words →
 in wordList
 put word i of wordList & " " before list
 end repeat
 delete char (the number of chars in list) of list
 return list

end reverse
```

For example:

```
put reverse("man eats dog")
-- "dog eats man"
```

*See also*    number of chars in, number of items in, number of lines  
               in **chunk functions**; word...in **chunk expression keyword**

## numToChar

function

*Syntax*    numToChar (*integerExpression*)

*Description*    This function returns a string containing the single character whose ASCII sequence number is the value of *integerExpression*. It is useful for interpreting data from outside sources that are presented as numbers rather than as characters.

*Examples*    put numToChar(65)  
                   -- "A"

- ◆ put numToChar(charToNum("A") + n) after alphabet

*See also*    charToNum **function**



# O

## objectP

function

**Syntax** `objectP (expression)`

**Description** This function returns TRUE (1) if *expression* evaluates to an object (either a factory-produced object or an instance of an XObject) and returns FALSE (0) if it doesn't.

**Example**

```
put SerialPort(mNew, 0) into modemPort
put objectP(modemPort)
-- 1
modemPort(mDispose)
put objectP(modemPort)
-- 0
```

**Note** The “P” in objectP stands for “predicate”.

**See also** integerP, floatP, stringP, and symbolP functions; mNew and mDispose predefined methods

## of

The word of is part of many Lingo properties, such as the foreColor of sprite, the number of cast, the name of menu, and so on.

## offset

function

**Syntax** `offset (stringExpression1, stringExpression2)`

**Description** This function returns the integer starting position of the first character of *stringExpression1* within *stringExpression2* (the first occurrence only). It returns 0 if *stringExpression1* is not found in *stringExpression2*.

*Examples*

```

put offset("mind", "MacroMind")
-- 6

♦ put offset("M", "MacroMind")
-- 1

♦ put offset("Micro", "MacroMind")
-- 0

♦ put char offset(".", s) + 1 to length(s) of s →
 into fractionalPart

```

*Notes* Spaces are counted as characters in both strings.

- ♦ The string comparison is not sensitive to case or diacritical marks; "a" and "Å" are considered the same.

*See also* chars and length functions; contains and starts comparison operators

## on

keyword

*Syntax* on *handlerName* [*argument1*] [, *argument2*] [, *argument3*] ...  
           [*statements*]  
 end *handlerName*

*Description* This keyword indicates the beginning of a handler. Handlers are collections of Lingo statements that you can execute by simply using the handler name. A handler can accept arguments as input values and return a value as a function result.

Handlers can be defined in Cast Scripts and in the Movie Script. A handler in a Cast Script can only be called by other handlers in the same Cast Script. A handler in the Movie Script can be called from anywhere.

*Example*

```

on annoy howMuch
 beep random(howMuch)
 alert "Aren't handlers fun?"
end annoy

```



*Note* Certain handlers, like a `mouseDown` handler in a Cast Script, are called automatically by Lingo in response to user events. Others, like a `stepMovie` handler in the Movie Script, are called automatically as part of the process of playing a movie.

*See also* `method` and `macro` keywords

## on idle

movie handler

*Syntax* `on idle`  
    `[statements]`  
`end idle`

*Description* If the Movie Script contains an `idle` handler, Lingo executes it repeatedly while the movie is playing, whenever it has no events to process.

*Example* `on idle`  
    `put the short time into field "Time"`  
`end idle`

*Note* This is a good place for the Lingo statements that you want executed as frequently as possible.

*See also* `on startMovie`, `on stepMovie`, and `on stopMovie` movie handlers

## on mouseDown

event handler

*Syntax* `on mouseDown`  
    `[statements]`  
`end mouseDown`

*Description* If a Cast Script contains a `mouseDown` handler, Lingo executes it each time the mouse button is pressed over a sprite displaying that castmember, unless the sprite has its own sprite script.

*Example*    on mouseDown  
              repeat while the stillDown  
                  incrementCounter  
              end repeat  
  
              end mouseDown

*See also*    on mouseUp event handler

## on mouseUp

event handler

*Syntax*    on mouseUp  
              [ *statements* ]  
              end mouseUp

*Description*    If a Cast Script contains a mouseUp handler, Lingo executes it each time the mouse button is released after being pressed over a sprite displaying that castmember, unless the sprite has its own sprite script.

*Example*    on mouseUp  
              go to marker(1)  
              end mouseUp

*See also*    on mouseDown event handler

## on startMovie

movie handler

*Syntax*    on startMovie  
              [ *statements* ]  
              end startMovie

*Description*    If the Movie Script contains a startMovie handler, Lingo executes it each time the movie starts playing, regardless of the position of the playback head.



*Example*    on startMovie

```
openResFile "Special Fonts"
openResFile "Special Cursors"
```

```
end startMovie
```

*Note*        This is a good place for the Lingo statements that initialize the state of your movie.

*See also*    on stepMovie, on stopMovie, and on idle movie handlers

## on stepMovie

movie handler

*Syntax*    on stepMovie

```
 [statements]
```

```
end stepMovie
```

*Description*    If the Movie Script contains a `stepMovie` handler, Lingo executes it each time the playback head moves to another frame in the movie while it is playing. This handler executes before anything else when the playback head moves.

*Example*    on stepMovie

```
 put the frame into field "Frame"
```

```
end stepMovie
```

*Notes*        This is a good place for the Lingo statements that you want executed once at every new frame.

- ◆ A `stepMovie` handler is generally easier to use than the `perFrameHook` property. However, `stepMovie` is not called at each subframe of a transition, so you must continue to use `perFrameHook` for video output purposes .

*See also*    on startMovie, on stopMovie, and on idle movie handlers;  
              `perFrameHook` property

## on stopMovie

movie handler

**Syntax**    on stopMovie  
              [statements ]  
              end stopMovie

**Description**    If the Movie Script contains a stopMovie handler, Lingo executes it each time the movie stops playing.

**Example**    on stopMovie  
              closeResFile "Special Fonts"  
              closeResFile "Special Cursors"  
              end stopMovie

**Note**    This is a good place for the Lingo statements that do cleanup work at the end of your movie.

**See also**    on startMovie, on stepMovie, and on idle movie handlers

## open

command

**Syntax**    open [whichDocument with] whichApplication

**Description**    This command launches the application specified by the string expression *whichApplication*. You can optionally specify a document for the application to open with the string expression *whichDocument*. If either is in a different folder than the current movie, specify a pathname.

**Examples**    open "MacWrite"  
              ♦ open "Finger Painting" with "MacPaint"  
              ♦ open plan with myDrive & "Applications:" & drawApp

**Notes**    If you are running MultiFinder there must be enough memory to run both MacroMind Director and the other application at the same time. Under Finder, when you quit the other application you opened, MacroMind Director reopens and reloads the document that you originally started from and begins playing from the beginning.

**See also**    openDA, openResFile, and openXlib commands



## openDA

command

*Syntax* openDA *DName*

*Description* This command opens a desk accessory.

The *DName* is the menu item name of any desk accessory installed in your System, or an expression that yields such a name.

*Examples* openDA "Calculator"

◆ openDA myFavoriteDA

*See also* closeDA command

## openResFile

command

*Syntax* openResFile *whichFile*

*Description* This command opens the resource file specified by the string expression *whichFile*. If the file is in another folder than the current movie, *whichFile* must specify a pathname.

You can use this command to make additional fonts and cursors available in your movies.

*Examples* openResFile "Special Fonts"

◆ openResFile myDrive & "Hot Stuff:Cool Icons"

◆ openResFile myResFile

*Notes* If the file is already open, openResFile has no effect.

◆ Do not use openResFile to open another application. (Its code resources will interfere with those of MacroMind Director.) Use a resource mover like ResEdit to move the resources you need to a separate resource file.

◆ It is good practice to close any file you have opened as soon as you are finished using it.

*See also* closeResFile and showResFile commands

## openXlib

command

**Syntax**    `openXlib whichFile`

**Description**    This command opens the Xlibrary file specified by the string expression *whichFile*. If the file is in another folder than the current movie, *whichFile* must specify a pathname.

Xlibrary files contain XObjects as XCOD resources. Unlike `openResFile`, `openXlib` makes these XObjects known to MacroMind Director. Using the `mNew` predefined method, you can then create instances of the XObjects in memory.

**Examples**    `openXlib "VideoDisc Xlibrary"`

- ◆ `openXlib "My Drive:New Stuff:Transporter XObjects"`
- ◆ `openXlib myXlib`

**Notes**    If the file is already open, `openXlib` has no effect.

- ◆ Do not use `openResFile` to open another application. (Its code resources will interfere with those of MacroMind Director.)
- ◆ `openXlib` also opens HyperCard XCMDs and XFCNs so you may use them with MacroMind Director. For more information, see Chapter 6. When you need to use an XCMD from more than one application in a presentation (such as the Farallon Fplay XCMD from both a HyperCard stack and a Director movie), it is better to use this command to open a link to the HyperCard stack, rather than to install the XCMD in both places with ResEdit. If you do that, a resource conflict occurs which results in a system beep.
- ◆ It is good practice to close any file you have opened as soon as you are finished using it.

**See also**    `closeXlib` and `showXlib` commands

## the optionDown

function

**Syntax**    `the optionDown`

**Description**    This function returns `TRUE` if the Option key is being pressed.



*Example* when keyDown then  $\neg$   
           if the optionDown then doOptionKey(the key)  
*See also* key, commandDown, shiftDown, and controlDown functions

## or logical operator

*Syntax* *logicalExpression1* or *logicalExpression2*  
*Description* This operator performs a logical OR operation on two logical expressions. When either *logicalExpression1* or *logicalExpression2* (or both) are TRUE, the result is TRUE (1). When both expressions are FALSE, the result is FALSE (0).

This is a logical operator with a precedence level of 4.

*Examples* put 1 < 2 or 1 > 2  
           -- 1

- ◆ put 1 > 2 and 2 > 3  
       -- 0
- ◆ if field "State" = "AK" or field "State" = "HI"  $\neg$   
       then alert "You're off the map!"

*See also* and and not logical operators

# P

## the pathName function

*Syntax* the pathName  
*Description* This function returns a string containing the full pathname of the folder in which the current movie is located.  
*Example* if the pathName contains "System" then beep  
*See also* movie function

## pause

command

*Syntax*    `pause`

*Description*    This command causes the playback head to halt. Typically, you would put the `pause` command in the Script channel of a frame, and then assign `continue` or `go` commands to one or more sprite scripts in that frame.

In most cases, using `pause` in a score script is recommended over looping on the same frame, or looping between two frames. The reason for this is that a `pause` is much less processor-intensive than moving the playback head to a frame. (Exceptions to this general rule are: when you need movable sprites or are using the `perFrameHook`, so keeping the playback head going to the same frame is required.)

*Example*    The following `mouseUp` handler for a button alternately pauses and continues the animation, like the pause button on a video cassette recorder:

```
on mouseUp
 if the 'pauseState = TRUE then
 continue
 else
 pause
 end if
end mouseUp
```

*See also*    `continue` command; `pauseState` function

## the pauseState

function

*Syntax*    `the pauseState`

*Description*    This function returns `TRUE` if the movie is currently paused.

*Example*    `if the pauseState = TRUE then continue`

*See also*    `pause` and `continue` commands



# the perFrameHook

property

*Syntax* the perFrameHook

*Description* The perFrameHook property designates an object (created by either a factory or an XObject) that is called every frame (or subframe) with a special message called mAtFrame. You specify what routines and procedures are used in mAtFrame.

The perFrameHook is powerful when you want to call a certain set of procedures (using mAtFrame) each frame. Without the perFrameHook, you would have to type this set of procedures (using a macro) into the Script channel of every single frame in which you wanted it to occur. With the perFrameHook, you need only set the proper object to the perFrameHook once and the procedures (contained in the mAtFrame method) will be executed at every frame. When you no longer want to use the perFrameHook, set it to 0 to turn it off. The perFrameHook is especially useful when recording animations frame-per-frame to videotape.

At every frame, the perFrameHook object is sent the mAtFrame message. Therefore, you must create a factory that defines an mAtFrame method (in the same factory that creates the object you set to the perFrameHook).

When recording frame-per-frame to videotape, you can define two arguments for mAtFrame that specify the frame and subframe (subframes occur during transitions; each change during the transition is a subframe):

```
method mAtFrame frame, subframe
```

The *frame* argument is sent for each frame and *subframe* is sent for each subframe. You can name the arguments whatever you like, if you prefer not to use *frame* or *subframe*. You can also define additional arguments for mAtFrame, whether you are recording frame-per-frame to videotape or not. For an example of how you can record frame-per-frame to videotape, see the *FramePerFrame* example on the *Device Control* folder.

*Examples* if the perFrameHook = 0 then beep 2

- ◆ set the perFrameHook to myObject
- ◆ set the perFrameHook to 0



- ◆ This factory lets you create objects which display the current frame and subframe numbers (when a transition is occurring) in the Message window:

```
factory myPerFrameHook
method mAtFrame n,sub
 put "at frame " & n & ", subframe " & sub
end mAtFrame
```

For more information, see Chapter 6.

*Note* The `perFrameHook` is primarily designed to be used with XObjects that have an `mAtFrame` argument. If you do use the `perFrameHook` with a factory, do not use the `updateStage` command or set the text property of a sprite. Otherwise, unexpected results could occur.

*See also* factory and method keywords

## the picture of cast

cast property

*Syntax* the picture of cast *whichCastmember*

*Description* This cast property determines the image displayed by a bitmap or PICT castmember.

The picture cast property can be tested and set.

*Examples* put the picture of cast "Sunset" into pict

- ◆ set the picture of cast i →  
to frameGrabber(mRecordToPicHandle)

*See also* type cast property

## play

command

*Syntax* play [frame] *whichFrame*

- ◆ play movie *whichMovie*
- ◆ play [frame] *whichFrame* of movie *whichMovie*



**Description** This command causes the playback head to jump to the specified frame of the specified movie. The expression *whichFrame* can evaluate either to a string marker label or to an integer frame number. The expression *whichMovie* must evaluate to a string which specifies a movie file. (If the movie is in another folder, *whichMovie* must specify a pathname.)

This is similar to the `go to` command, but with the `play` command, when the sequence being played is over, the playback head is automatically returned to the frame where the `play` command was called. If the `play` command is issued from a Script channel script, the playback head returns to the next frame; if the `play` command is issued from a sprite script, handler, or method the playback head returns to the same frame. A sequence is over when the playback head reaches the end of the movie, or the `play done` command is executed.

The `play` command can also be used for playing several movies from a single handler. The handler is suspended while each movie plays, but resumes when the movie is over. Contrast this with a series of `go` commands which, if called from a handler, play the first frame of each movie. The handler is not suspended while the movie plays but immediately continues executing.

**Examples** `play "blink"`

- ◆ `play marker(1)`
- ◆ `play movie "My Drive:More Movies:" & newMovie`
- ◆ `play frame "elevated" of movie "Chicago"`

**See also** `play done` and `go` commands; `marker` function

## play done

command

**Syntax** `play done`

**Description** This command indicates that the sequence being played is complete, when the current movie or sequence was started using the `play` or `go to` commands. The `play done` command causes the playback head to return to where the sequence was started from. If the `play` command is



issued from a Script channel script, the playback head returns to the next frame; if the `play` command is issued from a sprite script, the playback head returns to the same frame.

See also `play` command

## playAccel

command

**Syntax** `playAccel whichFile [, option1] [, option2] [, option3]...`

**Description** This command plays the MacroMind Accelerator document specified by the string expression *whichFile*. (If the document is in another folder, *whichFile* must specify a pathname.)

It works like the `play` command, in that when the Accelerator movie is finished, the playback head returns to where it was called from. If the `playAccel` command is issued from a Script channel script, the playback head returns to the next frame; if the `playAccel` command is issued from a sprite script, the playback head returns to the same frame.

The following options are available. They should be separated by commas.

| Option                         | Action                                                                                                         |
|--------------------------------|----------------------------------------------------------------------------------------------------------------|
| <code>byFrame</code>           | Read one frame at a time from disk.                                                                            |
| <code>click</code>             | Stop and pass on mouse event.                                                                                  |
| <code>clickStop</code>         | Stop and don't pass on mouse event.                                                                            |
| <code>loop</code>              | Play movie continuously.                                                                                       |
| <code>noFlush</code>           | Prevents the current interactive movie from being removed from memory when the Accelerator document is loaded. |
| <code>noSound</code>           | Don't play sound.                                                                                              |
| <code>noUpdate</code>          | Don't update screen at end of movie.                                                                           |
| <code>playRect, t, r, b</code> | Stop the movie when the pointer is moved outside of the coordinates (left, top, right, bottom).                |
| <code>repeat, n</code>         | Number of times to repeat.                                                                                     |
| <code>sync</code>              | Attempt to play in sync with scan rate of monitor.                                                             |
| <code>whatFits</code>          | Play only what fits into memory.                                                                               |
| <code>tempo, n</code>          | Set the tempo of the movie.                                                                                    |



*Examples*    `playAccel "Lungs.mma"`

- ◆ `playAccel "Bolitas.mma", loop, clickStop, whatFits`
- ◆ `playAccel "b.mma", playRect, 0, 0, 640, 240, repeat, 4`

*Note*    The Accelerator file must fit into the available memory.

*See also*    `play` command

## playFile

See `sound playFile` command.

## preLoad

command

*Syntax*    `preLoad`

- ◆ `preLoad toFrame`
- ◆ `preLoad fromFrame, toFrame`

*Description*    This command causes a preload of castmembers. Preloading stops when the memory is full or all of the specified castmembers have been loaded.

If it is used without arguments, the `preLoad` command causes a preload of all cast members used in the current frame to the last frame in the movie.

When it is used with one argument, *toFrame*, the `preLoad` command causes a preload of all castmembers used in the range of frames from the current frame to the frame *toFrame*, as specified by frame number or label name.

When it is used with two arguments, *fromFrame* and *toFrame*, the `preLoad` command causes a preload of all castmembers used in the range of frames from the frame *fromFrame* to the frame *toFrame*, as specified by frame number or label name.

*Examples*    `preLoad marker(1)`

- ◆ `preLoad "start", "the end"`
- ◆ `preLoad here, there`

*See also*    `preLoadCast` command



## preLoadCast

command

**Syntax**    `preLoadCast`

- ◆ `preLoadCast` *whichCastmember*
- ◆ `preLoadCast` *fromCastmember*, *toCastmember*

**Description**    This command causes a preload of castmembers. Preloading stops when the memory is full or all of the specified castmembers have been loaded.

When it is used without arguments, the `preLoadCast` command causes a preload of all cast members in the movie.

When it is used one argument, *whichCastmember*, the `preLoadCast` command causes a preload of that castmember, as specified by cast number or cast name.

When it is used two arguments, *fromCastmember* and *toCastmember*, the `preLoadCast` command causes a preload of all castmembers in the range from castmember *fromCastmember* to castmember *toCastmember*, as specified by cast number or cast name.

**Examples**    `preLoadCast "Airplane"`

- ◆ `preLoadCast` `logo`, `logo + 10`

**See also**    `preLoad` command

## printFrom

command

**Syntax**    `printFrom` *fromFrame* [, *toFrame*] [, *reduction*]

**Description**    This command prints whatever is displayed on the Stage in each frame starting at *fromFrame*. Optionally you can supply the *toFrame*, and the *reduction* (100, 50, or 25 percent).

**Examples**    `printFrom 1`

- ◆ `printFrom` `label("good part")`, `label("so-so part")`, 50



**Note** If printing at reduced size (that is, if *reduction* is less than 100), the document prints as a bitmap, so text does not print as sharply as it would at full size.

## the puppet of sprite

sprite property

**Syntax** the puppet of sprite *whichSprite*

**Description** This sprite property determines whether the sprite specified by the integer expression *whichSprite* is a puppet.

When a sprite is a puppet, it is controlled by Lingo scripts rather than by the Score. As an example, a script could cause a bitmap puppet flash on and off while it moves around the Stage. For more information on using puppets, see Chapter 3, and “Simple Puppets” in Chapter 4.

Setting the puppet sprite property is equivalent to using the `puppetSprite` command. For example, the statement

```
set the puppet of sprite 1 to TRUE
```

has the same effect as

```
puppetSprite 1, TRUE
```

The puppet sprite property can be tested and set, and the default value is FALSE.

**Examples** put the puppet of sprite 5 into `isPuppet`

◆ set the puppet of sprite (`i + 1`) to TRUE

**See also** `puppetSprite` command

## puppetPalette

command

**Syntax** `puppetPalette whichPalette [, speed] [, nframes]`

**Description** This command causes the Palette channel to act as a puppet. The current palette will now be controlled by Lingo and script commands will override any palette in the Palette channel of the Score.

The `puppetPalette` command sets the current palette to the palette `castmember` specified by the expression *whichPalette*. If *whichPalette* evaluates to a string, it specifies the cast name of the palette. If *whichPalette* evaluates to an integer, it specifies the cast number of the palette.

Optionally you can cause the palette to fade in by specifying the speed as an integer expression, with 1 meaning slow and 60 meaning fast. You can also cause the palette to fade in over several frames by specifying the number of frames as in integer expression.

A puppet palette remains in effect until you turn it off with the command `puppetPalette 0`; any successive palette changes in the Score are not obeyed.

*Examples*    `puppetPalette "Rainbow"`

- ◆ `puppetPalette "System", howFast`
- ◆ `puppetPalette customPalette, 15, -  
label("there") - label("here")`

## puppetSound

command

*Syntax*    `puppetSound castmemberName`

- ◆ `puppetSound menuItemNumber, submenuItemNumber`
- ◆ `puppetSound MIDIOption`
- ◆ `puppetSound 0`

*Description*    This command causes the Sound channel to act as a puppet. The sound will now be controlled by Lingo and script commands will override any sound in the Sound channel of the Score.

The `puppetSound` command causes the specified sound to start playing. To play a sound stored in the Cast, specify the *castmemberName*. To start an external *Sounds* file sound, specify the *menuItemNumber* and *submenuItemNumber*. You can also control a MIDI device by using the `puppetSound` command with the following options:



```
midiStart
midiStop
midiContinue
midiBeat, n
```

where  $40 \leq n \leq 280$

```
midiSong, n
```

where  $0 \leq n \leq 127$

```
midisongPointer, beat, measure
```

where  $1 \leq \textit{beat} \leq 4$  and  $1 \leq \textit{measure} \leq 1023$

The command `puppetSound 0` stops a sound from playing. It also turns off the puppet status of the sound and returns control of the sound to the Sound channel in the Score.

*Examples*    `puppetSound "samba"`

- ◆ `puppetSound 2, 5`
- ◆ `puppetSound midiSong, 3`

*Note*    The menu and item numbering displayed in the Sound menu is in hexadecimal format, but you must specify the numbers here in decimal. For numbers greater than 9, here is the conversion: A=10, B=11, C=12, D=13, E=14, F=15.

*See also*    `sound fadeIn`, `sound fadeOut`, and `sound playFile` commands

## puppetSprite

command

*Syntax*    `puppetSprite whichSprite, state`

*Description*    When *state* evaluates to `TRUE`, this command makes the sprite specified by the integer expression *whichSprite* into a puppet. This means that the sprite are no longer controlled from the Score, but is instead controlled by Lingo. Any sprite property can be controlled by Lingo. When *state* evaluates to `FALSE`, control is returned to the Score.

To make a sprite a puppet, you must specify the channel number it resides in. The initial properties of the puppet are taken from whatever sprite is in the channel of the frame the `puppetSprite` command is executed in. If there is no sprite in that frame, the puppet will be



invisible. Subsequent control of the sprite properties via Lingo can change the initial properties.

For more information on using puppets, see the sections “Sprites and Puppets” in Chapter 3 and “Simple Puppets” in Chapter 4.

*Examples*    `puppetSprite 15, TRUE`

◆ `puppetSprite i + 1, state`

*Note*    You must provide the command `puppetSprite whichSprite, FALSE` when you are finished with your puppet, otherwise unpredictable results can occur when the playback head returns to sprites in frames not intended to be puppets.

*See also*    `puppet sprite property`; `backColor`, `bottom castNum`, `constraint`, `cursor`, `foreColor`, `height`, `immediate`, `ink`, `left`, `lineSize`, `locH`, `locV`, `right`, `stretch`, `top`, `type`, and `width sprite properties`

## puppetTempo

command

*Syntax*    `puppetTempo framesPerSecond`

*Description*    This command sets the tempo to the value of the integer expression `framesPerSecond`. The maximum frames per second is 60.

*Examples*    `puppetTempo 30`

◆ `puppetTempo oldTempo + 10`

## puppetTransition

command

*Syntax*    `puppetTransition whichTransition [, time ] [, chunkSize ] [, changeArea]`

*Description*    This command performs the specified transition between the current frame and the next frame. The `puppetTransition` command must be executed in the frame before the destination frame (the frame prior to the frame where you would normally put the transition effect in the Transition channel).

The integer expression *whichTransition* specifies which transition is to be performed. These are listed in the table below.



The integer expression *time* is the time to complete the transition. It is specified in 1/4 seconds. The minimum is 0, the maximum is 120 (30 seconds).

The integer expression *chunkSize* tells how many pixels each step of the transition should be. If the chunk size (minimum 1, maximum 128) is smaller, the transition is smoother (but also slower).

To cause the transition to occur only over the **changing area** (whatever part of the revealed frame that is different from the previous frame), set *changeArea* to TRUE. The default for *changeArea* is TRUE.

| Code | Transition                | Code | Transition                    |
|------|---------------------------|------|-------------------------------|
| 01   | Wipe right                | 27   | Random rows                   |
| 02   | Wipe left                 | 28   | Random columns                |
| 03   | Wipe down                 | 29   | Cover down                    |
| 04   | Wipe up                   | 30   | Cover down, left              |
| 05   | Center out, horizontal    | 31   | Cover down, right             |
| 06   | Edges in, horizontal      | 32   | Cover left                    |
| 07   | Center out, vertical      | 33   | Cover right                   |
| 08   | Edges in, vertical        | 34   | Cover up                      |
| 09   | Center out, square        | 35   | Cover up, left                |
| 10   | Edges in, square          | 36   | Cover up, right               |
| 11   | Push left                 | 37   | Venetian blinds               |
| 12   | Push right                | 38   | Checkerboard                  |
| 13   | Push down                 | 39   | Strips on bottom, build left  |
| 14   | Push up                   | 40   | Strips on bottom, build right |
| 15   | Reveal up                 | 41   | Strips on left, build down    |
| 16   | Reveal up, right          | 42   | Strips on left, build up      |
| 17   | Reveal right              | 43   | Strips on right, build down   |
| 18   | Reveal down, right        | 44   | Strips on right, build up     |
| 19   | Reveal down               | 45   | Strips on top, build left     |
| 20   | Reveal down, left         | 46   | Strips on top, build right    |
| 21   | Reveal left               | 47   | Zoom open                     |
| 22   | Reveal up, left           | 48   | Zoom close                    |
| 23   | Dissolve, pixels fast*    | 49   | Vertical blinds               |
| 24   | Dissolve, boxy rectangles | 50   | Dissolve, bits fast*          |
| 25   | Dissolve, boxy squares    | 51   | Dissolve, pixels*             |
| 26   | Dissolve, patterns        | 52   | Dissolve, bits*               |

The dissolve-type transitions marked with an asterisk (\*) will not work on monitors that are set to 32 bits.

*Examples*   puppetTransition 1

- ◆ The following transition is a wipe from right lasting 1 second with a (medium) chunk size of 20:

puppetTransition 2, 4, 20

*Note*   There is not a direct relationship between a low *time* and a fast transition. The actual speed of the transition depends on the relation of *chunkSize* and *time*. As an example, if the *chunkSize* is one pixel, the transition will take a long time no matter how low the *time*, because the Macintosh has to do a lot of work. To make transitions occur faster you should use a larger *chunkSize*, instead of setting a shorter *time*.

## put

command

*Syntax*   put *expression*

*Description*   This command evaluates the *expression* and displays it in the Message window.

*Examples*   put 2+2  
-- 4

- ◆ put "hello"  
-- "hello"

- ◆ put the time  
-- "9:10 AM"

- ◆ put 3.1416 into pi  
put 2 \* pi  
-- 6.2832

*See also*   put...into, put...before, and put...after commands

## put...after

command

*Syntax*   put *expression* after *chunkExpression*

*Description*   This command evaluates a Lingo *expression*, converts the value to a string, and inserts the resulting string after a specified chunk in a text container. (If *chunkExpression* specifies a nonexistent target chunk, the



string value is inserted as appropriate into the text container.) The previous contents of the container remain.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*

```
put "fox dog cat" into s
put " elk" after word 2 of s
put s
-- "fox dog elk cat"

◆ put "The square root of 2.0 is " into field "Answer"
put sqrt(2.0) after field "Answer"
put field "Answer"
-- "The square root of 2.0 is 1.4142"
```

*See also* `put...before`, `put...into`, `char...of`, `word...of`, `item...of`, and `line...of` **chunk expression keywords**

## `put...before`

chunk expression keyword

*Syntax* `put expression before chunkExpression`

*Description* This command evaluates a Lingo *expression*, converts the value to a string, and inserts the resulting string before a specified chunk in a text container. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the text container.) The previous contents of the container remain.

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*

```
put "fox dog cat" into s
put "elk " before word 2 of s
put s
-- "fox elk dog cat"

◆ put 20.00 + 7.25 into field "Price"
put "$" before field "Price"
put field "Price"
-- "$27.25"
```



*See also* put...after, put...into, char...of, word...of, item...of, and line...of chunk expression keywords

## put...into

command

*Syntax* put *expression* into *variable*

- ◆ put *expression* into *chunkExpression*

*Descriptions* This command has two different usages.

- ◆ The first usage evaluates a Lingo *expression* and stores its value in a local, global, or instance *variable*. The value can be an integer, a floating point number, a string, an object, or a symbol; it resides unchanged in the variable.
- ◆ The second usage evaluates a Lingo *expression*, converts the value to a string, and uses the resulting string to replace a specified chunk in a text container. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the text container.)

Chunk expressions are used to refer to any character, word, item, or line in any container of text. Containers include fields (text castmembers) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

*Examples*

```
put sqrt(2.0) into x
put sqrt(2.0)
-- 1.4142
put x
-- 1.4142
```

- ◆ put SerialPort(mNew, 0) into modemPort
- ◆ put "fox dog cat" into s  
put "elk" into word 2 of s  
put s  
-- "fox elk cat"
- ◆ put 2 + 2 into field "Answer"

*Note* In Lingo, you can use set...to and set...= as well as put...into for variable assignments. However, since HyperTalk only allows set to be used with properties, its use with variables is not recommended.

*See also* put...after, put...before, char...of, word...of, item...of, and line...of chunk expression keywords; set command



# Q

## quit

command

**Syntax** quit

**Description** This command exits from MacroMind Director or MacroMind Player to the Finder.

**Example** if the key = "q" and the commandDown then quit

**See also** shutDown and restart commands

## QUOTE

character constant

**Syntax** QUOTE

**Description** This character constant represents the quote character. It is needed to refer to the literal quote character in a string, since the quote character itself is used by Lingo scripts to delimit strings.

**Example** put "Can you spell" && QUOTE & "Kalispell" →  
& QUOTE & "?"  
-- "Can you spell "Kalispell"?"

# R

## random

function

**Syntax** random(*integerExpression*)

**Description** This function returns a random integer from 1 to the value of *integerExpression*.

*Examples*    `put random(6) + random(6) into diceRoll`

◆    `delay random(maxDelay * 60)`

## repeat while

keyword

*Syntax*    `repeat while testExpression`  
              `[statements...]`  
              `end repeat`

*Description*    This construction repeatedly executes the *statements* (which can be more than one line) as long as *testExpression* is TRUE (or, more generally, non-zero).

*Example*    The following `mouseDown` handler for a castmember beeps as long as the mouse button remains down:

```
on mouseDown
 repeat while the stillDown
 beep
 end repeat
end mouseDown
```

*See also*    `repeat with`, `exit repeat`, and `exit` keywords

## repeat with

keyword

*Syntax*    `repeat with counter = start to finish`  
              `[statements...]`  
              `end repeat`

*Description*    This construction repeatedly executes the *statements* (which can be more than one line) with the specified *counter* variable taking on integer values starting with the *start* value and ending with the *finish* value, inclusive. The *counter* is incremented by 1 each time.

*Example*    The following handler sets all 24 sprites to be puppets:



```

on puppetize
 repeat with channel = 1 to 24
 puppetSprite channel, TRUE
 end repeat
end puppetize

```

*See also* repeat while, exit repeat, and exit keywords

## restart

command

*Syntax* restart

*Description* This command restarts the computer. It is equivalent to choosing Restart in the Finder's Special menu.

*Example* if the key = "r" and the commandDown then restart

*See also* quit and shutDown commands

## the result

function

*Syntax* the result

*Description* This function returns the value of the return expression in the last handler executed.

*Example* The following handler returns a random roll for two dice:

```

on diceRoll
 return random(6) + random(6)
end diceRoll

```

The two statements

```

diceRoll
put the result into roll

```

are equivalent to

```

put diceRoll() into roll

```

Note that

```
 put diceRoll into roll
```

does *not* call the handler; `diceRoll` here is considered a variable reference.

See also `return` keyword

## return

keyword

*Syntax* `return expression`

*Description* This keyword is used in handlers and methods that return values. It returns the value of *expression* and exits from a handler or method. The expression can evaluate to an integer, floating point number, string, object, or symbol.

*Example* The following handler returns the greater of two expressions:

```
on max a, b
 if a > b then
 return a
 else
 return b
 end if
end max
```

Here is an example of it in action:

```
put max(3, 7)
-- 7
```

*Note* When calling a handler or method which serves as a user-defined function and has a return value, you must use parentheses around the argument list. This is necessary even when there are no arguments, as in the `diceRoll` function handler discussed above under the entry “the return”.

See also `result` keyword



## RETURN

character constant

**Syntax** RETURN

**Description** This character constant represents the return key, marked “return”, on the main key block of the Macintosh keyboard. It can be used to check for the user pressing the return key, or to insert a RETURN character into a string.

**Examples** when keyDown then if the key = RETURN then continue

- ◆ alert "Last line in the file." & RETURN & ↵  
"Click Done to exit."

## the right of sprite

sprite property

**Syntax** the right of sprite *whichSprite*

**Description** This sprite property indicates the right horizontal coordinate of the bounding rectangle of the sprite specified by the integer expression *whichSprite*.

The right sprite property can be tested, but it cannot be set directly. The right horizontal coordinate of a sprite can be set with the `spriteBox` command.

**Examples** if the right of sprite 3 > (the stageRight ↵  
- the stageLeft) then offRightEdge

- ◆ put the right of sprite (i + 1) into rightMost

**Note** Sprite coordinates are expressed relative to the upper-left corner of the Stage. See “Working with Coordinates” in Chapter 9.

**See also** left, top, bottom, height, width, locH, and locV sprite properties; `spriteBox` command

## rollOver

function

**Syntax** rollOver(*whichSprite*)

**Examples** if rollOver(6) then go to frame "intro"

*Description* This function returns `TRUE` if the pointer is currently over the bounding rectangle of the sprite specified by the integer expression *whichSprite*.

*See also* `mouseCast` function

## S

### the script of menuItem

menu item property

*Syntax* the script of menuItem *whichItem* of menu *whichMenu*

*Description* This property determines the Lingo statement that is executed when the specified menu item is selected. The *whichItem* expression can evaluate to either a menu item name or a menu item number; the *whichMenu* expression can evaluate to either a menu name or a menu number.

When the menu is installed, the script is set to the text following the "≈" character in the menu definition.

The `script` property can be tested and set.

*Examples* put the script of menuItem "Go" of menu "Control" →  
into goScript

◆ set the script of menuItem "Go" of menu "Control" →  
to "goHandler"

*See also* `installMenu` command; `menu:` keyword; `checkMark` and  
enabled menu item properties

### selection

function

*Syntax* the selection

*Description* This function returns a string containing the highlighted portion of the currently editable text field. It is useful for testing what a user has selected in a text field.



*Example* if the selection = EMPTY then  $\neg$   
alert "Please select a word."

*See also* selStart and selEnd properties

## the selEnd

text property

*Syntax* the selEnd

*Description* This text property is used with selStart to determine a selection from the currently editable text, counting from the beginning character. selEnd defines the ending character.

For example, if the text is "abcdefghijklmnopqrstuvwxy<sup>z</sup>", you could select "cde" with the following statements:

```
set the selStart to 3
set the selEnd to 5
```

The selEnd text property can be tested and set, and the default value is 0.

*Examples* if the selEnd = the selStart then noSelection

◆ set the selEnd to the selStart + 20

*See also* selStart and text text properties; selection function; editableText command

## the selStart

text property

*Syntax* the selStart

*Description* This text property is used with selEnd to determine a selection from the currently editable text, counting from the beginning character. selStart defines the beginning character.

For example, if the text is "abcdefghijklmnopqrstuvwxy<sup>z</sup>", you could select "cde" with the following statements:

```
set the selStart to 3
set the selEnd to 5
```

The `selStart` text property can be tested and set, and the default value is 0.

*Examples* if the `selEnd` = the `selStart` then `noSelection`

- ◆ set the `selStart` to the `selEnd` - 20

*See also* `selEnd` and text text properties; selection function; `editableText` command

## set...to and set...=

command

*Syntax* set the *property* to *expression*

- ◆ set the *property* = *expression*
- ◆ set *variable* to *expression*
- ◆ set *variable* = *expression*

*Description* The `set...to` (or `set...=`) command serves to evaluate an expression and put the result into a property or a variable.

*Example* set the ink of sprite 3 to 8

- ◆ set the `soundEnabled` = not (the `soundEnabled`)
- ◆ set vowels to "aeiou"
- ◆ set `x` = `sqrt(3.0)/2.0`

*Note* Although in Lingo you can use `set...to` for variable assignments, it is not recommended. For consistency with HyperTalk, use `set...to` only to set properties.

*See also* instance, global, factory keywords

## setCallback

command

*Syntax* `setCallback XCMDname, value`



- Description* The `setCallBack` command specifies how Lingo handles unsupported callbacks from the HyperTalk XCMD or XFCN whose name is *XCMDname*.
- If *value* is `FALSE` (0), unsupported callbacks from the specified XCMD or XFCN are ignored.
- If *value* is `TRUE` (1), unsupported callbacks from the specified XCMD or XFCN cause a generic alert to be displayed.
- If *value* is an object created from a factory, unsupported callbacks from the specified XCMD or XFCN cause various messages to be sent to the object. See Chapter 8 for details.
- Example* `setCallBack SuperDuperXCMD, myCallBackObject`

## the shiftDown

function

- Syntax* `the shiftDown`
- Description* This function returns `TRUE` if the Shift key is being pressed.
- Example* `when keyDown then ↵  
if the shiftDown then doShiftKey(the key)`
- See also* `key`, `controlDown`, `optionDown`, and `commandDown` functions

## short

See the `date` and `time` functions.

## showGlobals

command

- Syntax* `showGlobals`
- Description* This command displays all global variables and open factories (including XObjects) in the Message window. It is useful for debugging scripts.
- See also* `showLocals` command; `global` keyword



## showLocals

command

*Syntax*    `showLocals`

*Description*    This command displays all local variables in the Message window. This command can only be used within handlers, macros, or factory methods. Local variables in handlers will have no value after the handler executes. This command is useful for debugging scripts.

*See also*    `showGlobals`    **command**

## showResFile

command

*Syntax*    `showResFile [whichFile ]`

*Description*    This command displays a list of resources in the resource file specified by the string expression *whichFile*. The file must be already open. If the resource file is in another folder than the current movie, *whichFile* must specify a pathname. If no file is specified, all open resource files are listed.

There may be many open resource files, and the listing may be very long. To cancel the listing, press the mouse button.

*Examples*    `showResFile "Special Fonts"`

- ◆ `showResFile myDrive & "Hot Stuff:Cool Icons"`
- ◆ `showResFile myResFile`
- ◆ `showResFile -- show all open resource files`

*See also*    `openResFile`, `closeResFile`    **commands**

## showXlib

command

*Syntax*    `showXlib [whichFile ]`

*Description*    This command shows all XObjects in *Xlibfilename* (it must be open), or all open Xlibraries if no file is specified. Xlibrary files are resource files that contain XCOD (XObjects) resources. If the file is in another folder than the current movie, specify the pathname.



The `mDescribe` method displays on-line documentation for an XObject. To use `mDescribe`, perform the following steps:

- 1 Type `showXlib` in the Message window and press RETURN. This displays all open Xlibrary resource files and all XObjects contained in those Xlibraries.
- 2 Using the list of XObjects displayed in the Message window, type `XObjectName (mDescribe)` and press RETURN. This will display the on-line documentation for that XObject.

*Examples* `showXlib "VideoDisc Xlibrary"`

- ◆ `showXlib myDrive & "New Stuff:Transporter XObjects"`
- ◆ `showXlib myXlib`
- ◆ `showXlib -- show all open Xlibraries`

*See also* `openXlib` and `closeXlib` commands

## shutDown

command

*Syntax* `shutDown`

*Example* if the key = "s" and the commandDown then shutDown

*Description* This command causes the computer to close all open applications and turn itself off (same as Shut Down in the Finder's Special menu).

*See also* `quit` and `restart` commands

## sound fadeIn

command

*Syntax* `sound fadeIn whichChannel`

- ◆ `sound fadeIn whichChannel, ticks`

*Description* This command fades in a sound in the specified sound channel over a period of frames or ticks. If ticks is specified, then the fade in will occur evenly over that period of time. If ticks is not specified, the default number of ticks is calculated as  $15 * (60 / (\text{Tempo setting}))$  based on the

Tempo setting for the first frame of the fade in. The fade in will continue at a pre-ordained rate until *ticks* has elapsed, or the sound in the specified channel changes.

*Example*    `sound fadeIn 1, 5 * 60`

*See also*    `sound fadeOut` command

## sound fadeOut

command

*Syntax*    `sound fadeOut whichChannel`

◆ `sound fadeOut whichChannel, ticks`

*Description*    This command fades out a sound in a sound channel over a period of frames or ticks. If *ticks* is specified, then the fade out will occur evenly over that period of time. If *ticks* is not specified, the default number of ticks is calculated as  $15 * (60 / (\text{Tempo setting}))$  based on the Tempo setting for the first frame of the fade out. The fade out will continue at a pre-ordained rate until *ticks* has elapsed, or the sound in the specified channel changes.

*Example*    `sound fadeOut 1, 5 * 60`

*See also*    `sound fadeIn` command

## sound playFile

command

*Syntax*    `sound playFile whichChannel, whichFile`

*Description*    This command plays the AIFF sound stored in *whichFile* in the sound channel specified by *whichChannel*.

*Example*    `sound playFile 1, "Jet Blast"`

*Note*    This requires System 6.0.7 or later to work, otherwise the sound playback will not occur. It only works with AIFF sound files.

*See also*    `sound stop` command



## sound stop

command

- Syntax** `sound stop whichChannel`
- Description** This command stops the playing of the sound playing in the specified channel.
- Example** `if soundBusy(1) then sound stop 1`
- See also** `soundBusy` function

## soundBusy

function

- Syntax** `soundBusy (whichChannel )`
- Description** This function returns `TRUE` (1) if the specified sound channel is busy playing a sound, or `FALSE` (0) if it isn't.
- Example** `if soundBusy(1) then sound stop 1`
- See also** `sound playFile` and `sound stop` commands

## the soundEnabled

property

- Syntax** `the soundEnabled`
- Description** This property determines whether the sound is on or off. `TRUE` means that the sound is on.
- The `soundEnabled` property can be tested and set, and the default value is `TRUE`.
- Examples** `put the soundEnabled into soundOn`
- ◆ `set the soundEnabled to not (the soundEnabled)`
- See also** `soundLevel` property

## the soundLevel

property

- Syntax** `the soundLevel`

*Description* This property determines the volume level of the sound that is played through the Macintosh's speaker. Settings range from 0 (no sound) to 7 (maximum sound volume).

The `soundLevel` property can be tested and set, and the default value is 7.

*Examples* put the `soundLevel` into `oldSound`

◆ set the `soundLevel` to 5

*Note* Now that Macintosh computers can produce multichannel sound, this property is becoming obsolete. Use the `volume sound` property instead to control the volume on a channel-by-channel basis.

*See also* `soundEnabled` property; `volume sound` property

## sprite

keyword

*Syntax* the *property* of *sprite* which *Sprite*

*Description* This keyword specifies to Lingo that the following integer expression evaluates to a sprite number. It is used with every sprite property.

*Examples* put the `locH` of `sprite 1` into `h`

◆ set the puppet of `sprite (i + 1)` to `TRUE`

*See also* `cast` keyword

## sprite...intersects

comparison operator

*Syntax* `sprite` *sprite1* intersects *sprite2*

*Description* This operator compares the position of two sprites. It is `TRUE` if the bounding rectangle of *sprite1* touches the bounding rectangle of *sprite2*.

If both sprites have matte ink, their actual outlines are used, not the bounding rectangles. A sprite's outline is defined by the non-white pixels that make up its border.

This is a comparison operator with a precedence level of 5.



*Example* `if sprite i intersects j then doCrash`

*See also* `sprite...within` comparison operator

## `sprite...within`

comparison operator

*Syntax* `sprite sprite1 within sprite2`

*Description* This operator compares the position of two sprites. It is `TRUE` if the bounding rectangle of *sprite1* is entirely inside the bounding rectangle of *sprite2*.

If both sprites have matte ink, their actual outlines are used, not the bounding rectangles. A sprite's outline is defined by the non-white pixels that make up its border.

This is a comparison operator with a precedence level of 5.

*Example* `if sprite mySprite within sprite boundary →  
then doInside`

*See also* `sprite...intersects` comparison operator

## `spriteBox`

command

*Syntax* `spriteBox whichSprite, left, top, right, bottom`

*Description* This command sets the bounding rectangle coordinates of the puppet sprite specified by the integer expression *whichSprite*. The `spriteBox` command gives you a way to set the left, top, right, and bottom sprite properties of a sprite directly without having to convert it into `locH`, `locV`, `width` and `height`. This is useful because the left, top, right, and bottom sprite properties cannot be set directly.

This command works only on puppet sprites. For bitmap sprites, the `stretch` property must be `TRUE` to use this command.

A sprite's coordinates will change based on its registration point. For bitmap sprites, it may be necessary to move the registration point (to the upper-left corner of the image, for example), in order to obtain proper results.

*Examples*    `spriteBox 3, 50, 50, 200, 250`

- ◆ `spriteBox mySprite, ↵`  
    `startH, startV, the mouseH, the mouseV`

*See also*    `left, right, top, bottom, width, height, stretch, and`  
             `puppet` **sprite properties**

## the sqrt

function

*Syntax*    `sqrt (n)`

- ◆ `the sqrt of n`

*Description*    This function returns the square root of *n*. If *n* is a floating point number, the result is a floating point number. If *n* is an integer, the result is rounded to the nearest integer.

*Examples*    `put sqrt(3.0)`  
              `-- 1.7321`

- ◆ `put the sqrt of 3`  
    `-- 2`
- ◆ `put sqrt(a*a + b*b) into c`

## the stageBottom

function

*Syntax*    `the stageBottom`

*Description*    This function—along with `stageLeft`, `stageRight`, and `stageTop`—indicates where the Stage is positioned on the desktop. It returns the bottom vertical coordinate of the Stage, relative to the upper-left corner of the main screen. The height of the Stage in pixels is given by the `stageBottom - the stageTop`.

*Example*    The following two statements position sprite 3 50 pixels from the bottom edge of the Stage:

```
put the stageBottom - the stageTop into ↵
stageHeight
set the locV of sprite 3 to stageHeight - 50
```



*Note* Sprite coordinates are expressed relative to the upper-left corner of the Stage. See “Working with Coordinates” in Chapter 9.

*See also* `stageLeft`, `stageRight`, and `stageTop` functions

## the `stageColor`

property

*Syntax* `the stageColor`

*Description* This property determines the color of the movie background.

The value of the `stageColor` ranges from 0 to 255 for 8-bit color, or from 0 to 15 for 4-bit color. You can click a color in the Palette window to see that color’s index number in the lower left corner of the window. Setting the `stageColor` in a Lingo script is equivalent to choosing the Stage color from the pop-up palette in the Panel window.

The `stageColor` property can be tested and set, and the default value is 0 (white).

*Examples* `put the stageColor into oldColor`

◆ `set the stageColor to 249`

*See also* `foreColor` and `backColor` sprite properties

## the `stageLeft`

function

*Syntax* `the stageLeft`

*Description* This function—along with `stageRight`, `stageTop`, and `stageBottom`—indicates where the Stage is positioned on the desktop. It returns the left horizontal coordinate of the Stage, relative to the upper-left corner of the main screen. If the Stage is in the upper-left corner of the main screen, this coordinate is zero.

*Example* `if the stageLeft < 0 then leftMonitorProcedure`

*Note* Sprite coordinates are expressed relative to the upper-left corner of the Stage. See “Working with Coordinates” in Chapter 9.

*See also* `stageRight`, `stageTop`, and `stageBottom` functions



## the stageRight

function

*Syntax*    the stageRight

*Description*    This function—along with stageLeft, stageTop, and stageBottom—indicates where the Stage is positioned on the desktop. It returns the right horizontal coordinate of the Stage, relative to the upper-left corner of the main screen's desktop. The width of the Stage in pixels is given by the stageRight - the stageLeft.

*Example*    The following two statements position sprite 3 50 pixels from the right edge of the Stage:

```
put the stageRight - the stageLeft into stageWidth
set the locH of sprite 3 to stageWidth - 50
```

*Note*    Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

*See also*    stageLeft, stageTop, and stageBottom functions

## the stageTop

function

*Syntax*    the stageTop

*Description*    This function—along with stageBottom, stageLeft, and stageRight—indicates where the Stage is positioned on the desktop. It returns the top vertical coordinate of the Stage, relative to the upper-left corner of the main screen's desktop. If the Stage is in the upper-left corner of the main screen, this coordinate is zero.

*Example*    if the stageTop < 0 then upperMonitorProcedure

*Note*    Sprite coordinates are expressed relative to the upper-left corner of the Stage. See "Working with Coordinates" in Chapter 9.

*See also*    stageLeft, stageRight, and stageBottom functions

## startMovie

See on startMovie movie handler.



## starts

comparison operator

**Syntax** `string1 starts string2`

**Description** This operator compares two strings. When *string1* starts with *string2*, the condition is TRUE (1); when *string1* does not start with *string2*, the condition is FALSE (0).

This is a comparison operator with a precedence level of 1.

**Examples** `put "MacroMind" starts "macro"`  
`-- 1`

◆ `put "MacroMind" starts "Mind"`  
`-- 0`

◆ `if field "Name" starts "Sir " then doKnight`

**Note** The string comparison is not sensitive to case or diacritical marks; “a” and “Å” are considered the same.

**See also** `contains` comparison operator

## startTimer

command

**Syntax** `startTimer`

**Description** This command sets the `timer` property to zero. It also resets all the accumulating timers for the `lastClick`, `lastEvent`, `lastKey`, and `lastRoll` functions to zero.

**Example** `when keyDown then startTimer`

**See also** `timer` property; `lastClick`, `lastEvent`, `lastKey`, and `lastRoll` functions

## stepMovie

See on `stepMovie` movie handler.

## the stillDown

function

*Syntax*    the stillDown

*Description*    This function returns TRUE if the mouse button is being pressed.

*Example*    if the stillDown then dragProcedure

*See also*    mouseDown function

## stop

See sound stop command.

## stopMovie

See on stopMovie movie handler.

## the stretch of sprite

sprite property

*Syntax*    the stretch of sprite *whichSprite*

*Description*    This sprite property determines whether the bitmap sprite specified by the integer expression *whichSprite* can be stretched by using the `spriteBox` command or the `width` and `height` properties. If it is TRUE, the bitmap sprite can be stretched.

The `stretch` sprite property can be tested and set, and the default value is FALSE. When FALSE, the bitmapped sprite stays at its default or normal size at all times.

The `stretch` property applies only to bitmap castmembers, not to shape, text, or button castmembers. Shapes can be stretched at any time by setting their height and width properties, regardless of the setting of their `stretch` property. Text and button castmembers cannot be stretched in any case.



*Examples* if the stretch of sprite 3 = TRUE then —  
set the width of sprite 3 to 10

- ◆ set the stretch of sprite (i + 1) to TRUE

*See also* spriteBox command; width and height sprite properties

## string

function

*Syntax* string(*expression*)

*Description* This function converts an integer, floating point, or symbol expression to a string.

*Examples* put string(2 + 2)  
-- "4"

- ◆ put string(2 \* 3.14)  
-- "6.28"

- ◆ put string(#red)  
-- "red"

- ◆ set the name of cast i to string(i)

*See also* value function

## stringP

function

*Syntax* stringP(*expression*)

*Description* This function returns TRUE (1) if *expression* evaluates to a string, and FALSE (0) if it doesn't.

*Examples* put stringP("3")  
-- 1

- ◆ put stringP(3)  
-- 0

- ◆ put stringP(3.0)  
-- 0

- ◆ if stringP(myObject(mGet, i))  $\neg$   
then do myObject(mGet, i)

*Note* The “P” in stringP stands for “predicate”.

*See also* integerP, floatP, objectP, and symbolP functions

## the switchColorDepth

property

*Syntax* the switchColorDepth

*Description* This property determines whether Director automatically switches the color depth, if necessary, when loading a movie. If switchColorDepth is TRUE, Director switches the monitor(s) that the Stage occupies to the color depth of the movie that is being loaded.

When this property is set to TRUE, nothing happens until a new movie is loaded.

The switchColorDepth property can be tested and set. The default value depends on the user’s preference settings.

*Examples* put the switchColorDepth into switcher

- ◆ if the colorDepth = 8 then  $\neg$   
set the switchColorDepth to TRUE

*See also* colorDepth property; colorQD function

## symbolP

function

*Syntax* symbolP(*expression*)

*Description* This function returns TRUE (1) if *expression* evaluates to a symbol and returns FALSE (0) if it doesn’t.

*Examples* put symbolP(#Up)  
-- 1

- ◆ put symbolP(3)  
-- 0
- ◆ put symbolP(3.0)  
-- 0



- ◆ `put symbolP("symbol")`  
`-- 0`

*Note* The “P” in `symbolP` stands for “predicate”.

*See also* `integerP`, `floatP`, `stringP`, and `objectP` functions;  
# `symbol operator`

## T

### TAB

character constant

*Syntax* `TAB`

*Description* This character constant represents the Tab key, marked “tab” on the Macintosh keyboard.

*Example* `when keyDown then if the key = TAB then doNextField`

### the text of cast

cast property

*Syntax* `the text of cast whichCastmember`

*Description* This cast property determines the string that is the text contained in the specified text castmember. The text of the castmember can be set and tested using this property.

This property requires that the text castmember already contain text, if only a space. It will not add text to a castmember that contains no text.

The `text` cast property can be tested and set.

*Examples* `if the text of cast A35 = EMPTY then ↵`  
`set the text of cast A35 to "Thank You."`

- ◆ `set the text of cast A23 = "Hello" && yourName & "!"`

*See also* `editableText` command; `selStart` and `selEnd` text properties

## the `textAlign` of field

text property

**Syntax**    `the textAlign of field whichField`

**Description**    This text property determines the alignment used to display text within the specified text castmember.

The value of the property is a string consisting of one of the following: "left", "center", or "right". The parameter *whichField* can evaluate to either a cast name or a cast number.

The `textAlign` text property can be tested and set.

**Examples**    put the `textAlign` of field "Poem" into alignment

- ◆ repeat with i = 1 to 3
  - set the `textAlign` of field "Rove" to word i of "left center right"
- end repeat

**Note**    This property requires that the text castmember already contain text, if only a space. It will not affect a castmember that contains no text.

**See also**    `text cast property`; `textFont`, `textHeight`, `textSize`, and `textStyle` text properties

## the `textFont` of field

text property

**Syntax**    `the textFont of field whichField`

**Description**    This text property determines the typeface of the font that is used to display the specified text castmember. The parameter *whichField* can evaluate to either a cast name or a cast number.

The `textFont` text property can be tested and set.

**Examples**    put the `textFont` of field "Poem" into oldFont

- ◆ set the `textFont` of field (i + 1) to 1

**Note**    This property requires that the text castmember already contain text, if only a space. It will not affect a castmember that contains no text.

**See also**    `text cast property`; `textAlign`, `textHeight`, `textSize`, and `textStyle` text properties



## the `textHeight` of field

text property

- Syntax** `the textHeight of field whichField`
- Description** This text property determines the line spacing used to display the specified text castmember. The parameter *whichField* can evaluate to either a cast name or a cast number.
- The `textHeight` text property can be tested and set.
- Examples** put the `textHeight` of field "Poem" into `oldHeight`
- ◆ set the `textHeight` of field (i + 1) to 16
- Note** This property requires that the text castmember already contain text, if only a space. It will not affect a castmember that contains no text.
- See also** `text cast property`; `textAlign`, `textFont`, `textSize`, and `textStyle` text properties

## the `textSize` of field

text property

- Syntax** `the textSize of field whichField`
- Description** This text property determines the size of the font used to display the specified text castmember. The parameter *whichField* can evaluate to either a cast name or a cast number.
- The `textSize` text property can be tested and set.
- Examples** put the `textSize` of field "Poem" into `oldSize`
- ◆ set the `textSize` of field (i + 1) to 12
- Note** This property requires that the text castmember already contain text, if only a space. It will not affect a castmember that contains no text.
- See also** `text cast property`; `textAlign`, `textFont`, `textHeight`, and `textStyle` text properties



## the textStyle of field

text property

*Syntax*    the textStyle of field *whichField*

*Description*    This text property determines the styles applied to the font used to display the specified text castmember.

The value of the property is a string of styles delimited by commas. Lingo uses a font that is a combination of the styles in the string. The available styles are: plain, bold, italic, underline, shadow, outline, condense, and extend. In addition, you can use the word `normal` to remove all of the styles currently applied. The parameter *whichField* can evaluate to either a cast name or a cast number.

The `textStyle` text property can be tested and set.

*Examples*    put the textStyle of field "Poem" into oldStyle

- ◆ set the textStyle of field "Poem" to "bold, italic"
- ◆ set the textStyle of field (i + 1) to "normal"

*Note*    This property requires that the text castmember already contain text, if only a space. It will not affect a castmember that contains no text.

*See also*    text cast property; textAlign, textFont, textHeight, and textSize text properties

## the

*Syntax*    the *property*

*Description*    All Lingo properties and many sprite properties/functions require the keyword `the` to precede the property. This distinguishes the property from a variable or object name.

Properties have "super-global" scope, which means they are available within handlers and methods even without a global declaration. Like global variables, Lingo system properties are available between different movies in the same presentation (unless changed by system events.) Naturally sprite properties would change when a new movie is loaded.



## then

See `if...then keywords`; `when keyDown then`, `when mouseDown then`, `when mouseUp then`, and `when timeOut then commands`

## the ticks

function

*Syntax*    `the ticks`

*Description*    This function returns the current time in ticks (1/60ths of a second). The counting of ticks begins from the time the computer is started.

*Example*    `put the ticks/60/60 into minutesOn`

*See also*    `time` and `timer functions`

## the time

function

*Syntax*    `the time`  
             `the short time`  
             `the long time`  
             `the abbreviated time`  
             `the abbrev time`  
             `the abbr time`

*Description*    This function returns the current time in the system clock as a string in one of three formats: `short`, `long`, or `abbreviated`. If no format is specified, the default is `short`. The `abbreviated` format can also be referred to as `abbrev` and `abbr`. In the United States, the `short` and `abbreviated` formats are the same.

*Examples*    `put the short time`  
              `-- "1:30 PM"`

- ◆ `put the long time`  
   `-- "1:30:24 PM"`
- ◆ `put the abbreviated time`  
   `-- "1:30 PM"`
- ◆ `put the short time into field "Time Now"`

*Note* The three time formats vary, depending on the country for which your System file was designed. The examples above are for the United States.

*See also* date function

## timeOut

---

See when timeOut then command.

## the timeoutKeyDown

---

property

*Syntax* the timeoutKeyDown

*Description* When this property is TRUE, keyDown events set the timeoutLapsed property to zero.

The timeoutKeyDown property can be tested and set, and the default value is TRUE.

*Examples* put the timeoutKeyDown into timing

◆ set the timeoutKeyDown to FALSE

*See also* when timeOut then command; keyDownScript property

## the timeoutLapsed

---

property

*Syntax* the timeoutLapsed

*Description* This property indicates the number of ticks elapsed since the last timeout. A timeout event occurs when the timeoutLapsed property reaches the time specified by the timeoutLength property.

The timeoutLapsed property can be tested, but it cannot be set directly in Lingo.

*Example* put the timeoutLapsed / 60 into field "Countdown"

*See also* when timeOut then command



## the timeoutLength

property

*Syntax*    `the timeoutLength`

*Description*    This property determines the number of ticks before a timeout event occurs. A timeout occurs when the `timeoutLapsed` property reaches the time specified by the `timeoutLength` property.

The `timeoutLength` property can be tested and set, and the default value is 10,800 ticks (3 minutes).

*Examples*    `put the timeoutLength into oldTimeout`

◆ `set the timeoutLength to 10 * 60`

*See also*    when `timeOut` command

## the timeoutMouse

property

*Syntax*    `the timeoutMouse`

*Description*    This property determines if `mouseDown` events reset the `timeoutLapsed` property to zero. When this property is `TRUE`, `mouseDown` events reset the `timeoutLapsed` property.

The `timeoutMouse` property can be tested and set, and the default value is `TRUE`.

*Examples*    `put the timeoutMouse into timing`

◆ `set the timeoutMouse to FALSE`

*See also*    when `timeOut` then command; `mouseDownScript` and `mouseUpScript` properties

## the timeoutPlay

property

*Syntax*    `the timeoutPlay`

*Description*    This property determines whether the `timeoutLapsed` property is reset to zero when a movie is played. When `timeoutPlay` is `TRUE`, the playing of a movie resets the `timeoutLapsed` property to zero. This allows timeouts to occur only when the animation is paused.

The `timeoutPlay` property can be tested and set, and the default value is `FALSE`.

*Examples*    `if the timeoutPlay then beep`

◆    `set the timeoutPlay to TRUE`

*See also*    `when timeOut` command

## the timeoutScript

property

*Syntax*    `the timeoutScript`

*Description*    This property determines the string that is executed as a Lingo statement when a `timeOut` condition is reached.

Setting the `timeOutScript` property is equivalent to executing a `when timeOut then` command; both establish what Lingo will do when a `timeOut` event occurs. For example, the statements

```
set the timeOutScript to
to "go to frame 1"
```

and

```
when timeOut then
go to frame 1
```

are equivalent.

When the event script you've assigned is no longer appropriate, turn it off with either of the following statements:

```
set the timeOutScript to EMPTY
when timeOut then nothing
```

The `timeOutScript` property can be tested and set, and the default value is `EMPTY`.

*Examples*    `put the timeoutScript into oldTimes`

◆    `set the timeoutScript to "timeoutProcedure"`

*See also*    `when timeOut` command



## the timer

property

*Syntax* the timer

*Description* The timer property is a free running timer that counts time in ticks (60ths of a second). It has nothing to do with when timeOut. It is only for convenience in timing certain events. The startTimer command zeroes the value of the timer property.

*Example* put the timer into startTicks

*See also* lastClick, lastEvent, lastKey, and lastRoll functions; startTimer command

## to

The word to occurs in a number of Lingo constructs.

See set...to and repeat with commands; char...of, word...of, item...of, and line...of chunk expression keywords.

## the top of sprite

sprite property

*Syntax* the top of sprite whichSprite

*Description* This sprite property returns the top vertical coordinate of the bounding rectangle of the sprite specified by the integer expression whichSprite.

The top sprite property can be tested, but not set directly. The top vertical coordinate of a sprite can be set with the spriteBox command.

*Examples* if the top of sprite 3 < 0 then offTopEdge

◆ put the top of sprite (i + 1) into highest

*Note* Sprite coordinates are expressed relative to the upper-left corner of the Stage. See “Working with Coordinates” in Chapter 9.

*See also* bottom, left, right, height, width, locH, and locV sprite properties; spriteBox command



## TRUE

logical constant

*Syntax* TRUE

*Description* This logical constant represents the value of a logically true expression, such as `2 < 3`. It has a numerical value of 1.

*Example* set the `soundEnabled` to TRUE

*See also* FALSE logical constant

## the type of sprite

sprite property

*Syntax* the type of sprite *whichSprite*

*Description* This sprite property determines the type of sprite *whichSprite*. The type can be a bitmap, a shape, a text field, or a button. This command is useful with puppet sprites, for example, to change a shape sprite into a bitmap prior to replacing it with a bitmapped castmember, or to replace one button sprite with another type, or to make it invisible on the stage.

The sprite types are given below:

- 0 inactive sprite (turns the sprite off)
- 1 bitmap
- 7 text

Within Director 2.0 files, the following sprite types are also available:

- 2 rectangle
- 3 rounded rectangle
- 4 oval
- 5 line topleft to bottomright
- 6 line bottomleft to topright
- 8 button
- 9 check box
- 10 radio button

The `type` sprite property can be tested and set.

*Examples* put the type of sprite 4 into `spriteType`

- ◆ set the type of sprite 4 to 7
- ◆ set the type of sprite 1 to 0



*Notes* Before setting a sprite to type 1 [bitmap], it is necessary to set the `stretch` property of the sprite to `FALSE`. This will prevent it from stretching to the size of the previous sprite.

- ◆ If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

*See also* `stretch` property

## U

### updateStage

command

*Syntax* `updateStage`

*Description* This command redraws the Stage immediately. This is typically used with the `puppetSprite` command to cause animation to occur within a handler or macro when the playback head is not moving. Normally the Stage is updated only between frames, but the `updateStage` command can be called at any time from a handler, macro, or factory method to force the Stage to update.

Do not use `updateStage` with the `perFrameHook` property. Otherwise, unexpected results could occur.

*Example*

```
on moveRight whichSprite, howFar
 puppetSprite whichSprite, TRUE
 set the locH of sprite whichSprite to
 to the locH of sprite whichSprite + howFar
 updateStage
 puppetSprite whichSprite, FALSE
end moveRight
```

# V

## value

function

*Syntax*    `value (string)`

*Description*    This function returns the numerical value of a string. Useful when making use of a numerical string that the user has typed into a text castmember or data from XObjects which return numerical strings.

*Example*    `put value("3")  
-- 3`

◆ `put value("the sqrt of" && "2.0")  
-- 1.4142`

◆ `put value("penny")  
-- <NoValue>`

◆ `set the timeoutLength to 60*value(the text of cast-  
"Timeout")`

*See also*    `string function`

## version

system variable

*Syntax*    `version`

*Description*    This system variable contains the version string for MacroMind Director. (The same string appears in the Finder's Get Info box.) If it is `EMPTY`, then you are using Director 2.0.

*Example*    `put version  
-- "3.0"`

◆ `if version = EMPTY then alert "This movie needs a  
more recent version of Director."`



## the volume of sound

sound property

**Syntax** the volume of sound *whichChannel*

**Description** This sound property determines the volume of the sound channel specified by the integer expression *whichChannel*. Sound channels are numbered 1, 2, 3, ... with 1 and 2 being the two that appear in the Score.

The value of the `volume` sound property ranges from 0 (silent) to 255 (maximum volume).

**Example** put the volume of sound 1 into vol

◆ set the volume of sound i to 170

**See also** `soundEnabled` and `soundLevel` properties

## W

### when

See `when keyDown then`, `when mouseDown then`, `when mouseUp then`, and `when timeOut then` commands.

### when keyDown then

command

**Syntax** `when keyDown then statement`

**Description** This command establishes a Lingo statement to be executed each time a key is pressed (at each `keyDown` event ).

The *statement* to be executed must be only one line long. It can be a single command, a one-line test, or —if you need to execute multiple statements when a `keyDown` event occurs—a handler call.

The `keyDown` action remains in effect until you turn it off with `when keyDown then nothing`.



For more information, see *Chapter 3: Using Lingo*.

*Examples* when keyDown then beep

- ◆ when keyDown then —  
if the key = RETURN then go to frame "ending"
- ◆ when keyDown then doKeyDown
- ◆ when keyDown then nothing

*Notes* The keyDown action is automatically turned off when you load a new movie.

*See also* dontPassEvent command, keyDownScript property, keyCode, key functions

## when mouseDown then

command

*Syntax* when mouseDown then *statement*

*Description* This command establishes a Lingo statement to be executed each time the mouse button is pressed (at each mouseDown event).

The *statement* to be executed must be only one line long. It can be a single command, a one-line test, or —if you need to execute multiple statements when a mouseDown event occurs—a handler call.

The mouseDown action remains in effect until you turn it off with when mouseDown then nothing.

For more information, see *Chapter 3: Using Lingo*.

*Examples* when mouseDown then beep

- ◆ when mouseDown then —  
if the optionDown then go to frame "ending"
- ◆ when mouseDown then doMouseDown
- ◆ when mouseDown then nothing

*Notes* The mouseDown action is automatically turned off when you load a new movie.

*See also* dontPassEvent command, mouseDownScript property



## when mouseUp then

command

**Syntax** `when mouseUp then statement`

**Description** This command establishes a Lingo statement to be executed each time the mouse button is released (at each `mouseUp` event).

The *statement* to be executed must be only one line long. It can be a single command, a one-line test, or—if you need to execute multiple statements when a `mouseUp` event occurs—a handler call.

The `mouseUp` action remains in effect until you turn it off with `when mouseUp then nothing`.

For more information, see *Chapter 3: Using Lingo*.

**Examples** `when mouseUp then beep`

- ◆ `when mouseUp then ↵  
    if the optionDown then go to frame "ending"`
- ◆ `when mouseUp then doMouseUp`
- ◆ `when mouseUp then nothing`

**Notes** The `mouseUp` action is automatically turned off when you load a new movie.

**See also** `dontPassEvent` command, `mouseUpScript` property

## when timeOut then

command

**Syntax** `when timeOut then statement`

**Description** This command establishes a Lingo statement to be executed each time when the user doesn't click the mouse, type a key, or play a movie for a specified amount of time (at each `timeOut`). For example, if the user doesn't interact with your interactive application, you may want to activate a script that provides some on-screen help.

The `when timeOut then` command instructs MacroMind Director what to do when a timeout occurs. A timeout occurs when the user has done nothing for a specified time period. The length of the timeout is determined by the `timeoutLength` property:

`set the timeoutLength to numberOfTicks`



The system keeps track of how long the user has been inactive in the `timeoutLapsed` property. A timeout occurs when the `timeoutLapsed` property reaches the time specified by the `timeoutLength` property. Whenever the user interacts with the system (for example, by pressing the mouse button), the `timeoutLapsed` property is reset to zero. Therefore, the value of the `timeoutLapsed` property usually never reaches the time specified in the `timeoutLength` property while the user is doing things. You can select which actions (such as `mouseDown`, `keyDown`, or playing a movie) reset the `timeoutLapsed` to zero with the following commands:

```
set the timeoutKeydown to TRUE
```

```
set the timeoutMouse to TRUE
```

```
set the timeoutPlay to TRUE
```

Setting these properties to `TRUE` means that clicking the mouse, typing a key, or playing a movie resets the `timeoutLapsed` property to zero. Setting them to `FALSE` means that these events do not reset the `timeoutLapsed` property to zero. The defaults are as follows:

```
timeoutKeydown - TRUE
```

```
timeoutMouse - TRUE
```

```
timeoutPlay - FALSE
```

You can also set the `timeoutLapsed` property to zero directly via Lingo with this script:

```
set the timeoutLapsed to 0
```

For more information, see Chapter 3.

*Examples* when timeOut then go to frame "help"

◆ when timeOut then nothing

*Notes* A `timeOut` action remains in effect even if you go to another movie, so make sure the action is valid for any movies it may be executed in.

*See also* `dontPassEvent` command, `timeoutKeydown`, `timeoutLapsed`, `timeoutLength`, `timeoutMouse`, `timeoutPlay` properties



## while

---

See `repeat while` keyword.

## the width of sprite

---

sprite property

**Syntax** the width of sprite *whichSprite*

**Description** This sprite property determines the horizontal size in pixels of the sprite specified by the integer expression *whichSprite*. The `width` applies only to bitmap and shape castmembers. It does not affect text or button castmembers.

The `width` sprite property can be tested and set.

**Examples** set the width of sprite 10 to 26

- ◆ put the width of sprite (i + 1) into w

**Notes** Setting this property does not have any effect for bitmap sprites unless the sprite's `stretch` property is set to `TRUE`.

- ◆ If you set this property within a script while the playback head is not moving, be sure to use the command `updateStage` to redraw the Stage. If you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

**See also** `height` and `stretch` sprite properties; `spriteBox` command

## with

---

See `repeat with` keyword.

## within

---

See `sprite...within` comparison operator.

## word...of

chunk expression keyword

*Syntax* word *whichWord* of *chunkExpression*

- ◆ word *firstWord* to *lastWord* of *chunkExpression*

*Description* This keyword is used to specify a word or a range of words in a chunk expression. A word chunk is any sequence of characters delimited by spaces. (Any non-visible character is considered a space. That is, tabs and returns are also considered spaces.)

The expressions *whichWord*, *firstWord*, and *lastWord* must evaluate to integers which specify a word in the chunk.

Chunk expressions are used to refer to any character, word, item, or line in any source of text. Sources of text include fields (text castmembers) and variables that hold strings.

*Examples* put word 2 of "fox dog cat"  
-- "dog"

- ◆ put word 2 to 3 of "fox dog cat"  
-- "dog cat"
- ◆ put word 5 of "fox dog cat"  
-- ""
- ◆ put char 1 of word 2 of line 3 of field "Zombie" →  
into c
- ◆ put " and" after word 3 of myString

*See also* char...of, line...of, and item...of chunk expression keywords;  
the number of words in chunk function

## words

See the number of words in chunk function.



# X

## xFactoryList

function

**Syntax** xfactoryList (*whichLibrary*)

**Description** This function returns a string list of all the currently available XObject factories in the XLibrary file specified by the string expression *whichLibrary*. The XLibrary must have been previously opened with the openXlib command. If *whichLibrary* evaluates to EMPTY, this function returns a list of all XObject factories in all open XLibraries.

The XObject factories appear one per line in the returned string list. Each line ends with a RETURN character.

**Examples** put xfactoryList("AppleCD XObj")  
-- "AppleCD  
"

◆ put line 1 of xfactoryList(EMPTY)  
-- "FileIO"

# Z

## zoomBox

command

**Syntax** zoomBox startSprite, endSprite [, delayTicks]

**Description** This command creates a zooming effect, like the expanding windows in the Finder. The zoom effect starts at the bounding rectangle of startSprite and finishes at the bounding rectangle of endSprite. zoomBox uses the following logic when executing:

1. Looks for endSprite in the current frame, otherwise,
2. Looks for endSprite in the next frame.

Note, however, that the `zoomBox` command does not work for an *endSprite* in the same channel as *startSprite*.

*delayTicks* is the delay in ticks between each movement of the zoom rectangles. If *delayTicks* is not specified, the delay is 1.

*Examples*    `zoomBox 7, 3`

◆ `zoomBox mySprite, mySprite + 1, 15`

## Symbols

( )

grouping operator

*Syntax*    `(expression)`

*Description*    This operator performs a grouping operation on an expression. It is used to control the order of execution of the operators in an expression, and override the automatic precedence order. It causes the expression contained within the parentheses to be evaluated first. When parentheses are nested, the contents of inner ones are evaluated before the contents of outer ones.

This is a grouping operator with a precedence level of 5.

*Examples*    `put (2 + 3) * (4 + 5)`  
              `-- 45`

◆ `put 2 + (3 * (4 + 5))`  
   `-- 29`

◆ `put 2 + 3 * 4 + 5`  
   `-- 19`



---

**-** arithmetic operator

---

**Syntax**    `-expression`

**Description**    This operator performs an arithmetic negation on a numerical expression. This operation reverses the sign of the value of the expression.

This is an arithmetic operator with a precedence level of 5.

**Example**    `put -(2 + 3)`  
              `-- -5`

---

**-- (double hyphens)** comment delimiter

---

**Syntax**    `-- [comment]`

**Description**    This comment symbol indicates the beginning of a script comment. On any line, what is between the comment symbol (double hyphens) and the end-of-line RETURN character, will be interpreted as a comment instead of a Lingo statement.

**Example**    `on resetColors`  
              `-- This handler resets the sprite's colors.`  
              `set the foreColor of sprite 1 to 35 -- bright red`  
              `set the backColor of sprite 1 to 36 -- light blue`  
  
              `end resetColors`

---

**⏏ (Option-Return character)** special character

---

**Syntax**    *first part of a statement on this line ⏏*  
              *second part of same statement on next line ⏏*  
              *third part of same statement*

**Description**    The special character ⏏ is created in Lingo by pressing the Option and Return keys simultaneously. When using this as the last character in a line, the statement continues on the next line. This can be done on several successive lines.

*Example*    set the castNum of sprite mySprite to  
              to the number of cast  
              "This is a long cast name."

\* arithmetic operator

---

*Syntax*    *expression1* \* *expression2*

*Description*    This operator performs an arithmetic multiplication on two numerical expressions. If both expressions evaluate to integers, the product is an integer. If either or both expressions evaluate to floating point numbers, the product is a floating point number.

              This is an arithmetic operator with a precedence level of 4.

*Examples*    put 2 \* 3  
              -- 6

              ◆ put 2.0 \* 3.1416  
              -- 6.2832

              ◆ put width \* height into area

/ arithmetic operator

---

*Syntax*    *expression1* / *expression2*

*Description*    This operator performs an arithmetic division on two numerical expressions, dividing *expression1* by *expression2*. If both expressions evaluate to integers, the quotient is an integer. If either or both expressions evaluate to floating point numbers, the quotient is a floating point number.

              This is an arithmetic operator with a precedence level of 4.

*Examples*    put 22 / 7  
              -- 3

              ◆ put 22.0 / 7.0  
              -- 3.1429

              ◆ put distance / time into speed



+

arithmetic operator

*Syntax*    *expression1* + *expression2*

*Description*    This operator performs an arithmetic sum on two numerical expressions. If both expressions evaluate to integers, the sum is an integer. If either or both expressions evaluate to floating point numbers, the sum is a floating point number.

This is an arithmetic operator with a precedence level of 4.

*Examples*    put 2 + 3  
                  -- 5

◆ put 2.5 + 3.25  
   -- 5.75

◆ put apples + oranges into fruitCount

-

arithmetic operator

*Syntax*    *expression1* - *expression2*

*Description*    This operator performs an arithmetic subtraction on two numerical expressions, subtracting *expression2* from *expression1*. If both expressions evaluate to integers, the difference is an integer. If either or both expressions evaluate to floating point numbers, the difference is a floating point number.

This is an arithmetic operator with a precedence level of 3.

*Examples*    put 13 - 8  
                  -- 5

◆ put 2.5 - 3.25  
   -- -0.75

◆ put finishTime - startTime into duration

## &

text operator

*Syntax*    *expression1* & *expression2*

*Description*    This operator performs a string concatenation of two expressions. If either *expression1* or *expression2* evaluates to a number, it is first converted to a string. The resulting expression is a string.

This is a text operator with a precedence level of 2.

*Examples*    `put "abra" & "cadabra"`  
              `-- "abracadabra"`

- ◆ `put "This is line 1." & RETURN & "And this is line 2."`  
      `-- "This is line 1."`  
      `And this is line 2."`
- ◆ `put "$" & price into field "Price"`

## &&

text operator

*Syntax*    *expression1* && *expression2*

*Description*    This operator performs a string concatenation of two expressions, inserting a space character between the original string expressions. If either *expression1* or *expression2* evaluates to a number, it is first converted to a string. The resulting expression is a string.

This is a text operator with a precedence level of 2.

*Examples*    `put "abra" && "cadabra"`  
              `-- "abra cadabra"`

- ◆ `put "Today is" && the long date`  
      `-- "Today is Tuesday, May 14, 1991"`
- ◆ `put "Age" && age into field "Age"`



<

comparison operator

*Syntax*    *expression1* < *expression2*

*Description*    This operator compares two expressions. When *expression1* is less than *expression2*, the condition is **TRUE**. When *expression1* is greater than or equal to *expression2*, the condition is **FALSE**.

This operator can compare strings as well as integers and floating point numbers.

This is a comparison operator with a precedence level of 1.

<=

comparison operator

*Syntax*    *expression1* <= *expression2*

*Description*    This operator compares two expressions. When *expression1* is less than or equal to *expression2*, the condition is **TRUE**. When *expression1* is greater than *expression2*, the condition is **FALSE**.

This operator can compare strings as well as integers and floating point numbers.

This is a comparison operator with a precedence level of 1.

>

comparison operator

*Syntax*    *expression1* > *expression2*

*Description*    This operator compares two expressions. When *expression1* is greater than *expression2*, the condition is **TRUE**. When *expression1* is less than or equal to *expression2*, the condition is **FALSE**.

This operator can compare strings as well as integers and floating point numbers.

This is a comparison operator with a precedence level of 1.

**>=**

comparison operator

- Syntax** *expression1* >= *expression2*
- Description** This operator compares two expressions. When *expression1* is greater than or equal to *expression2*, the condition is TRUE. When *expression1* is less than *expression2*, the condition is FALSE.
- This operator can compare strings as well as integers and floating point numbers.
- This is a comparison operator with a precedence level of 1.

**<>**

comparison operator

- Syntax** *expression1* <> *expression2*
- Description** This operator compares two expressions. When *expression1* is not equal to *expression2*, the condition is TRUE. When *expression1* is equal to *expression2*, the condition is FALSE.
- This operator can compare strings as well as integers and floating point numbers.
- This is a comparison operator with a precedence level of 1. This operator also works with symbols and objects.

**=**

comparison operator

- Syntax** *expression1* = *expression2*
- Description** This operator compares two expressions or strings. When *expression1* is equal to *expression2*, the condition is TRUE. When *expression1* is not equal to *expression2*, the condition is FALSE.
- This operator can compare strings as well as integers and floating point numbers.
- This is a comparison operator with a precedence level of 1. This operator also works with symbols and objects.



# #

symbol definition operator

*Syntax* #*symbolName*

*Description* This symbol operator defines a symbol. In addition to integers, floating point numbers, strings, and objects, Lingo also has a symbol data type. A *symbolName* begins with an alphabetical character and may be followed by any number of alphabetical or numerical characters.

The valid operations on symbols are:

- ◆ assignment to a variable
- ◆ comparison
- ◆ being passed as a parameter to a handler or method
- ◆ being returned as a value from a handler or method

Symbols take up much less space than strings and can be manipulated faster than strings can. Essentially, symbols have the speed and memory advantages of integers but give you the descriptive power of strings!

A symbol is a self-contained unit, which can be used to represent a condition or flag. It does not consist of individual characters in the same sense as a string. However, you can convert a symbol to a string for display purposes by using the string function.

*Examples* put #Playing into state

- ◆ repeat while state = #Paused
- ◆ put string(#Stopped) into field "Status"

*See also* symbolP function





# *Appendix A: Lingo Quick Reference*

This appendix provides you with a quick reference summary of Lingo's vocabulary and a list of keyboard shortcuts.

The vocabulary summary includes the basic syntax of each word, grouped by category and organized alphabetically into the groups:

- Castmembers and Sprites
- Code Structures: Loops, Methods, and Factories
- Constants
- Event Handlers and Messages
- Events
- External to MacroMind Director
- Keyboard
- Logical Operators and Functions
- MacroMind Player
- Menus
- Mouse and Pointer
- Operators and Math Functions
- Output
- Playing Movies
- Predefined Methods
- Puppets
- Sound
- System
- Text
- Time
- Variables

## Lingo Syntax

The following typographic conventions are used in this section:

|                 |                                                 |
|-----------------|-------------------------------------------------|
| word            | actual Lingo word                               |
| <i>word</i>     | place holder for a specific name or parameter   |
| [word]          | optional items                                  |
| [one two three] | list of alternative optional items              |
| {one two three} | list of alternative items, one item is required |

### Castmembers and Sprites

| Word           | Syntax                                                | Category         |
|----------------|-------------------------------------------------------|------------------|
| All...H88      | All                                                   | cast identifiers |
| the            | the <i>property</i>                                   | keyword          |
| cast           | the <i>property</i> of cast <i>castMember</i>         | keyword          |
| backColor      | the backColor of sprite <i>whichSprite</i>            | sprite property  |
| bottom         | the bottom of sprite <i>whichSprite</i>               | sprite property  |
| buttonStyle    | the buttonStyle                                       | property         |
| castNum        | the castNum of sprite <i>whichSprite</i>              | sprite property  |
| checkBoxAccess | the checkBoxAccess                                    | property         |
| checkBoxType   | the checkBoxType                                      | property         |
| constrainH     | constrainH ( <i>whichSprite</i> , <i>integerExp</i> ) | function         |
| constraint     | the constraint of sprite <i>whichSprite</i>           | sprite property  |
| constrainV     | constrainV ( <i>whichSprite</i> , <i>integerExp</i> ) | function         |
| field          | field <i>whichField</i>                               | function         |
| foreColor      | the foreColor of sprite <i>whichSprite</i>            | sprite property  |
| height         | the height of sprite <i>whichSprite</i>               | sprite property  |
| hilite         | the hilite of cast <i>whichCastmember</i>             | button property  |
| immediate      | the immediate of sprite <i>whichSprite</i>            | sprite property  |
| ink            | the ink of sprite <i>whichSprite</i>                  | sprite property  |
| left           | the left of sprite <i>whichSprite</i>                 | sprite property  |
| lineSize       | the lineSize of sprite <i>whichSprite</i>             | sprite property  |
| locH           | the locH of sprite <i>whichSprite</i>                 | sprite property  |
| locV           | the locV of sprite <i>whichSprite</i>                 | sprite property  |
| moveableSprite | moveableSprite                                        | command          |
| name           | the name of cast <i>whichCastmember</i>               | cast property    |
| number         | the number of cast <i>whichCastmember</i>             | cast property    |
| number         | the number of castmembers                             | property         |
| picture        | the picture of cast <i>whichCastmember</i>            | cast property    |
| right          | the right of sprite <i>whichSprite</i>                | sprite property  |



| Word        | Syntax                                                 | Category        |
|-------------|--------------------------------------------------------|-----------------|
| spriteBox   | spriteBox <i>whichSprite, left, top, right, bottom</i> | command         |
| the stretch | the stretch of sprite <i>whichSprite</i>               | sprite property |
| text        | the text of cast <i>whichCastmember</i>                | text property   |
| top         | the top of sprite <i>whichSprite</i>                   | sprite property |
| type        | the type of sprite <i>whichSprite</i>                  | sprite property |
| updateStage | updateStage                                            | command         |
| width       | the width of sprite <i>whichSprite</i>                 | sprite property |
| zoomBox     | zoomBox <i>startSprite, endSprite [, delayTicks]</i>   | command         |

## Code Structures: Loops, Methods, and Factories

| Word         | Syntax                                                                                                                                           | Category          |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|
| --           | -- [comment ]                                                                                                                                    | comment delimiter |
| ↵            | <i>first part of a statement on this line</i> ↵<br><i>continuation of the statement on the next line</i>                                         | special character |
| exit         | exit                                                                                                                                             | keyword           |
| exit repeat  | exit repeat                                                                                                                                      | keyword           |
| factory      | factory <i>factoryname</i><br><i>methods</i>                                                                                                     | keyword           |
| factory      | factory ( <i>factoryname</i> )                                                                                                                   | function          |
| global       | global <i>variable1</i> [, <i>variable2</i> ] [, <i>variable3</i> ]...                                                                           | keyword           |
| if           | if <i>test</i> then <i>true-part</i> else <i>false-part</i>                                                                                      | keyword           |
| if           | if <i>logicalExpression</i> then<br><i>then-statements</i> ...<br>else<br><i>else-statement</i> ...<br>end if                                    | keyword           |
| instance     | instance <i>variable1</i> [, <i>variable2</i> ]...                                                                                               | keyword           |
| me           | me ( <i>messageName</i> [ <i>arg1</i> ] [, <i>arg2</i> ]...)                                                                                     | keyword           |
| method       | method <i>methodName</i> [ <i>arg1</i> ] [, <i>arg2</i> ]...<br><i>commands</i> ...                                                              | keyword           |
| nothing      | nothing                                                                                                                                          | command           |
| on           | on <i>handlerName</i> [ <i>argument1</i> ] [, <i>argument2</i> ] [, <i>argument3</i> ] ...<br>[ <i>statements</i> ...]<br>end <i>handlerName</i> | keyword           |
| repeat while | repeat while <i>testExpression</i><br>[ <i>statements</i> ...]<br>end repeat                                                                     | keyword           |
| repeat       | repeat with <i>counter</i> = <i>start</i> to <i>finish</i><br>[ <i>statements</i> ...]<br>end repeat                                             | keyword           |
| result       | the result                                                                                                                                       | function          |
| return       | return <i>expression</i>                                                                                                                         | keyword           |

## Constants

| Word        | Syntax      | Category           |
|-------------|-------------|--------------------|
| A11 ... H88 | A11 ... H88 | cast identifiers   |
| BACKSPACE   | BACKSPACE   | character constant |
| EMPTY       | EMPTY       | character constant |
| ENTER       | ENTER       | character constant |
| FALSE       | FALSE       | logical constant   |
| QUOTE       | QUOTE       | character constant |
| RETURN      | RETURN      | character constant |
| TAB         | TAB         | character constant |
| TRUE        | TRUE        | logical constant   |

## Event Handlers and Messages

| Word          | Syntax                                                                                                                                           | Category      |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| on            | on <i>handlerName</i> [ <i>argument1</i> ] [, <i>argument2</i> ] [, <i>argument3</i> ] ...<br>[ <i>statements...</i> ]<br>end <i>handlerName</i> | keyword       |
| on idle       | on idle<br>[ <i>statements...</i> ]<br>end idle                                                                                                  | movie handler |
| on mouseDown  | on mouseDown<br>[ <i>statements...</i> ]<br>end mouseDown                                                                                        | event handler |
| on mouseUp    | on mouseUp<br>[ <i>statements...</i> ]<br>end mouseUp                                                                                            | event handler |
| on startMovie | on startMovie<br>[ <i>statements...</i> ]<br>end startMovie                                                                                      | movie handler |
| on stepMovie  | on stepMovie<br>[ <i>statements...</i> ]<br>end stepMovie                                                                                        | movie handler |
| on stopMovie  | on stopMovie<br>[ <i>statements...</i> ]<br>end stopMovie                                                                                        | movie handler |



## Events

| Word          | Syntax                               | Category |
|---------------|--------------------------------------|----------|
| dontPassEvent | dontPassEvent                        | command  |
| keyDown       | when keyDown then <i>statement</i>   | command  |
| lastEvent     | the lastEvent                        | function |
| mouseDown     | when mouseDown then <i>statement</i> | command  |
| mouseUp       | when mouseUp then <i>statement</i>   | command  |
| nothing       | nothing                              | command  |
| perFrameHook  | the perFrameHook                     | property |
| timeOut       | when timeOut then <i>statement</i>   | command  |

## External to MacroMind Director

| Word           | Syntax                                                    | Category |
|----------------|-----------------------------------------------------------|----------|
| closeDA        | closeDA                                                   | command  |
| closeResFile   | closeResFile [ <i>whichFile</i> ]                         | command  |
| closeXlib      | closeXlib [ <i>whichFile</i> ]                            | command  |
| open           | open [ <i>whichDocument</i> with] <i>whichApplication</i> | command  |
| openDA         | openDA <i>DAname</i>                                      | command  |
| openResFile    | openResFile <i>whichFile</i>                              | command  |
| openXlib       | openXlib <i>whichFile</i>                                 | command  |
| pathName       | the pathName                                              | function |
| setCallBack    | setCallBack <i>XCMDname, value</i>                        | command  |
| showResFile    | showResFile [ <i>whichFile</i> ]                          | command  |
| showXlib       | showXlib [ <i>whichFile</i> ]                             | command  |
| sound playFile | sound playFile <i>whichChannel, whichFile</i>             | command  |
| xFactoryList   | xFactoryList ( <i>whichLibrary</i> )                      | function |

## Keyboard

| Word          | Syntax            | Category |
|---------------|-------------------|----------|
| commandDown   | the commandDown   | function |
| controlDown   | the controlDown   | function |
| key           | the key           | function |
| keyCode       | the keyCode       | function |
| keyDownScript | the keyDownScript | property |
| lastKey       | the lastKey       | function |
| optionDown    | the optionDown    | function |
| shiftDown     | the shiftDown     | function |
| stillDown     | the stillDown     | function |

## Logical Operators and Functions

| Word       | Syntax                                                      | Category            |
|------------|-------------------------------------------------------------|---------------------|
| and        | <i>logicalExpression1</i> and <i>logicalExpression2</i>     | logical operator    |
| contains   | <i>string Expression1</i> contains <i>stringExpression2</i> | comparison operator |
| floatP     | floatP( <i>expression</i> )                                 | function            |
| integerP   | integerP( <i>expression</i> )                               | function            |
| intersects | sprite <i>sprite1</i> intersects <i>sprite2</i>             | comparison operator |
| not        | not <i>logicalExpression</i>                                | logical operator    |
| objectP    | objectP( <i>expression</i> )                                | function            |
| or         | <i>logicalExpression1</i> or <i>logicalExpression2</i>      | logical operator    |
| soundBusy  | soundBusy ( <i>whichChannel</i> )                           | command             |
| sprite     | sprite <i>sprite1</i> intersects <i>sprite2</i>             | comparison operator |
| sprite     | sprite <i>sprite1</i> within <i>sprite2</i>                 | comparison operator |
| starts     | <i>string1</i> starts <i>string2</i>                        | comparison operator |
| stringP    | stringP( <i>expression</i> )                                | function            |
| symbolP    | symbolP( <i>expression</i> )                                | function            |
| within     | sprite <i>sprite1</i> within <i>sprite2</i>                 | comparison operator |

## MacroMind Player

| Word         | Syntax           | Category |
|--------------|------------------|----------|
| centerStage  | the centerStage  | property |
| exitLock     | the exitLock     | property |
| fixStageSize | the fixStageSize | property |
| quit         | quit             | command  |

## Menus

| Word        | Syntax                                                              | Category          |
|-------------|---------------------------------------------------------------------|-------------------|
| checkMark   | the checkMark of menuItem <i>whichItem</i> of menu <i>whichMenu</i> | menuItem property |
| enabled     | the enabled of menuItem <i>whichItem</i> of menu <i>whichMenu</i>   | menuItem property |
| installMenu | installMenu <i>castNum</i>                                          | command           |
| menu        | menu: <i>menuName</i><br><i>itemName</i> ≈ <i>script</i>            | keyword           |
| name        | the name of menu <i>whichMenu</i>                                   | menu property     |
| name        | the name of menuItem <i>whichItem</i> of menu <i>whichMenu</i>      | menuItem property |



| Word   | Syntax                                                           | Category          |
|--------|------------------------------------------------------------------|-------------------|
| number | the number of menuItems of menu <i>whichMenu</i>                 | menu property     |
| number | the number of menus                                              | menu property     |
| script | the script of menuItem <i>whichItem</i> of menu <i>whichMenu</i> | menuItem property |

## Mouse and Pointer

| Word            | Syntax                                  | Category        |
|-----------------|-----------------------------------------|-----------------|
| clickOn         | the clickOn                             | function        |
| cursor          | cursor <i>whichCursor</i>               | command         |
| cursor          | the cursor of sprite <i>whichSprite</i> | sprite property |
| doubleClick     | the doubleClick                         | function        |
| lastClick       | the lastClick                           | function        |
| lastRoll        | the lastRoll                            | function        |
| mouseCast       | the mouseCast                           | function        |
| mouseChar       | the mouseChar                           | function        |
| mouseDown       | the mouseDown                           | function        |
| mouseDownScript | the mouseDownScript                     | property        |
| mouseH          | the mouseH                              | function        |
| mouseItem       | the mouseItem                           | function        |
| mouseLine       | the mouseLine                           | function        |
| mouseUp         | the mouseUp                             | function        |
| mouseUpScript   | the mouseUpScript                       | property        |
| mouseV          | the mouseV                              | function        |
| mouseWord       | the mouseWord                           | function        |
| rollOver        | rollOver ( <i>spriteNum</i> )           | function        |
| stillDown       | the stillDown                           | function        |

## Operators and Math Functions

| Word | Syntax                                   | Category            |
|------|------------------------------------------|---------------------|
| &    | <i>expression1</i> & <i>expression2</i>  | text operator       |
| &&   | <i>expression1</i> && <i>expression2</i> | text operator       |
| ()   | ( <i>expression</i> )                    | grouping operator   |
| *    | <i>expression1</i> * <i>expression2</i>  | arithmetic operator |
| +    | <i>expression1</i> + <i>expression2</i>  | arithmetic operator |
| -    | <i>expression1</i> - <i>expression2</i>  | arithmetic operator |
| -    | - <i>expression</i>                      | arithmetic operator |

| Word    | Syntax                                           | Category            |
|---------|--------------------------------------------------|---------------------|
| /       | <i>expression1 / expression2</i>                 | arithmetic operator |
| <       | <i>expression1 &lt; expression2</i>              | comparison operator |
| <=      | <i>expression1 &lt;= expression2</i>             | comparison operator |
| <>      | <i>expression1 &lt;&gt; expression2</i>          | comparison operator |
| =       | <i>expression1 = expression2</i>                 | comparison operator |
| >       | <i>expression1 &gt; expression2</i>              | comparison operator |
| >=      | <i>expression1 &gt;= expression2</i>             | comparison operator |
| #       | <i># symbolName</i>                              | definition operator |
| abs     | <i>abs (numericalExpression)</i>                 | function            |
| integer | <i>integer (numericalExpression)</i>             | function            |
| mod     | <i>integerExpression1 mod integerExpression2</i> | arithmetic operator |
| random  | <i>random (integerExpression)</i>                | function            |
| sqrt    | <i>sqrt (numericalExpression)</i>                | function            |
| sqrt    | <i>the sqrt of numericalExpression</i>           | function            |
| value   | <i>value ("string")</i>                          | function            |

## Output

| Word         | Syntax                                            | Category |
|--------------|---------------------------------------------------|----------|
| perFrameHook | <i>the perFrameHook</i>                           | property |
| printFrom    | <i>printFrom fromFrame [, toFrame] [, reduce]</i> | command  |

## Playing Movies

| Word       | Syntax                                                | Category |
|------------|-------------------------------------------------------|----------|
| continue   | <i>continue</i>                                       | command  |
| frame      | <i>the frame</i>                                      | function |
| go         | <i>go [to] [frame] whichFrame</i>                     | command  |
| go         | <i>go [to] movie whichMovie</i>                       | command  |
| go         | <i>go [to] [frame] whichFrame of movie whichMovie</i> | command  |
| label      | <i>label (stringExpression)</i>                       | function |
| labelList  | <i>the labelList</i>                                  | function |
| marker     | <i>marker (integerExpression)</i>                     | function |
| movie      | <i>the movie</i>                                      | function |
| pathName   | <i>the pathName</i>                                   | function |
| pause      | <i>pause</i>                                          | command  |
| pauseState | <i>the pauseState</i>                                 | function |
| play       | <i>play [frame] whichFrame</i>                        | command  |



| Word             | Syntax                                                    | Category |
|------------------|-----------------------------------------------------------|----------|
| play             | play movie <i>whichMovie</i>                              | command  |
| play             | play [frame] <i>whichFrame</i> of movie <i>whichMovie</i> |          |
|                  |                                                           | command  |
| play done        | play done                                                 | command  |
| playAccel        | playAccel <i>filename</i> [, <i>options</i> ...]          | command  |
| preLoad          | preLoad [fromFrame] [toFrame]                             | command  |
| preLoadCast      | preLoadCast [fromCast] [toCast]                           | command  |
| quit             | quit                                                      | command  |
| switchColorDepth | the switchColorDepth                                      | command  |

## Predefined Methods and Special Messages

| Word                | Syntax                                                                         | Category          |
|---------------------|--------------------------------------------------------------------------------|-------------------|
| mAtFrame            | method mAtFrame frameNumber, subFrameNumber<br>.. [statements]<br>end mAtFrame | special message   |
| mDescribe           | XObject (mDescribe)                                                            | predefined method |
| mDispose            | object mDispose                                                                | predefined method |
| mGet                | object (mGet, <i>whichElement</i> )                                            | predefined method |
| mInstanceRespondsTo | XObject (mInstanceRespondsTo, "message")                                       | predefined method |
| mMessageList        | XObject (mMessageList)                                                         | predefined method |
| mName               | XObject (mName)                                                                | predefined method |
| mNew                | factory (mNew [arg1] [,arg2]...)                                               | predefined method |
| mNew                | XObject (mNew [arg1] [,arg2]...)                                               | predefined method |
| mPerform            | object (mPerform, message [arg1][,arg2])                                       | predefined method |
| mPut                | object (mPut, <i>whichElement</i> , <i>expression</i> )                        | predefined method |
| mRespondsTo         | XObjectInstance (mRespondsTo, "message")                                       | predefined method |

## Puppets

| Word          | Syntax                                                                  | Category        |
|---------------|-------------------------------------------------------------------------|-----------------|
| puppet        | the puppet of sprite <i>whichSprite</i>                                 | sprite property |
| puppetPalette | puppetPalette <i>whichPalette</i> [, <i>speed</i> ] [, <i>nframes</i> ] |                 |
|               |                                                                         | command         |
| puppetSound   | puppetSound <i>castmemberName</i>                                       | command         |
| puppetSound   | puppetSound <i>menuItemName</i> , <i>submenuItemNumber</i>              |                 |
|               |                                                                         | command         |
| puppetSound   | puppetSound <i>MIDIoption</i>                                           | command         |
| puppetSound   | puppetSound 0                                                           | command         |

| Word             | Syntax                                                                                                     | Category |
|------------------|------------------------------------------------------------------------------------------------------------|----------|
| puppetSprite     | puppetSprite <i>whichSprite, state</i>                                                                     | command  |
| puppetTempo      | puppetTempo <i>framesPerSecond</i>                                                                         | command  |
| puppetTransition | puppetTransition <i>whichTransition</i> [, <i>time</i> ] [, <i>chunkSize</i> ] —<br>[, <i>changeArea</i> ] | command  |
| xFactoryList     | xFactoryList ( <i>whichLibrary</i> )                                                                       | function |
| updateStage      | updateStage                                                                                                | command  |

## Sound

| Word         | Syntax                                              | Category |
|--------------|-----------------------------------------------------|----------|
| fadeIn       | sound fadeIn <i>whichChannel</i> [, <i>ticks</i> ]  | command  |
| fadeOut      | sound fadeOut <i>whichChannel</i> [, <i>ticks</i> ] | command  |
| playFile     | sound playFile <i>whichChannel, whichFile</i>       | command  |
| puppetSound  | puppetSound <i>castmemberName</i>                   | command  |
| puppetSound  | puppetSound <i>menuItemName, submenuItemNumber</i>  | command  |
| puppetSound  | puppetSound <i>MIDIoption</i>                       | command  |
| puppetSound  | puppetSound 0                                       | command  |
| sound stop   | sound stop <i>whichChannel</i>                      | command  |
| soundBusy    | soundBusy ( <i>whichChannel</i> )                   | command  |
| soundEnabled | the soundEnabled                                    | property |
| soundLevel   | the soundLevel                                      | property |

## System

| Word           | Syntax                        | Category |
|----------------|-------------------------------|----------|
| the            | the <i>property</i>           | keyword  |
| beep           | beep [ <i>numberOfTimes</i> ] | command  |
| beepOn         | the beepOn                    | property |
| colorDepth     | the colorDepth                | property |
| colorQD        | the colorQD                   | function |
| floatPrecision | the floatPrecision            | property |
| freeBlock      | the freeBlock                 | function |
| freeBytes      | the freeBytes                 | function |
| machineType    | the machineType               | function |
| memorySize     | the memorySize                | function |
| quit           | quit                          | command  |
| restart        | restart                       | command  |
| shutDown       | shutDown                      | command  |



| Word             | Syntax               | Category        |
|------------------|----------------------|-----------------|
| stageBottom      | the stageBottom      | function        |
| stageColor       | the stageColor       | property        |
| stageLeft        | the stageLeft        | function        |
| stageRight       | the stageRight       | function        |
| stageTop         | the stageTop         | function        |
| switchColorDepth | the switchColorDepth | property        |
| version          | version              | system variable |

## Text

| Word            | Syntax                                                                           | Category            |
|-----------------|----------------------------------------------------------------------------------|---------------------|
| after           | put <i>expression</i> after <i>chunkExpression</i>                               | keyword             |
| alert           | alert <i>message</i>                                                             | command             |
| before          | put <i>expression</i> before <i>chunkExpression</i>                              | keyword             |
| char            | char <i>whichCharacter</i> of <i>chunkExpression</i>                             | keyword             |
| char...to       | char <i>firstCharacter</i> to <i>lastCharacter</i> of <i>chunkExpression</i>     | keyword             |
| chars           | chars ( <i>stringExpression</i> , <i>firstCharacter</i> , <i>lastCharacter</i> ) | function            |
| charToNum       | charToNum( <i>stringExpression</i> )                                             | function            |
| contains        | <i>stringExpression1</i> contains <i>stringExpression2</i>                       | comparison operator |
| delete          | delete <i>chunkExpression</i>                                                    | command             |
| do              | do <i>stringExpression</i>                                                       | command             |
| editableText    | editableText                                                                     | command             |
| field           | field <i>whichField</i>                                                          | keyword             |
| hilite          | hilite <i>chunkExpression</i>                                                    | command             |
| into            | put <i>expression</i> into <i>chunkExpression</i>                                | keyword             |
| item            | item <i>whichItem</i> of <i>chunkExpression</i>                                  | keyword             |
| item...to       | item <i>firstItem</i> to <i>lastItem</i> of <i>chunkExpression</i>               | keyword             |
| length          | length( <i>stringExpression</i> )                                                | function            |
| line            | line <i>whichLine</i> of <i>chunkExpression</i>                                  | keyword             |
| line...to       | line <i>firstLine</i> to <i>lastLine</i> of <i>chunkExpression</i>               | keyword             |
| number of chars | the number of chars in <i>chunkExpression</i>                                    | chunk function      |
| number of items | the number of items in <i>chunkExpression</i>                                    | chunk function      |
| number of lines | the number of lines in <i>chunkExpression</i>                                    | chunk function      |
| number of words | the number of words in <i>chunkExpression</i>                                    | chunk function      |
| numToChar       | numToChar( <i>n</i> )                                                            | function            |
| offset          | offset ( <i>target</i> , <i>source</i> )                                         | function            |

| Word         | Syntax                                                             | Category            |
|--------------|--------------------------------------------------------------------|---------------------|
| put...after  | put <i>expression</i> after <i>chunkExpression</i>                 | command             |
| put...before | put <i>expression</i> before <i>chunkExpression</i>                | command             |
| put...into   | put <i>expression</i> into <i>chunkExpression</i>                  | command             |
| selection    | the selection                                                      | function            |
| selEnd       | the selEnd                                                         | text property       |
| selStart     | the selStart                                                       | text property       |
| starts       | <i>string1</i> starts <i>string2</i>                               | comparison operator |
| string       | string ( <i>expression</i> )                                       | function            |
| text         | the text of cast <i>whichCastmember</i>                            | text property       |
| textAlign    | the textAlign of field <i>whichField</i>                           | text property       |
| textFont     | the textFont of field <i>whichField</i>                            | text property       |
| textHeight   | the textHeight of field <i>whichField</i>                          | text property       |
| textSize     | the textSize of field <i>whichField</i>                            | text property       |
| textStyle    | the textStyle of field <i>whichField</i>                           | text property       |
| value        | value ( <i>string</i> )                                            | function            |
| word         | word <i>whichWord</i> of <i>chunkExpression</i>                    | keyword             |
| word...to    | word <i>firstWord</i> to <i>lastWord</i> of <i>chunkExpression</i> | keyword             |

## Time

| Word           | Syntax                                                                                     | Category |
|----------------|--------------------------------------------------------------------------------------------|----------|
| date           | the [short long abbreviated abbrev abbr] date                                              | function |
| delay          | delay <i>numberOfTicks</i>                                                                 | command  |
| framesToHMS    | framesToHMS ( <i>frames</i> , <i>tempo</i> , <i>dropFrame</i> , <i>fractionalSeconds</i> ) | function |
| HMStoFrames    | HMStoFrames ( <i>frames</i> , <i>tempo</i> , <i>dropFrame</i> , <i>fractionalSeconds</i> ) | function |
| startTimer     | startTimer                                                                                 | command  |
| ticks          | the ticks                                                                                  | function |
| time           | the [short long abbreviated abbrev abbr] time                                              | function |
| timeOut        | when timeOut then <i>statement</i>                                                         | command  |
| timeoutKeyDown | the timeoutKeyDown                                                                         | property |
| timeoutLapsed  | the timeoutLapsed                                                                          | property |
| timeoutLength  | the timeoutLength                                                                          | property |
| timeoutMouse   | the timeoutMouse                                                                           | property |
| timeoutPlay    | the timeoutPlay                                                                            | property |
| timeoutScript  | the timeoutScript                                                                          | property |
| timer          | the timer                                                                                  | property |



## Variables

| Word        | Syntax                                         | Category |
|-------------|------------------------------------------------|----------|
| put         | put <i>expression</i> [into <i>variable</i> ]  | command  |
| set...=     | set <i>variable</i> = <i>expression</i>        | command  |
| set...=     | set [the] <i>property</i> = <i>expression</i>  | command  |
| set...to    | set <i>variable</i> to <i>expression</i>       | command  |
| set...to    | set [the] <i>property</i> to <i>expression</i> | command  |
| showGlobals | showGlobals                                    | command  |
| showLocals  | showLocals                                     | command  |

## Keyboard Shortcuts

### Always Available

| Keys               | Actions                                         |
|--------------------|-------------------------------------------------|
| Command-Shift-U    | opens Movie Script                              |
| Command-M          | opens Message window                            |
| Command-W          | stops movie playback                            |
| Command-. (period) | stops movie playback                            |
| Command-R          | rewinds a movie (puts playback head at frame 1) |
| Command-A          | starts playback of a movie                      |

### In Script Window

| Keys               | Actions                                   |
|--------------------|-------------------------------------------|
| Command-Up Arrow   | increases font size only in script window |
| Command-Down Arrow | decreases font size only in script window |

## ***In Score Window***

| Keys                | Actions                                               |
|---------------------|-------------------------------------------------------|
| Command-D           | steps back one frame                                  |
| Command-F           | steps forward one frame                               |
| Command-Right Arrow | goes to the next marker (or jumps 10 frames)          |
| Command-Left Arrow  | goes to the previous marker (or jumps back 10 frames) |

## ***In Text and Script Windows***

| Keys      | Actions                          |
|-----------|----------------------------------|
| Command-H | does Find command                |
| Command-G | does Find Again command          |
| Command-Y | does Find In Next Script command |
| Command-T | does Replace and Find command    |

## ***In Message Window***

| Keys               | Actions                                                                       |
|--------------------|-------------------------------------------------------------------------------|
| Command-Up Arrow   | moves to top of Message window                                                |
| Command-Down Arrow | moves to bottom of Message window                                             |
| Command-Delete     | clears text in Message window from the insertion point to the end of the text |

## ***In Cast Window***

| Keys                 | Action                                      |
|----------------------|---------------------------------------------|
| Control-click        | opens the castmember's Cast Info dialog box |
| Control-Option-click | opens the castmember's Cast Script          |

## ***On Stage Sprite***

| Keys                 | Action                                  |
|----------------------|-----------------------------------------|
| Control-click        | opens the sprite's Cast Info dialog box |
| Control-Option-click | opens the sprite's Cast Script          |



## Appendix B: ASCII Chart

This appendix shows the Macintosh character set, and additional ASCII characters which are not necessarily present in all Macintosh fonts.

### Macintosh Character Set

| Character | Hex | Decimal | Name | Keystrokes on US keyboard |
|-----------|-----|---------|------|---------------------------|
|           | 00  | 0       | NUL  |                           |
|           | 01  | 1       | SOH  |                           |
|           | 02  | 2       | STX  |                           |
|           | 03  | 3       | ETX  | Enter                     |
|           | 04  | 4       | EOT  |                           |
|           | 05  | 5       | ENQ  |                           |
|           | 06  | 6       | ACK  |                           |
|           | 07  | 7       | BEL  |                           |
|           | 08  | 8       | BS   | Delete                    |
|           | 09  | 9       | HT   | Tab                       |
|           | 0A  | 10      | LF   |                           |
|           | 0B  | 11      | VT   |                           |
|           | 0C  | 12      | FF   |                           |
|           | 0D  | 13      | CR   | Return                    |
|           | 0E  | 14      | SO   |                           |
|           | 0F  | 15      | SI   |                           |
|           | 10  | 16      | DLE  |                           |
|           | 11  | 17      | DC1  |                           |

| Character | Hex | Decimal | Name  | Keystrokes on US keyboard |
|-----------|-----|---------|-------|---------------------------|
|           | 12  | 18      | DC2   |                           |
|           | 13  | 19      | DC3   |                           |
|           | 14  | 20      | DC4   |                           |
|           | 15  | 21      | NAK   |                           |
|           | 16  | 22      | SYN   |                           |
|           | 17  | 23      | ETB   |                           |
|           | 18  | 24      | CAN   |                           |
|           | 19  | 25      | EM    |                           |
|           | 1A  | 26      | SUB   |                           |
|           | 1B  | 27      | ESC   | Clear                     |
|           | 1C  | 28      | FS    | Left arrow                |
|           | 1D  | 29      | GS    | Right arrow               |
|           | 1E  | 30      | RS    | Up arrow                  |
|           | 1F  | 31      | US    | Down arrow                |
|           | 20  | 32      | space | Spacebar                  |
| !         | 21  | 33      |       | !                         |
| "         | 22  | 34      |       | "                         |
| #         | 23  | 35      |       | #                         |
| \$        | 24  | 36      |       | \$                        |
| %         | 25  | 37      |       | %                         |
| &         | 26  | 38      |       | &                         |
| '         | 27  | 39      |       | '                         |
| ( ,       | 28  | 40      |       | (                         |
| )         | 29  | 41      |       | )                         |
| *         | 2A  | 42      |       | *                         |
| +         | 2B  | 43      |       | +                         |
| ,         | 2C  | 44      |       | ,                         |
| -         | 2D  | 45      |       | -                         |
| .         | 2E  | 46      |       | .                         |
| /         | 2F  | 47      |       | /                         |
| 0         | 30  | 48      |       | 0                         |
| 1         | 31  | 49      |       | 1                         |
| 2         | 32  | 50      |       | 2                         |
| 3         | 33  | 51      |       | 3                         |



| Character | Hex | Decimal | Name | Keystrokes on US keyboard |
|-----------|-----|---------|------|---------------------------|
| 4         | 34  | 52      |      | 4                         |
| 5         | 35  | 53      |      | 5                         |
| 6         | 36  | 54      |      | 6                         |
| 7         | 37  | 55      |      | 7                         |
| 8         | 38  | 56      |      | 8                         |
| 9         | 39  | 57      |      | 9                         |
| :         | 3A  | 58      |      | :                         |
| ;         | 3B  | 59      |      | ;                         |
| <         | 3C  | 60      |      | <                         |
| =         | 3D  | 61      |      | =                         |
| >         | 3E  | 62      |      | >                         |
| ?         | 3F  | 63      |      | ?                         |
| @         | 40  | 64      |      | @                         |
| A         | 41  | 65      |      | A                         |
| B         | 42  | 66      |      | B                         |
| C         | 43  | 67      |      | C                         |
| D         | 44  | 68      |      | D                         |
| E         | 45  | 69      |      | E                         |
| F         | 46  | 70      |      | F                         |
| G         | 47  | 71      |      | G                         |
| H         | 48  | 72      |      | H                         |
| I         | 49  | 73      |      | I                         |
| J         | 4A  | 74      |      | J                         |
| K         | 4B  | 75      |      | K                         |
| L         | 4C  | 76      |      | L                         |
| M         | 4D  | 77      |      | M                         |
| N         | 4E  | 78      |      | N                         |
| O         | 4F  | 79      |      | O                         |
| P         | 50  | 80      |      | P                         |
| Q         | 51  | 81      |      | Q                         |
| R         | 52  | 82      |      | R                         |
| S         | 53  | 83      |      | S                         |
| T         | 54  | 84      |      | T                         |
| U         | 55  | 85      |      | U                         |

| Character | Hex | Decimal | Name | Keystrokes on US keyboard |
|-----------|-----|---------|------|---------------------------|
| V         | 56  | 86      |      | V                         |
| W         | 57  | 87      |      | W                         |
| X         | 58  | 88      |      | X                         |
| Y         | 59  | 89      |      | Y                         |
| Z         | 5A  | 90      |      | Z                         |
| [         | 5B  | 91      |      | (                         |
| \         | 5C  | 92      |      | \                         |
| ]         | 5D  | 93      |      | )                         |
| ^         | 5E  | 94      |      | ^                         |
| _         | 5F  | 95      |      | _                         |
| `         | 60  | 96      |      | `                         |
| a         | 61  | 97      |      | a                         |
| b         | 62  | 98      |      | b                         |
| c         | 63  | 99      |      | c                         |
| d         | 64  | 100     |      | d                         |
| e         | 65  | 101     |      | e                         |
| f         | 66  | 102     |      | f                         |
| g         | 67  | 103     |      | g                         |
| h         | 68  | 104     |      | h                         |
| i         | 69  | 105     |      | i                         |
| j         | 6A  | 106     |      | j                         |
| k         | 6B  | 107     |      | k                         |
| l         | 6C  | 108     |      | l                         |
| m         | 6D  | 109     |      | m                         |
| n         | 6E  | 110     |      | n                         |
| o         | 6F  | 111     |      | o                         |
| p         | 70  | 112     |      | p                         |
| q         | 71  | 113     |      | q                         |
| r         | 72  | 114     |      | r                         |
| s         | 73  | 115     |      | s                         |
| t         | 74  | 116     |      | t                         |
| u         | 75  | 117     |      | u                         |
| v         | 76  | 118     |      | v                         |
| w         | 77  | 119     |      | w                         |



| Character | Hex | Decimal | Name | Keystrokes on US keyboard    |
|-----------|-----|---------|------|------------------------------|
| x         | 78  | 120     |      | x                            |
| y         | 79  | 121     |      | y                            |
| z         | 7A  | 122     |      | z                            |
| {         | 7B  | 123     |      | {                            |
|           | 7C  | 124     |      |                              |
| }         | 7D  | 125     |      | }                            |
| ~         | 7E  | 126     |      | ~                            |
|           | 7F  | 127     | DEL  |                              |
| Ä         | 80  | 128     |      | Option-U, Shift-A            |
| Å         | 81  | 129     |      | Option-Shift-A               |
| Ç         | 82  | 130     |      | Option-Shift-C               |
| É         | 83  | 131     |      | Option- E, Shift-E           |
| Ñ         | 84  | 132     |      | Option-N, Shift-N            |
| Ö         | 85  | 133     |      | Option-U, Shift-O            |
| Ü         | 86  | 134     |      | Option-U, Shift-U            |
| á         | 87  | 135     |      | Option-E, a                  |
| à         | 88  | 136     |      | Option-~ ( <i>tilde</i> ), a |
| â         | 89  | 137     |      | Option-I, a                  |
| ä         | 8A  | 138     |      | Option-U, a                  |
| ã         | 8B  | 139     |      | Option-N, a                  |
| å         | 8C  | 140     |      | Option-A                     |
| ç         | 8D  | 141     |      | Option-C                     |
| é         | 8E  | 142     |      | Option-E, e                  |
| è         | 8F  | 143     |      | Option-~ ( <i>tilde</i> ), e |
| ê         | 90  | 144     |      | Option-I, e                  |
| ë         | 91  | 145     |      | Option-U, e                  |
| í         | 92  | 146     |      | Option-E, i                  |
| ì         | 93  | 147     |      | Option-~ ( <i>tilde</i> ), i |
| î         | 94  | 148     |      | Option-I, i                  |
| ï         | 95  | 149     |      | Option-U, i                  |
| ñ         | 96  | 150     |      | Option-N, n                  |
| ó         | 97  | 151     |      | Option-E, o                  |
| ò         | 98  | 152     |      | Option-~ ( <i>tilde</i> ), o |
| ô         | 99  | 153     |      | Option-I, o                  |

| Character | Hex | Decimal | Name | Keystrokes on US keyboard       |
|-----------|-----|---------|------|---------------------------------|
| ö         | 9A  | 154     |      | Option-U, o                     |
| õ         | 9B  | 155     |      | Option-N, o                     |
| ú         | 9C  | 156     |      | Option-E, u                     |
| ù         | 9D  | 157     |      | Option-~ ( <i>tilde</i> ), u    |
| û         | 9E  | 158     |      | Option-I, u                     |
| ü         | 9F  | 159     |      | Option-U, u                     |
| †         | A0  | 160     |      | Option-T                        |
| °         | A1  | 161     |      | Option-Shift-8                  |
| ¢         | A2  | 162     |      | Option-4                        |
| £         | A3  | 163     |      | Option-3                        |
| §         | A4  | 164     |      | Option-6                        |
| •         | A5  | 165     |      | Option-8                        |
| ¶         | A6  | 166     |      | Option-7                        |
| ß         | A7  | 167     |      | Option-S                        |
| ®         | A8  | 168     |      | Option-R                        |
| ©         | A9  | 169     |      | Option-G                        |
| ™         | AA  | 170     |      | Option-2                        |
| '         | AB  | 171     |      | Option-E, Spacebar              |
| "         | AC  | 172     |      | Option-U, Spacebar              |
| ≠         | AD  | 173     |      | Option-= ( <i>equal sign</i> )  |
| Æ         | AE  | 174     |      | Option-Shift-" ( <i>quote</i> ) |
| Ø         | BF  | 175     |      | Option-Shift-O                  |
| ∞         | B0  | 176     |      | Option-5                        |
| ±         | B1  | 177     |      | Option-Shift-=                  |
| ≤         | B2  | 178     |      | Option-, ( <i>comma</i> )       |
| ≥         | B3  | 179     |      | Option-. ( <i>period</i> )      |
| ¥         | B4  | 180     |      | Option-Y                        |
| μ         | B5  | 181     |      | Option-M                        |
| ∂         | B6  | 182     |      | Option-D                        |
| Σ         | B7  | 183     |      | Option-W                        |
| Π         | B8  | 184     |      | Option-Shift-P                  |
| π         | B9  | 185     |      | Option-P                        |
| ∫         | BA  | 186     |      | Option-B                        |
| ª         | BB  | 187     |      | Option-9                        |



| Character | Hex | Decimal | Name          | Keystrokes on US keyboard               |
|-----------|-----|---------|---------------|-----------------------------------------|
| ø         | BC  | 188     |               | Option-0                                |
| Ω         | BD  | 189     |               | Option-Z                                |
| æ         | BE  | 190     |               | Option-" ( <i>quote</i> )               |
| ø         | BF  | 191     |               | Option-O                                |
| ¿         | C0  | 192     |               | Option-Shift-?                          |
| ¡         | C1  | 193     |               | Option-I                                |
| ¬         | C2  | 194     |               | Option-L                                |
| √         | C3  | 195     |               | Option-V                                |
| f         | C4  | 196     |               | Option-F                                |
| ≈         | C5  | 197     |               | Option-X                                |
| Δ         | C6  | 198     |               | Option-O                                |
| «         | C7  | 199     |               | Option-\ ( <i>backslash</i> )           |
| »         | C8  | 200     |               | Option-Shift-\                          |
| ...       | C9  | 201     |               | Option-; ( <i>semicolon</i> )           |
|           | CA  | 202     | (fixed space) | Option-Spacebar                         |
| À         | CB  | 203     |               | Option-~ ( <i>tilde</i> ), Shift-A      |
| Ã         | CC  | 204     |               | Option-N, Shift-A                       |
| Õ         | CD  | 205     |               | Option-N, Shift-O                       |
| Œ         | CE  | 206     |               | Option-Shift-Q                          |
| œ         | CF  | 207     |               | Option-Q                                |
| –         | D0  | 208     | (n-dash)      | Option- - ( <i>hyphen</i> )             |
| —         | D1  | 209     | (m-dash)      | Option-Shift- - ( <i>hyphen</i> )       |
| “         | D2  | 210     |               | Option-] ( <i>right bracket</i> )       |
| ”         | D3  | 211     |               | Option-Shift-] ( <i>right bracket</i> ) |
| ‘         | D4  | 212     |               | Option-[ ( <i>left bracket</i> )        |
| ’         | D5  | 213     |               | Option-Shift-[ ( <i>left bracket</i> )  |
| ÷         | D6  | 214     |               | Option-/                                |
| ◊         | D7  | 215     |               | Option-Shift-V                          |
| ÿ         | D8  | 216     |               | Option-U, y                             |

## Additional Characters

These characters are not part of the Macintosh character set, but are included with many fonts. The characters above D8 (216) will vary from font to font, so use them with the `textFont` text property. The characters illustrated here are Helvetica.

| Character | Hex | Decimal | Name | Keystrokes on US keyboard           |
|-----------|-----|---------|------|-------------------------------------|
| ÿ         | D9  | 217     |      | Option-Shift-~ ( <i>tilde</i> )     |
| /         | DA  | 218     |      | Option-Shift-1                      |
| ²         | DB  | 219     |      | Option-Shift-2                      |
| ³         | DC  | 220     |      | Option-Shift-3                      |
| ´         | DD  | 221     |      | Option-Shift-4                      |
| fi        | DE  | 222     |      | Option-Shift-5                      |
| fl        | DF  | 223     |      | Option-Shift-6                      |
| ‡         | E0  | 224     |      | Option-Shift-7                      |
| ·         | E1  | 225     |      | Option-Shift-9                      |
| ,         | E2  | 226     |      | Option-Shift-0                      |
| ”         | E3  | 227     |      | Option-Shift-W                      |
| ‰         | E4  | 228     |      | Option-Shift-E                      |
| Â         | E5  | 229     |      | Option-Shift-R                      |
| Ê         | E6  | 230     |      | Option-Shift-T                      |
| Á         | E7  | 231     |      | Option-Shift-Y                      |
| Ë         | E8  | 232     |      | Option-Shift-U                      |
| È         | E9  | 233     |      | Option-Shift-I                      |
| Í         | EA  | 234     |      | Option-Shift-S                      |
| Î         | EB  | 235     |      | Option-Shift-D                      |
| Ï         | EC  | 236     |      | Option-Shift-F                      |
| Ì         | ED  | 237     |      | Option-Shift-G                      |
| Ó         | EE  | 238     |      | Option-Shift-H                      |
| Ô         | EF  | 239     |      | Option-Shift-J                      |
| ⌘         | F0  | 240     |      | Option-Shift-K                      |
| Ò         | F1  | 241     |      | Option-Shift-L                      |
| Ú         | F2  | 242     |      | Option-Shift-; ( <i>semicolon</i> ) |
| Û         | F3  | 243     |      | Option-Shift-Z                      |
| Ü         | F4  | 244     |      | Option-Shift-X                      |
| ı         | F5  | 245     |      | Option-Shift-B                      |



| Character | Hex | Decimal | Name | Keystrokes on US keyboard        |
|-----------|-----|---------|------|----------------------------------|
| ^         | F6  | 246     |      | Option-Shift-N                   |
| ~         | F7  | 247     |      | Option-Shift-M                   |
| -         | F8  | 248     |      | Option-Shift-, ( <i>comma</i> )  |
| ~         | F9  | 249     |      | Option-Shift-. ( <i>period</i> ) |
| ·         | FA  | 250     |      | Option-H                         |
| „         | FB  | 251     |      |                                  |
| “         | FC  | 252     |      |                                  |
| „         | FD  | 253     |      |                                  |
| ”         | FE  | 254     |      |                                  |





# Appendix C: Suggested Reading

Here are some publications that can increase your general knowledge of movie design.

## Books

*Apple Human Interface Guidelines*, Reading, Massachusetts: Addison-Wesley, 1987.

Friedhoff, Richard Mark and William Benzon, *Visualization: The Second Computer Revolution*, New York, NY: Abrams, 1989.

Kamins, Scot and Dan Winkler, *HyperCard 2.0: The Book*, New York, NY: Bantam Computer Books, 1990.

*HyperTalk Script Language Guide*, Reading, Massachusetts: Addison-Wesley, 1988.

*HyperCard User Interface Guidelines*, Reading, Massachusetts: Addison-Wesley, 1987, revised 1988.

Tufte, Edward R., *Envisioning Information*, Cheshire, Connecticut: Graphics Press, 1990.

## Magazines

*AV Video*, Torrance, California, Montage Publishing, Inc.

*Computer Graphics World*, Westford, Massachusetts, PennWell Publishing Co.

## Newsletter

*MacroMind Overview*, San Francisco, California, MacroMind, Inc.





# Glossary

**AIFF** An acronym for Audio Interchange File Format. A standard for exchanging sound information between applications.

**aliasing** In sampling an image, the impairment produced when the input signal contains frequency components higher than half of the sampling rate. In computer animations, this impairment is colloquially known as “the jaggies:” jagged steps on diagonal lines, and is more apparent for smaller images. Compare **anti-aliasing**.

**anti-aliasing** The process of reducing the visibility of aliasing by using gradient pixel values to smooth the appearance of jagged edges. If this is done well, the eye creates smooth curved lines and diagonals.

**ampersand (&)** The name of the symbol that means “and”; in Lingo, & is a **concatenation operator** that joins together two or more strings.

**animation** A series of still images which, when presented in rapid succession, create the illusion of motion.

**argument** A value upon which a **handler**, **macro**, or **function** operates. Arguments are passed as **parameters** from other parts of the code when you **call** a handler or function. Arguments can also be included when sending a message to an object to use one of its methods.

**array** An arrangement of data elements in memory that can be named, and from which data can be retrieved by providing the name of the array and the index number of the data element to be retrieved. All objects created by a Lingo factory automatically contain an array. Using the

predefined Lingo methods `mPut` and `mGet`, various data types can be assigned and retrieved from the array, including: numbers, character strings, symbols, and objects.

**ASCII / ASCII Character** Acronym for “American Standard Code for Information Interchange.” In ASCII, alphanumeric (letters, numbers) characters and special codes are assigned a number between 0 and 128.

**background** Objects that appear to be behind other objects on the Stage, or are in a lower-numbered channel in the Score are said to be in the background.

**bitmap** A graphic representation composed of individual pixels.

**boolean** Of or pertaining to the algebraic system invented by George Boole. In boolean algebra, expressions are evaluated to either 0 or 1, representing a FALSE or TRUE condition, respectively.

**boolean value** A value of either 0 or 1, representing a FALSE or TRUE condition, respectively.

**button** In the Macintosh computer user interface, a graphic icon on the screen which, when clicked with the mouse pointer, results in some action taking place. Buttons are usually activated by single mouseclicks.

**C** A software programming language frequently used to create application programs and code resources such as XObjects on the Macintosh computer.

**call** In programming, to summon and execute. For example, if you’ve defined a handler named `myHandler`, you call the handler by using just its name in the calling script.

**camcorder** A device for recording moving video images. The camcorder combines what used to be two separate functions into one mechanical unit: a camera and a video recorder, hence camcorder.

**castmember** Any individual graphic, text, film loop, sound, or palette stored in Director’s Cast window.

**CAV (CAVL)** An acronym for Constant Angular Velocity. A way of encoding information on a videodisk such that the disk always rotates at a constant 1800 rpm (revolutions per minute). This type of encoding allows freeze frames, slow motion, and other special effects on playback.



**CD-ROM** An acronym for Compact Disk—Read Only Memory. An optical storage medium that can hold large amounts of digital data, including programs, animation, and sound.

**cell** The smallest unit in Director's Score window. Each **sprite channel** is composed of a row of cells, one cell per frame.

**channel** A single row in Director's Score window. Each channel holds one piece of artwork or a special effect. The script channel contains scripts, the sound channels contain sounds, and so on.

**chrominance** In computer and video display systems, the signals which represent the color components of an image, such as hue and saturation. A black-and-white image will have chrominance values of zero. In the NTSC television system, the I and Q signals carry the chrominance information. Chrominance is sometimes abbreviated as *chroma*.

**character** A primary unit of information, such as a letter or number. See **ASCII / ASCII character**.

**chunk** A portion of a value in a container. A chunk of any container is, itself, a container.

**Chunk Size** In Director, a value that represents the number of pixels a transition moves in each step of the transition. You can set Chunk Size for every transition with the `setTransition` command.

**CLV** An acronym for Constant Linear Velocity. A way of encoding information on videodisk so that the rate at which the disk is rotated decreases as the playback beam moves toward the outside of the disk, maintaining a constant velocity of the disk track relative to the pickup beam. This method allows for a greater amount of video information to be recorded on the disk at the expense of special playback effects such as freeze frame and slow motion.

**color cycling** In Director, the process of changing color palettes so that ranges of colors are displayed in succession.

**color depth** A measure of the number of colors that can be displayed on the screen for a given **pixel**. A pixel with a color depth of 1-bit can only be black or white (two colors). A pixel with a color depth of 8-bits can be any one of 256 colors.



**color look-up table (CLUT)** A table of color values used internally by computer programs that manipulate color information.

**command** Any Lingo instruction that causes something to happen while a movie is playing.

**comment** Any text between the double-dash (--) and the end of a line within a Lingo script or macro. Lingo ignores comments. You use comments to annotate your scripts.

**compiled code** In computer software, code that has been translated from a form readable by human beings (such as C or Pascal) into a form that the computer can use.

**component video** Three color video signals that describe a color image. Typical component systems are RGB, YIQ, and YUV.

**composite video** A color video signal that contains all of the color information in a single signal. Typical composite television signal standards are NTSC, PAL, and SECAM.

**concatenation** The joining of two or more **string values** using the concatenation operators & and &&.

**constant** A value that doesn't change. In Lingo, a name that represents a value, such as `empty` or `false`.

**container** A place in code or memory that holds data supplied either by a Lingo script or by the user of an interactive movie. Examples of containers in Lingo are fields, variables, `theSelection`, the Message Box, and **chunks** of containers.

**control structure** A set of Lingo keywords at the beginning and end (and sometimes within the body) of a statement list, which defines the order of execution of statements within the list, for example `if...then`.

**CR (Carriage Return)** 1) Shorthand for "press the Return key." 2) ASCII character code 13, used to signify that data entry is finished.

**current frame** In the Score window, the current frame is the one marked by the location of the playback head. This location corresponds to the current frame number displayed in the Control Panel.

**data fork** In the Macintosh computer, the part of a file that contains data. Compare **resource fork**.



**deselect** To remove the highlight from an element (button, menu item, or chunk of text). Compare **select**.

**dialog** Shorthand for “dialog box.” Any box that’s displayed on the screen to present information to and/or ask for information from the user before anything else can happen.

**digital video** Video in which all the image information is stored and transmitted in some kind of computer data form. Digital video can then be manipulated and displayed in a variety of ways by a computer.

**digitize** The process of converting an analog signal into a digital signal. In Director, digitizing may refer to some form of scanning, or to the process of converting an image from analog to digital format, for example, a video camera connected to a box that converts the video image into a pixel image that the computer can display.

**dithering** The process of changing a range of grey scales to patterns of black-and- white.

**DVI** An acronym for Digital Video Interactive. DVI is the name of a technology developed at David Sarnoff Research Center for hardware and software that makes it easier to manage an all-digital multimedia system (combining audio, video, and computer information).

**driver** A piece of software that provides an interface for and control over a specific piece of hardware, such as a printer, videodisk player, or VTR.

**empty** A string of length 0; the null string; holding no value.

**empty script** A script that doesn’t have any executable statements in it.

**evaluate** To determine the value of an expression.

**event** An event is a specific kind of message generated by the user, by Lingo, or by the Macintosh system. Examples of Lingo events include keypresses (`keyDown`), the press or release of the mouse (`mouseDown`, `mouseUp`), or a system timeout (`timeOut`).

**event script** One or more Lingo statements, defined by the `when` command, that are executed on the occurrence of specific events, such as the press of the mouse or a keyboard button.

**execute** To activate. A script is executed when its statements are sent to Director to cause the scripted actions to take place.



**expression** Any value, group of values, or source(s) of a value, possibly combined with **operators**, meant to be taken as a whole.

**external device** Any piece of equipment that is not normally part of a generic computer system, such as fog makers, fireworks display controls, videodisk players, VTRs, and CD-ROM drives.

**factor** The first fully-resolvable part of a Lingo expression.

**factory** In Lingo, a set of programmer-defined objects that are stored together in a Movie Script.

**file** Any named collection of information that is stored on a disk. Director movie documents are files.

**formatting** The way that text appears in a script, or that numbers appear in a visible container.

**frame accurate** In video editing, the ability to record a single frame at a time.

**frame grabbing** In video editing and computer video applications, the ability to digitize and capture a single frame of the image for further manipulation on the computer.

**frame rate** See FPS.

**frame** 1) In Director, a single column of information in the Score window. The column organizes all the elements that are displayed on the stage and any related activity (such as playback of a sound) that is supposed to happen at the same time. 2) In video and display technology, the complete scanning of one image. Full-motion video displays 30 frames per second (FPS). 3) In filmmaking, a frame is any single still image. Movies are made up of thousands of frames that are projected one at a time (usually at 24 FPS)

**frame label** A name that is attached to a specific frame in the Score window. Thereafter, you use the frame label within a script to refer to the frame.

**FPS** An acronym for frames per second: the rate at which images are displayed on the screen. Video images are displayed at 30 FPS.

**function call** Use of a **function** to get a **value**.



**function handler** In Lingo, a function that you define.

**function** A Lingo word that calculates and returns a value.

**genlock / genlocking** As a noun, a genlock is a device for synchronizing two video signals. As a verb, genlocking is the process of synchronizing two or more video signals. Genlocking is required whenever you want to mix one or more video signals with the output of one or more computer graphic systems.

**global variable** A variable whose content can be changed or accessed by any script, handler, or factory.

**handler** A named group of Lingo statements that are meant to work together. A handler is defined by the keywords `on` and `end`. The handler's name then becomes the message to which the handler responds.

**Hi-8** A trademarked name for a particular videotape cassette recording, storage, and playback format whose hallmark is high quality image storage and playback from a small cassette.

**hue** The apparent color of a given point: red, green, yellow, violet, and so on.

**initialize** To create a **global** or **local variable** and assign it its first value.

**instance variable** A variable that is defined within a **factory**. The content of an instance variable can be changed or accessed only within the factory that uses it. Compare **local** and **global variable**.

**integer** A whole number without any decimal fractional part. Compare floating point.

**interactive** Interactivity is a matter of degree. Almost all computer programs are interactive, insofar as the user provides input, direction, and data to make the program work. Director itself, while you are engaged in the process of creating movies, is highly interactive. You can design less-interactive presentations and movies that are meant to be run and experienced without any input from the viewer or user. Lingo and Director together let you create user experiences that are highly interactive.



**interface** That part of a computer program or Director movie that presents information to and gets information from the user. The interface is composed of a graphic screen presentation, user syntax (mouseclicks, dragging, and so on), and feedback to the user from the program or presentation.

**iteration** A single pass through a set of repeating Lingo statements.

**JPEG** An acronym for Joint Photographic Experts Group, a standard setting committee that is working on a standardization of the algorithms for compressing still image information. With image compression, information is encoded so that it takes less space and less time to transmit. It is then decoded and displayed as a normal image.

**key color** a color signal that controls how elements of one picture will be replaced by elements of another. A common example is the weather map displayed behind your local TV weatherperson. The map itself is generated by computer. The weathermaven stands in front of a backdrop of a particular color, usually blue, called the key color. All the blue parts of the picture that contains the person are then replaced (in the signal that eventually goes to your TV) by the map display. In Director, you can use this technique to create titles for video productions, for example.

**keying** In video, the process of inserting one picture into another picture under control of another signal, called the keying signal. The keying signal is usually a **key color**.

**keyword** A Lingo vocabulary word that is interpreted directly by Lingo, without passing through the message-passing path.

**label** In the Score window, a word or phrase that you attach to a frame that has been marked with a **marker**. You can then use the label in your scripts in lieu of frame numbers.

**LANC** An acronym for Local Application Control Bus. A hardware and signal standard that allows devices such as camcorders, edit controllers, and videotape recorders to be controlled by other devices, such as a remote controller or computer.

**literal** Any **string** meant to be taken for its actual rather than its symbolic value.



**local variable** A variable whose value can be changed or accessed only by the macro or factory that uses it.

**logical** A type of operation that results in either a 0 (False) or 1 (True) value; the kind of operator (*and*, *or*) used to produce a logical value.

**loop** A repeating sequence of commands or any repeating audio or visual sequence in a presentation.

**luminance** In a video image, refers to the brightness values of all the points in the image. A luminance-only representation of an image is black-and-white.

**marker** In the Score window, a black triangle symbol that is attached to a particular frame. The marker indicates that the frame has a **label**.

**marker channel** In the Score window, the channel that shows whether or not a frame has a **label**. If it does, then the frame is marked, in the marker channel, by a black triangle indicator.

**marker well** In the Score window, the black triangle area in the left border of the marker channel. You set a marker for a frame by dragging a marker from the marker well to the frame in the marker channel that you want to **label**.

**message** Any statement that can be responded to by Lingo. Messages can be generated by a script, system event, or user actions.

**Message window** A testing and debugging tool for Lingo scripts.

**MIDI** An acronym for Musical Instrument Digital Interface. 1) The name of the standards and methods, hardware, and software that have been developed for digitally capturing, manipulating, recording, and playing back sounds. 2) A serial digital bus specification for interfacing digital musical instruments. MIDI is widely used in the music industry, and got its start as a microcomputer standard.

**modulo** A division operation that returns only the remainder resulting from the division.

**Movie Script** A script that's associated with a movie. It usually contains handlers that can be executed globally, throughout a movie, regardless of where the playback head is located.

**MPEG** An acronym for Motion Picture Experts Group, a standards committee working on algorithm standards that will allow compression, storage, and transmission of moving image information.

**multimedia** Short for "multiple media." A computer data set which can include more than one type of medium. By this definition, sound and images that can be manipulated and played back on a computer are examples of multimedia. In general, the term is used to cover any combination of (digital or analog) video, text, sound, (computer) graphics, and external devices that may be connected and coordinated by a computer program.

**null string** A string that contains no characters; the value of the expression " "; empty.

**numeric** Having a value composed of digits (numbers from 0 to 9). Numeric values can be **integer** or **floating point** values.

**object oriented programming (OOP)** A programming methodology in which every element in a program is self-contained. That is, each element has within itself all the data and instructions that operate on that data as appropriate for that object. One element sends a **message** to another, and the recipient carries out the task independently.

**operand** The part of a Lingo expression upon which an operator performs some action to produce a value.

**operator** A Lingo symbol or word that changes the value of a single operand, or that combines two operands to produce a single value.

**order of execution (See precedence)**

**palette** The set of colors associated with a graphic object; the on-screen presentation of a set of related tools (the drawing palette, for example); when used with CLUTs, the range of colors from which the CLUT colors can be selected. The number of colors in the palette is equal to 2 raised to the (number of bits in a CLUT entry) power.

**parameter** A value that is passed to a function or command.

**Pascal** A computer programming language.

**pass** To transfer control from one script or part of a script to another. When speaking of values or parameters, to send the values to the called script. When speaking of a message, to put back on the message path.



**pattern** A regular graphic design that covers an area of pixels on the screen.

**PICT** A graphic file format for saving information created in a paint or draw application.

**pixel** Shorthand term for “picture element.” A pixel is one dot on the screen.

**playback head** In Studio, the small black square at the bottom of the Score window that indicates the frame that is currently displayed on the Stage. In Overview, the rectangle that travels above each column of slides in the Overview window and indicates which slide is currently on the Stage.

**precedence** Rules for the order of execution of Lingo statement elements and expressions.

**property** An attribute of an object. One of the properties of a text castmember, for example, is the text style.

**puppet** A sprite whose movements and actions are controlled by a script. Once it has been declared to be a puppet, a sprite can no longer be controlled by the Score, but only by a Lingo script.

**quoted literal** In a Lingo statement, any set of characters that appears between a set of double quotation marks.

**relational operation / relational operator** 1) A comparison of two values that results in a boolean value (true or false). 2) A type of operator (<,>,<=>, =, etc.) used to compare two values and return either a true or false.

**resource** All the externals, sounds, fonts, icons, and other elements used by a program (including Director movies), stored in the resource fork of a file.

**resource fork** In the Macintosh computer, the part of a file that contains information that’s meaningful only to the Macintosh, such as application code, icons, or cursor bitmaps. Compare **data fork**.

**return** 1) To evaluate a Lingo expression and pass its value to the calling function. 2) The Return key on the keyboard.



**saturation** The depth of color intensity. Zero saturation is white (no color). Maximum saturation of a given color is the deepest, most intense color possible.

**scan line** In a scanner, a single pass of the scanning head across the image area. In a video display, a single line of pixels from left to right.

**Score** The contents of the animation, effect, and script channels in the Score window. The Score controls how a movie is played.

**Score window** A notational editor for the animation score that displays the animation information from left to right and represents events through time. The score uses its own notation conventions just as a musical score does.

**script** A related set of Lingo **statements** that control Director.

**script number** The number shown in the Score cells when the "Script" option is selected from the Score display pop-up menu in the lower-left border of the Score window.

**select** To highlight by clicking, clicking and dragging, or double-clicking. You can select a checkbox, radio button, or text chunks.

**selection** A highlighted portion of text or button that indicates the action that will be performed next, or the element on which the next action will be performed.

**Stage** The window in which animation is displayed. The content of the Stage at any point in time is determined by the current frame in Studio or the current slide in Overview.

**sprite script** One or more Lingo statements associated with a sprite. The sprite script is executed when the user clicks on the sprite with the mouse.

**sprite** One piece of artwork at one point in time. Sprites are instances of animated art or text, called castmembers. A single castmember can become any number of sprites on the stage.

**standard file dialog** In the Macintosh computer environment, a dialog box from which you choose a file. The standard file dialog looks and operates the same way across all Macintosh applications, including Director.



**statement list** A group of Lingo commands separated by return characters. Statements outside of control structures are executed in the order listed. Statements within control structures are executed according to the results of operations that take place within the control structure.

**statement** Any single line of valid Lingo code.

**string** A group of characters.

**symbol** Any variable name preceded by the Lingo keyword #.

**syntax** The rules governing how Lingo statements are written.

**tick** In the Macintosh computer, a time unit of 1/60th of a second duration. Sixty ticks equals one second.

**trace** To make a transcription of the sequence of execution of a script or group of scripts. The transcription is then used for debugging or further testing. In Lingo, the Message window has a Trace checkbox which, when checked, creates a recording of the execution of scripts during playback of a movie.

**value** 1) A quantity assigned to a constant, variable, parameter or symbol. 2) The result of the computation of an expression.

**variable** A temporary holder of a value. A variable is defined the first time you name it and give it an initial value. Compare **global variable**, **local variable**, **instance variable**, **object variable**.

**VCR** An acronym for videocassette recorder, a device for recording and playing back video signals.

**VHS (S-VHS)** A trademarked name for the first popular videotape cassette recording, storage, and playback format whose hallmark is consumer acceptance and the resulting economies of scale.

**video input card** A card that can be installed in a modular Macintosh II computer to support a variety of special video features, such as video overlay, video-in-a-window, and frame grabbing.

**video-in-window** A feature of some video input cards that allows you to mix computer graphics and real-time video. You can overlay Macintosh animations or graphics on top of the video, or vice versa.



**Video8** A trademarked name for a videotape cassette recording, storage, and playback format whose hallmark is consumer quality image storage and playback from a small cassette.

**videodisk** An optical storage medium for full-motion video images. The main advantage of videodisks is that you can choose to display images from anywhere on the disk in a short period of time (called random access display), compared with videotape. Most videodisks can only be played back (read-only). Videodisks that can record and play back images are more expensive.

**videotape** A magnetic recording, storage, and playback medium for full-motion video images. The main advantage of tape is that it is a very low-cost storage and distribution medium, and it can be erased and reused.

**VTR** An acronym for video tape recorder, a device that is used to record and play back video signals, using videotape as its storage medium.

**word** In Lingo, a set of characters delimited by spaces, meant to be taken as a whole.

**XCMD** The resource type for and name given to an external command for HyperCard.

**XCOD** The resource type for and name given to external objects in Lingo.

**XFCN** The resource type for and name given to an external function for HyperCard.

**XLibrary** A group of XObjects stored together in a file.

**XObjects** Compiled code modules that Lingo can access.



# *Acknowledgments*

## **Inspiration**

Marc Canter

## **Producer**

David Kaiser

## **Product Manager**

Terry R. Schussler

David Kleinberg

## **Project Manager**

Joe Dunn

## **Engineering**

John Thompson, Lead

Dan Sadowski

Al McNeil

Greg O'Neil

Gordon Smith

Don Melton

John Schlag

Dan Michael

Jay Fenton

Erik Neumann

## **Software Testing**

Don Whitt, Manager

Mark Castle, Lead

Matt Berardo

Mike Fottrell

Jon Gillespie

Jeff Parker

Kenn Johnson

William Schulze

David Shields

### **3.0 Documentation Update**

Robin Foster, *Getting Started*, *Overview*, and *Studio* manuals  
CommuniTree Group, San Francisco, *Interactivity* manual

### **Manual Production**

Grafica Multimedia, Inc., *Getting Started*, *Overview*, and *Studio* manuals  
CommuniTree Group, San Francisco, *Interactivity* manual

### **Package Design**

Clement Mok

### **Art and Design**

James McAllister, Art Director  
Lon Richter, Lead  
Roger Jones  
Linno Llenos  
Roy Santiago  
Stuart Sharpe  
Chuck Walker



# Index

" (double quotation marks)  
for testing character strings, 57  
for text string literals, 57, 99

# (pound sign) symbol definition  
operator, indicating  
symbols, 62, 333, 342

& (concatenation) text operator, 70,  
99, 330, 341

&& (concatenation with space) text  
operator, 56, 70, 99, 330, 341

() (parentheses)  
arithmetic operators, 68  
grouping operator, 326, 341  
*See also* parentheses

\* (asterisk), multiplication arithmetic  
operator, 47, 68, 328, 341

+ (addition), arithmetic operator, 68,  
329

+ (plus sign), indicating castmembers  
which have Cast Scripts, 14,  
20, 341

- (negation), arithmetic operator, 68,  
327, 341

- (subtraction), arithmetic operator,  
68, 329, 341

- (double hyphens), indicating  
comments, 51, 56, 327, 337

/ (division), arithmetic operator, 68,  
328, 342

/ symbol, for creating command key  
equivalents, 49

< (less than) comparison operator,  
70, 331, 342

<= (less than or equal to) comparison  
operator, 70, 331, 342

<> (not equal to) comparison  
operator, 70, 332, 342

= (equal to) comparison operator, 70,  
332, 342

== (double equal signs), indicating  
script location, 51

> (greater than) comparison  
operator, 69-70, 331, 342

> (greater than) pointer, nesting level  
of the script, 51, 342

>= (greater than or equal to)  
comparison operator, 70,  
332, 342

[] (square brackets), indicating  
optional elements, 64-65,  
109, 176

¬ (Option-Return character) special  
symbol, continuation  
symbol, 4, 29, 327-28, 337

≈ sign, creating menu items, 48, 79,  
238

## A

A11...H88 cast identifiers, 177, 336,  
338

abbreviated function. *See* date  
function; time function

About text castmember, information  
on XObjects, 130, 133

abs function, 178, 342

addition (+) arithmetic operator, 68,  
329, 341

after command, 63, 345. *See*  
*also* put...after command

AIFF (Audio Interchange File  
Format) sound files  
defined, 361  
playing, 296

alert command, 178-79, 345

alert dialog box, displaying, 178-79

aliasing, defined, 361

alpha channel, and color, 150, 152-54

American Standard Code for  
Information Interchange  
(ASCII). *See* ASCII  
character set

ampersand (&) concatenation  
operator, 70, 99, 330, 341  
defined, 361



- and keyword, 54
- and logical operator, 68, 70, 179, 340
- animation, defined, 361
- animation sequence, creating, 22
- anti-aliasing, defined, 361
- Apartment, The folder, 89, 90
  - built-in XObjects, 132-38
- Apartment, The* sample movies, 89-103
  - BasicUserEvents* sample movie, 98-99
  - common script elements, 92
  - editableText* sample movie, 93-94
  - Face Kit* sample movie, 94-96
  - learning from, 90
  - Main Menu* movie, 90-92
  - rollOver* sample movie, 96-98
  - Simple Puppets* sample movie, 100-103
  - XObject examples contained in, 134-38
- AppleAudioCD XObject, 134, 136, 138
- AppleCD SC CD-ROM player, controlling with the AppleAudioCD XObject, 134, 136
- applications
  - closing, 295
  - launching, 266
- arguments
  - for communication between objects and Lingo, 128
  - defined, 361
  - for handlers, 64
  - multiple, 64
  - parameters, 64
- arithmetic operators, 68, 168
  - dictionary of, 326-33
  - quick reference summary of vocabulary and syntax, 341-42
  - See also* operators
- Array factory, 116

- arrays
  - creating and using, 111
  - defined, 361-62
  - placing values in, 111, 252
  - retrieving values from, 111, 240-41
- ASCII character set, 349-57
  - defined, 362
- asterisk (\*) (multiplication)
  - arithmetic operator, 47, 68, 328, 341
- Audio Interchange File Format (AIFF). *See* AIFF (Audio Interchange File Format)
  - sound files
- audio transitions, 6

**B**

- backColor sprite property, 179-80, 336
- background, defined, 362
- BACKSPACE character constant, 62, 181, 338
- backspace key, 181
- BasicUserEvents* sample movie, 98-99
- Basic With* demonstration movie, 21, 29-32
  - creating scripts for, 32-40
  - looking at the scripts, 30-32
  - previewing, 29-30
- Basic Without* demonstration movie, 21, 29
  - adding scripts to, 32-40
  - testing, 33
  - testing new scripts, 34-36
- beep command, 17, 181, 344
- beepOn property, 181-82, 344
- beeps, causing, 178-79, 181
- before command, 63, 345. *See also* put...before command
- bitmap, defined, 362
- Bitmap Cast Info dialog box, 10-11
- bitmap graphic castmembers, creating, 10

- boldface type, as used in this manual, 3
- books related to movie design, 359
- boolean expressions, 58
  - and comparison operators, 69, 168
  - defined, 362
- boolean values, defined, 362
- border properties, of sprites, 163
- bottom sprite property, 81, 163, 164, 182, 194, 336
- breakFilament handler, 101-2
- built-in XObjects, 132-38. *See also* XObjects
- button castmembers, creating, 10
- buttons
  - attaching scripts to, 26
  - controlling, 83-86
  - controlling the visual action of, 183
  - creating, 10, 24-25, 84
  - defined, 83, 362
  - naming, 25
  - placing, 28
  - properties, 25
  - radio, 24
  - reacting to clicked, 84
  - regular, 24-26
  - for stopping loops, 24-26
  - using, 83-86
- buttonStyle property, 183, 336
- byFrame option, of the playAccel command, 274

## C

- call, defined, 362
- callback errors, 146
- callback factories, defining, 142, 143-46
- callback handlers
  - defined, 142
  - specifying, 146
  - using, 142
- callback objects, creating, 142, 146



- callback requests (HyperCard), 141-42, 146-48
  - listing of, 147-48
- callbacks. *See* callback requests (HyperCard)
- camcorder, defined, 362
- Carriage Return, defined, 364
- case conventions, for Lingo language elements, 172, 176
- case sensitivity, 56, 172
- cast identifiers, 177
- Cast Info dialog box
  - making a text field editable, 93
  - opening the Cast Script window, 8
- cast keyword, 184, 336
- castmembers
  - button, 10
  - cast identifiers, 177
  - cast numbers and names, 58, 177, 256
  - changing the colors of, 154
  - creating, 10
  - defined, 362
  - graphic, 10
  - making editable, 41
  - managing with the virtual cast facility, 158-59
  - naming, 28, 45, 171
  - opening script windows for, 11-12
  - preloading of, 275, 276
  - quick reference summary of
    - vocabulary and syntax, 336-37
  - text field, 10
  - types of, 10
  - viewing scripts of, 8
- Cast menu, accessing Cast Scripts, 8
- cast names, 28, 177
  - as literals, 58
- cast numbers, 28, 177, 256
  - as literals, 58
- castNum sprite property, 121, 184-85, 336
- Cast Script editing window, opening, 11
- Cast Scripts, 7, 8
  - accessing, 8
  - defining factories, 171
  - defining handlers, 8, 171
  - editing window
    - illustrated, 9
    - opening, 11
  - executing, 8
  - handlers in, 63
  - opening, 32
  - scripting guidelines, 170
  - writing, 14
  - See also* scripts
- Cast Script window, opening, 8
- CAV (CAVL) (Constant Angular Velocity), defined, 362
- CD-Audio Sound example XObject, 136
- CD-ROM, defined, 363
- CD-ROM Audio example XObject, 134
- cells
  - defined, 363
  - displaying script numbers
    - controlling, 14
  - installing scripts in, 32
  - removing scripts, 19-20
  - viewing scripts of, 32
  - See also* sprite cells
- centerStage property, 162, 185, 340
- channel, defined, 363
- characters
  - defined, 363
  - deleting, 199
  - determining the number of, 228-29, 257
  - highlighting, 215
- character spaces, 55-56
  - automatic insertion of, 56
  - in text strings, 63
- character strings, testing, 57
- char...of chunk expression
  - keyword, 185-86, 345
- chars chunk function. *See* number of chars in chunk function
- chars function, 186-87, 345
- charToNum function, 187, 345
- checkBoxAccess property, 86, 187-88, 336
- check boxes
  - creating, 24, 84
  - defined, 83
  - determining the action of, 187-88
  - displaying selection status, 86, 188
  - preventing changes to, 86
  - reacting to clicked, 85
  - selecting, 85-86
- checkBoxType property, 86, 188, 336
- checkCursor command, 99
- checkField handler, 94
- checkKey handler, 42
  - turning off, 44
- checkMark menu item property, 188-89, 340
- chrominance
  - defined, 363
  - of video output, 155
- chunk expressions, 259, 283, 284
  - specifying characters in, 185-86, 257
  - specifying items in, 223-24, 257-58
  - specifying lines in, 229, 258
- chunks, defined, 363
- Chunk Size, defined, 363
- cleanUp handler, 116, 122-23
- Clear command (Edit menu), using in the script editor, 16
- clickOn function, 189, 341
- click option, of the playAccel command, 274
- clickStop option, of the playAccel command, 274
- closeDA command, 190, 339



- `closeResFile` command, 190, 339
- `closeXlib` command, 190-91, 339
  - closing XCMDs and XFCNs, 141
  - closing XLibraries, 130
- CLUT (color lookup table), 151-52, 364
- CLV (Constant Linear Velocity), defined, 363
- code
  - automatic formatting, 16
  - compiled, 364
  - formatting long lines, 16
  - as represented in this manual, 3
- code structures, quick reference
  - summary of vocabulary and syntax, 337
- color
  - active palette, 154
  - and the alpha channel, 150, 152-54
  - background
    - of movies, 301
    - of sprites, 179-80
  - color lookup table (CLUT), 151-52
  - color palette, 151
  - custom palettes, 152
  - cycling, 363
  - depth, 150, 191, 363
  - direct color schemes, 151
  - foreground, of sprites, 209-10
  - indexed, 151-52
  - key color, 368
  - managing, 150-54
  - NTSC palette, 152
  - Palette channel, 154
  - pixels, 150
  - 16-bit display mode, 151
  - System Palette, 152
  - 32-bit display mode, 151
  - 32-bit QuickDraw, 150, 192
- color cycling, defined, 363
- color depth, 150, 191
  - defined, 363
- `colorDepth` property, 191, 344
- color lookup table (CLUT), 151-52, 364

- color monitors, using Director with, 7, 150-54, 157
- Color Picker (Macintosh), 150
- `colorQD` function, 192, 344
- color saturation, and video output, 157
- Command-. (period), stopping endless loops, 73, 347
- Command-A, starting movies, 30, 347
- Command-D, step back one frame, 347
- Command-Delete, clearing text, 14, 348
- Command-Down Arrow
  - moving to bottom of Message Window, 348
  - moving the insertion point, 13, 347
- `commandDown` function, 192, 339
- Command-F, step forward one frame, 347
- Command-G, Find Again command, 19, 348
- Command-H, Find command, 18, 348
- command key shortcuts, adding to menu commands, 49
- Command-Left Arrow, go to previous marker, 347
- Command-M, displaying the Message window, 12, 347
- Command-R, rewinding movies, 28, 347
- Command-Right Arrow, go to next marker, 347
- commands
  - abbreviated, 55
  - case conventions, 172
  - components of, 54
  - defined, 364
  - described, 166
  - getting descriptions of, 16
  - inserting at the cursor location, 15
  - Lingo, dictionary of, 175-326
  - See also* individual commands by name

- Command-Shift-U, accessing Movie Scripts, 8, 347
- Command-T, Replace And Find command, 19, 348
- Command-Up Arrow
  - moving the insertion point, 13, 347
  - moving to top of Message window, 348
- Command-W
  - stopping loops, 24
  - stopping playback, 23, 347
- Command-Y, Find In Next Script command, 19, 348
- comments
  - for control structures, 173
  - creating out of statements, 90
  - dedicated, 173
  - defined, 364
  - for handlers, 173
  - in-line, 173
  - in scripts, 56
  - symbol indicating, 51, 56, 327
  - uses of, 172-73
- comparison operators, 68, 69-70, 168
  - quick reference summary of vocabulary and syntax, 341-42
  - See also* operators
- compiled code, defined, 364
- component analog format, and video output, 155
- component digital format, and video output, 155
- component video, defined, 364
- composite analog format, and video output, 155
- composite digital format, and video output, 155
- composite video, defined, 364
- concatenation, defined, 364
- concatenation (&) text operator, 70, 99, 330, 341
- concatenation with space (&&) text operator, 56, 70, 99, 330, 341



- conditional expressions, 71-72
  - connectDot handler, 97
  - Constant Angular Velocity (CAV).  
See CAV (CAVL)
  - Constant Linear Velocity (CLV). *See*  
CLV (Constant Linear Velocity)
  - constants, 62
    - case conventions, 172
    - defined, 364
    - described, 167
    - Lingo, dictionary of, 175-326
    - quick reference summary of  
vocabulary and syntax, 338
    - See also* individual constants by  
name
  - constrainFace handler, 96
  - constrainH function, 192-93, 336
  - constraint property, constraining  
the position of sprites, 96,  
164, 193-94, 336
  - constrainV function, 194, 336
  - container, defined, 364
  - contains comparison operator, 69-  
70, 195, 340, 345
  - continuation symbol (→), 4, 29, 327-  
28, 337
  - continue command, 33, 36-37,  
195, 342
    - in event scripts, 66-67
  - Control-click, opening and  
displaying Cast Info dialog  
box, 348
  - controlDown function, 195-96, 339
  - Control-Option-Click, opening and  
displaying Cast Scripts, 348
  - Control Panel, Loop button, 22, 23
  - control structures, 70-79
    - comments for, 173
    - defined, 364
    - indentation, 172
    - using blank lines with, 172
  - conventions
    - used in the Lingo dictionary, 176
    - used in the quick reference  
summary, 336
    - used in this manual, 3-4
    - for writing Lingo scripts, 56
  - coordinates
    - converting differences into  
distances, 178
    - display coordinates, 160
    - mouse coordinates, 161
    - mouse pointer position, 162-63
    - screen coordinate system,  
illustrated, 161
    - sprite border properties, 163
    - sprite coordinates, 161
    - sprite positions, 163-64
    - Stage coordinates, 160-64
    - Stage coordinate system,  
illustrated, 162
    - working with, 160-64
  - Copy command (Edit menu), using  
in the script editor, 16
  - Courier font
    - as used in the Lingo dictionary, 176
    - as used in this manual, 3
  - C programming language
    - defined, 362
    - and writing XObjects, 128, 168
  - CR (Carriage Return), defined, 364
  - current frame, defined, 364
  - cursor command, 196-97, 341
  - cursor locations, inserting  
commands at, 15
  - cursor sprite property, 197, 341
  - cursor status, checking, 99
  - custom menus, 6
    - creating, 47-49, 79
    - creating menu items, 48
    - defining content of, 237-38
    - installing, 48
    - naming, 48, 237-38
    - removing, 79
    - special symbols in, 238
    - See also* menus
  - Cut command (Edit menu), using in  
the script editor, 16
- D**
- data fork, defined, 364
  - date formats, 198
  - date function, 198, 346
  - debugging
    - movies, 12, 49-51
    - using the put command, 62
  - decimal numbers
    - for additional ASCII characters,  
356-57
    - for the Macintosh character set,  
349-55
  - delay command, 198-99, 346
    - in event scripts, 66-67
    - and repeat while loops, 67
    - and when... then tests, 67
  - delays, specifying during a repeat  
while sequence, 77
  - delete command, 199, 345
  - deltaH local variable, 102
  - deltaV local variable, 102
  - demonstration movies, 21, 29-32,  
89-103
    - creating loops for, 22-24
  - deselect, defined, 365
  - desk accessories
    - closing, 190
    - opening, 267
  - dialog, defined, 365
  - dictionary of Lingo language  
elements, 175-333
    - typographic conventions, 176
  - digital video, defined, 365
  - Digital Video Interface (DVI). *See*  
DVI (Digital Video Interface)
  - digitize, defined, 365
  - direct color schemes, 151. *See also*  
color



## Director

- exiting from, 285
  - new Lingo features, 6
  - scripting environment, 9-14
  - setting up on the Macintosh, 7
  - and stored XObjects, 129
  - Studio functions, 1
  - system requirements, 2
  - term defined, 3
  - version number, 318
- display coordinates, 160. *See also* coordinates
- dithering, defined, 365
- division (/) arithmetic operator, 68, 328, 342
- do command, 199, 345
- done command. *See* play done command
- dontPassEvent command, stopping event messages, 66, 75-76, 200-201, 339
- doSomething handler, and cast scripts, 170
- dotNumber argument, 97
- double ampersand (&&), concatenation with space operator, 56, 70, 99, 330, 341
- doubleClick function, 201, 341
- double equal signs (==), indicating script location, 51
- double hyphens (-), indicating a comment, 51, 56, 327, 337
- double quotation marks ("), for text string literals, 57, 99
- drivers
  - defined, 365
  - of XObjects, 126
- dropFrame argument, 211, 216
- duplicate names, in the message hierarchy, 87
- DVI (Digital Video Interface), defined, 365

## E

- editableText command, 81, 93-94, 95, 201, 345
- Editable Text command (Cast Info dialog box), 93, 201
- editable text fields, 41-42, 93-94
  - creating, 26-27
  - See also* text
- editableText sample movie, 93-94
- editable text sprites, 27, 81, 93-94
- Edit menu, using commands in the script editor, 16
- else keyword. *See* if...then keyword
- empty, defined, 365
- EMPTY character constant, 202, 338
- empty script, defined, 365
- enabled menu item property, 202-3, 340
- end if statement, 72
- end keyword, 54, 203
  - defining handlers, 43, 64
- end repeat statement, 204
- ENTER character constant, 62, 203, 338
- Enter key
  - closing the script editor, 16
  - viewing script numbers, 34
- equal to (=) comparison operator, 70, 332, 342
- Error dialog box, 49
- error messages
  - for invalid statements, 13
  - for misspelled words, 24
- evaluate, defined, 365
- eventActionsOff handler, 99
- eventActionsOn handler, 98-99
- event handlers, quick reference summary of vocabulary and syntax, 338

## events

- defined, 365
  - quick reference summary of vocabulary and syntax, 339
- event scripts, 7-8, 9, 66-67
  - adding, 42
  - defined, 365
  - defining, 9
  - described, 66
  - overriding, 75-76
  - resetting, 44
  - reviewing, 31
  - turning off, 75, 78
  - and when...then keywords, 74-76
  - See also* scripts
- event states
  - setting to false, 78
  - turning off, 78
- execute, defined, 365
- exit keyword, 123, 203-4, 337
- exitLock property, 204-5, 340
- exit repeat keyword, 73, 204, 337
- expressions
  - calculating the value of, 62
  - chunk, 185-86, 223-24, 229, 256-58, 259, 283, 284
  - conditional, 71-72
  - defined, 54, 366
  - evaluating, 199
  - evaluating logical, 217-18
  - joining two, 56
  - using handlers in, 65
- external device, defined, 366
- external objects. *See* objects; XObjects

## F

- Face Kit sample movie, 94-96
- factor, defined, 366
- factories, 105-12
  - callback, defining, 142, 143-46
  - calling methods by objects, 112
  - creating and managing arrays, 107



## Factories (continued)

- creating objects from, 50, 110, 117-18, 243-44
  - creating and using object arrays, 111
  - defined, 366
  - defining, 108-10, 171, 205-6
  - described, 106-7, 205
  - instance variables, 109-10
  - naming, 108
  - objects and messages, 107-8
  - predefined methods, 168
  - quick reference summary of
    - vocabulary and syntax, 337
  - special methods in, 111-12
- Factories folder, 113
- factory function, 206
- factory keyword, defining
  - factories, 108, 205-6, 337
- fadeIn command. *See* sound
  - fadeIn command
- fadeOut command. *See* sound
  - fadeOut command
- FALSE logical constant, 58, 207, 338
- Feature Examples movie, 2
- field keyword, 207, 336, 345
- fields, naming, 45
- File I/O example XObject, 134
- fileIO XObject, 129, 134, 135
  - recording movies to videotape, 137
- files, defined, 366
- Find Again command (Script menu), 19
- Find command (Script menu), 18
- Find dialog box, illustrated, 18
- Find In Next Script command (Script menu), 19
- Find operations, 18
- firstCharacter expression, 186-87
- fixStageSize property, 162, 207-8, 340
- flash HyperTalk XCMD, 141
- floating point numbers, 58, 208-9
- floatP function, 208, 241, 340
- floatPrecision property, 209, 344
- fonts
  - additional ASCII characters, 356-57
  - Macintosh character set, 349-55
  - styles, 310
  - of text castmembers, 308
- font sizes, and video output, 157, 309
- foreColor sprite property, 209-10, 336
- formatting, defined, 366
- FPS (frames per second), defined, 366
- fractionalSeconds argument, 211, 216
- frame accurate, defined, 366
- frame command. *See* go
  - command; play
  - command
- frame function, 210, 342
- frame grabbing, defined, 366
- frame label, defined, 366
- frame markers
  - creating, 28
  - scripting guidelines, 171
  - using in loops, 27-28
- frame numbers, and label names, 27
- FramePerFrame example XObject, 136-37
- frame rate. *See* FPS (frames per second)
- frames
  - converting to hours-minutes-seconds, 210-11
  - converting from hours-minutes-seconds, 215-16
  - current, 364
  - defined, 366
  - frame numbers, 27
  - jumping to, 6, 23
  - labeling, 27
  - transitions between, 280-82
- frames per second (FPS), defined, 366

- framesToHMS function, 210-11, 346
- freeBlock function, 211, 344
- freeBytes function, 212, 344
- function call, defined, 366
- function handlers, 64, 65
  - defined, 367
  - See also* handlers
- functions, 64, 67-68
  - case conventions, 172
  - defined, 367
  - described, 166
  - Lingo, dictionary of, 175-326
  - properties, 67
  - quick reference summary of
    - vocabulary and syntax, 340
  - See also* handlers; individual functions by name

## G

- genlock/genlocking, defined, 367
- global keyword, 212-13, 337
  - establishing global variables, 61
- global variables, 60, 61, 109, 212-13
  - defined, 367
  - displaying, 293
  - displaying current, 61
  - See also* variables
- global variables, 116
- go command, 45, 55, 213-14, 342
  - opening Director documents, 91-92
  - See also* go to command
- go to command, 23, 45-46, 213-14, 342
  - abbreviated form, 55
  - entering the script for, 45-46
  - and the play command, 39
  - variations with, 39-40
  - See also* go command
- graphic castmembers, creating, 10
- graphic images, and the alpha channel, 152-54
- graphic objects, dragging, 6
- graphic transitions, 6, 280-82



- greater than (>) comparison operator, 69-70, 331, 342
- greater than (>) pointer, nesting level of the script, 51
- greater than or equal to (>=) comparison operator, 70, 332, 342
- grouping operator (parentheses) (), 326, 341

## H

- handlers, 63-65
  - arguments, 64
  - calling custom, 116
  - comments for, 173
  - copying pre-written, 43-44
  - creating, 43-47
  - defined, 43, 63, 262, 367
  - defining, 63-65, 107, 171, 262
    - in Cast Scripts, 63
    - in Movie Scripts, 8, 63
  - global variables, 60, 61, 109, 212-13
  - grouping related, 63
  - Handlers submenu, 15
  - indentation, 172
  - initializing variables, 59
  - local variables, 60, 61, 109
  - naming, 64, 107, 173
  - parameters, 64
  - as parts of expressions, 65
  - performing multiple operations, 74-75
  - quick reference summary of vocabulary and syntax, 338
  - testing, 50
  - using, 43-47
  - using blank lines with, 172
  - See also* individual handlers by name
- Handlers submenu, of the Lingo menu, 15
- height sprite property, 214, 336
- Help pointer, 16
- Help window, opening, 16
- hexadecimal numbers

- for additional ASCII characters, 356-57
- for the Macintosh character set, 349-55
- Hi-8, defined, 367
- hilite command, 215, 345
- hilite property, of buttons, 85, 215, 336
- h local variable, 102
- HMStoFrames function, 215-16, 346
- hours-minutes-seconds (HMS)
  - converting from frames, 210-11
  - converting to frames, 215-16
- howFar handler, 102, 120
- hue, defined, 367
- HyperCard callback requests, listing of, 147-48. *See also* callback requests (HyperCard)
- HyperCard Message box, and the Message Window, 13
- HyperTalk
  - notes in this manual for users of, 4
  - XCMD resources, 126-27
  - XFCN resources, 126-27
  - and XObjects, 126-27

## I

- idle handler, in Movie scripts, 169, 263. *See also* on idle movie handler
- if keyword, 28, 54, 71-72, 172, 218, 337
- if...then...else keyword, 71-72, 99, 337
- if...then keywords, 44, 217-18, 337
- immediate sprite property, 218, 336
  - turning off, 78
- in. *See* the number of chars in chunk function; the number of items in chunk function; the number of lines in chunk function; the number of words in chunk function
- indentation, automatic, 172, 173

- indexed color, 151-52. *See also* color
- init handler, using built-in XObjects, 133
- initialize, defined, 367
- initialize handler, 116, 117
- ink effects, of sprites, 219-20
- ink sprite property, 219-20, 336
- input, testing, 42
- insertion point, moving in the Message window, 13
- installMenu command, 220-21, 340
  - creating custom menus, 47-49, 79
  - removing custom menus, 79
- instance keyword, defining
  - instance variables, 109-10, 221-22, 337
- instance tests, 71
- instance variables, 221-22
  - defined, 367
  - defining, 243-44
  - in factories, 109-10
  - initializing, 118
  - See also* variables
- integer function, 222, 342
- integerP function, 222-23, 241, 340
- integers
  - defined, 367
  - as literals, 57
- interactive, defined, 367
- Interactive Color Examples disk, XObject example documents, 133
- interactive movies, 6. *See also* movies
- Interactive Tutorials folder, demonstration movies, 21, 29
- interface, defined, 368
- internal objects. *See* factories; objects
- intersects comparison operator. *See* sprite...intersects comparison operator
- into command. *See* put...into command



*italic type*  
 as used in the Lingo dictionary, 176  
 as used in this manual, 64, 109

`itemName` command, naming items  
 in custom menus, 79

`item...of` chunk expression  
 keyword, 223-24, 345

items  
 deleting, 199  
 highlighting, 215  
 specifying in chunk expressions,  
 223-24, 257-58

items chunk function. *See* the  
 number of items in  
 chunk function

iteration, defined, 368

## J

Joint Photographic Experts Group  
 (JPEG). *See* JPEG (Joint  
 Photographic Experts  
 Group)

JPEG (Joint Photographic Experts  
 Group), defined, 368

## K

keyboard, quick reference summary  
 of vocabulary and syntax,  
 339

keyboard shortcuts, quick reference  
 summary of vocabulary  
 and syntax, 347-48

`keyCode` function, 225, 339

key color, defined, 368

`keyDown` events, 339  
 and the `dontPassEvent`  
 command, 200  
*See also* when `keyDown` then  
 command

`keyDown` event script, 98-99  
 testing keystrokes, 42, 74

`keyDownScript` property, 225-26, 339

`key` function, 224, 339

keying, defined, 368

keystrokes  
 for the ASCII character set, 349-57  
 for the Macintosh character set,  
 349-55  
 testing, 42

keywords  
 case conventions, 172  
 defined, 368  
 described, 166  
 Lingo, dictionary of, 175-326  
 multiline form, 54  
 optional, 55  
*See also* words; individual  
 keywords by name

kilobyte (KB), 211

## L

`label` function, 226, 342

`labelList` function, 226, 342

labels  
 adding to sequences, 42  
 creating, 28  
 defined, 368  
 scripting guidelines, 171  
 using in loops, 27-28

LANC (Local Application Control  
 Bus), defined, 368

language elements, Lingo  
 access to, 11  
 dictionary of, 175-333  
 inserting at the cursor location, 15  
 quick reference summary of,  
 335-48

`lastCharacter` expression, 186-87

`lastClick` function, 227, 341  
 resetting the timer for, 76

`lastEvent` function, 227, 339  
 resetting the timer for, 76-77

`lastKey` function, 227, 339  
 resetting the timer for, 76

`lastRoll` function, 227-28, 341  
 resetting the timer for, 76

`left` sprite property, 81, 163, 164,  
 193, 228, 336

`length` function, 228-29, 345

less than (<) comparison operator,  
 70, 331, 342

less than or equal to (<=) comparison  
 operator, 70, 331, 342

line breaks, in scripts, 29

`line...of` chunk expression  
 keyword, 229, 345

lines  
 deleting, 199  
 highlighting, 215  
 specifying in chunk expressions,  
 229, 258

lines chunk function. *See* the  
 number of lines in  
 chunk function

`lineSize` sprite property, 230, 336

line spacing, of text castmembers, 309

Lingo  
 basics, 4-20  
 described, 1  
 dictionary of language elements,  
 175-333  
 guidelines for script placement,  
 166-73  
 introduction to, 6-7  
 message hierarchy, 87  
 multiline statements, 4  
 new features, 6  
 quick reference summary of  
 vocabulary and syntax,  
 335-48  
 representation of syntax in this  
 manual, 3  
 scripting environment, 9-14  
 scripts  
   types, 7-9  
   uses of, 6  
 statements, 7  
 suggested additional reading, 359  
 summary of major word groups,  
 166-73  
 syntax, 7  
 term defined, 3  
 working with, 21-51  
*See also* scripts; statements; syntax

Lingo menu, 9, 15-16



- access to Lingo language elements, 11
- Handlers submenu, 15
- illustrated, 15
- using with the script editor, 15-16, 17
- Lingo Tutorials & Apartment disk, 90
- XObject example documents, 133
- literals, 57-58
  - cast numbers and names, 58
  - character spaces in, 56
  - defined, 57, 368
  - floating point numbers, 58
  - integers, 57
  - quoted, 371
  - text strings, 57
- 1 local variable, 102
- Local Application Control Bus (LANC). *See* LANC (Local Application Control Bus)
- local variables, 60, 61, 109
  - defined, 369
  - displaying, 294
  - See also* variables
- locHandler handler, establishing to the location of sprites, 82
- locH sprite property, 81, 102, 120, 164, 193, 230, 336
- lockCast command, and the preLoad command, 158
- locV sprite property, 81, 102, 164, 193, 231, 336
- logical operation, defined, 369
- logical operators, 68, 70, 168
  - quick reference summary of vocabulary and syntax, 340
- long. *See* date function; time function
- Loop button, repeating movies, 22
- loop movies, creating, 22-24. *See also* loops
- loop option, of the playAccel command, 274

- loops, 21, 22-29
  - controlling execution, 24-27
  - controlling with text field input, 26-27
  - creating, 22-24
    - with Score scripts, 23-24
  - defined, 22, 369
  - moving between, 24
  - quick reference summary of vocabulary and syntax, 337
  - stopping, 24
  - testing text input, 28-29
  - using frame markers in, 27-28
  - using labels in, 27-28
- Loops demonstration movie, 21
- lowercase letters, in scripts, 56, 172, 176
- luminance
  - defined, 369
  - of video output, 155

## M

- mAccelerate method, 121
- machineType function, 231-32, 344
- Macintosh
  - determining model currently in use, 232
  - restarting, 287
  - setting up Director, 7
  - shutting down, 295
  - system requirements, 2
  - using a color monitor, 7
- Macintosh character set, 349-55
  - additional ASCII characters, 356-57
- Macintosh Finder, quitting, 204-5
- Macintosh system, changing properties of, 62, 67
- Macintosh System 7, and the virtual cast facility, 158
- Macintosh Toolbox, and XObjects, 126
- macro keyword, 232-33

## MacroMind Director 3.0 Interactivity Manual

- conventions used, 3-4
- how to use, 2-3
- notes for HyperTalk users, 4
- purpose of, 1
- tips & hints, 4
- warnings, 4

MacroMind Director movies. *See* movies

## MacroMind Director Studio Manual, 1, 5

- creating an animation sequence, 22
- creating castmembers, 10
- setting the size and position of the stage, 160

MacroMind Director version 3.0. *See* Director

MacroMind Player
 

- quick reference summary of vocabulary and syntax, 340
- and stored XObjects, 129

macros
 

- defining, 232-33
- described, 232-33
- scripting guidelines, 171

magazines related to movie design, 359

•Main Menu movie, 90-92

managing color, 150-54

marker channel, defined, 369

marker function, 97-98, 233, 342

markers
 

- defined, 369
- frame number of, 233

marker well, defined, 369

MASS Microsystems ColorSpaceFX board, 137-38

mAtFrame special message, 234, 343
 

- in factories, 110
- recording movies to videotape, 137



- math functions, quick reference
  - summary of vocabulary and syntax, 341-42. *See also* arithmetic operators
- maxSprites variable, 116, 117
- mBounce method, and XObjects, 128
- mClick method, 121
- mDescribe predefined method, 234-35, 343
  - viewing an XObject's methods, 130 and XObjects, 127, 295
- mDispose predefined method, 235-36, 343
  - deleting objects from memory, 111-12, 116, 123
  - removing XObjects from memory, 131, 132
- megabyte (MB), 211
- me keyword, 236, 337
  - for calling methods by factory objects, 112
- memory
  - automatic management of, 6
  - deleting objects from, 111-12
  - free, 211, 212
  - freeing up, 235-36
  - managing with the virtual cast facility, 158-59
  - total amount allocated to the program, 237
- memorySize function, 237, 344
- menu. *See* checkMark menu item property; enabled menu item property; name menu item property; name menu property; number menu item property; script menu item property
- menu: keyword, 237-38, 340
  - naming menus, 48, 79
  - using special symbols in menus, 49, 238
- menu commands, adding command key shortcuts, 49, 79
- menuItem. *See* checkMark menu item property; enabled menu item property; name menu item property; script menu item property
- menuItems. *See* number menu item property
- menus
  - adding command key shortcuts, 49, 79
  - adding new, 48, 79, 220-21
  - creating menu items, 48, 79, 237-38
  - custom, 6
    - creating, 47-49, 79
    - defining content of, 237-38
    - installing, 48, 220-21
    - removing, 79, 220-21
  - displaying enabled status of menu items, 202-3
  - displaying menu items with checkmarks, 188-89
  - installing, 31, 79, 220-21
  - naming, 48, 79, 237-38
  - pull-down, 47-49
  - quick reference summary of vocabulary and syntax, 340-41
  - using special symbols, 49, 238
- menus. *See* number menu property
- message hierarchy
  - duplicate names in, 87 of Lingo, 87
- messageName placeholder, specifying methods, 168
- messages
  - defined, 369
  - quick reference summary of vocabulary and syntax, 338
  - special, 168
    - quick reference summary of vocabulary and syntax, 339
- Message window, 12-14
  - clearing text from, 14
  - defined, 369
  - defining global variables, 60
  - displaying, 12
  - displaying global variables, 61, 293
  - displaying local variables, 61, 294
  - displaying the value of an expression, 62
  - executing script statements, 13
  - and the HyperCard Message box, 13
  - illustrated, 12
  - moving around in, 13
  - pasting commands to, 15
  - testing statements, 13, 50
  - tracing scripts, 12, 49-51
  - tracing symbols, 50-51
  - use of parentheses, 55
  - uses of, 12
  - viewing XObjects, 129
- method keyword, 239-40, 337
  - creating method definitions, 108-9
- methods
  - called by objects, 112
  - case conventions for names, 172
  - creating lists of, 234-35
  - defining, 107, 108-9, 239-40
  - described, 168
  - naming, 173
  - not requiring arguments, 128
  - predefined, 168
    - quick reference summary of vocabulary and syntax, 343
  - quick reference summary of vocabulary and syntax, 337
  - specifying, 168
  - of XObjects, 127
  - viewing, 130
  - See also* individual methods by name
- mGet predefined method, retrieving values from arrays, 111, 116, 119, 123, 135, 240-41, 343



- MIDI (Musical Instrument Digital Interface)
  - controlling devices with scripts, 278-79
  - defined, 369
- mInstanceRespondsTo predefined method, 241, 343
- minus sign (-), subtraction
  - arithmetic operator, 68, 329, 341
- NSMutableArray predefined method, 242, 343
- mName predefined method, and
  - XObjects, 131, 242-43, 343
- mNew predefined method, 243-44, 343
  - creating new instances of XObjects, 131
  - creating objects from factories, 110, 116, 117-18, 240
  - defining instance variables, 110
  - destroying objects created with, 235-36
  - and XObjects, 128, 130
- mNextFrame method, 119, 120
- mod arithmetic operator, 68, 244, 342
- modulo, defined, 369
- modulo (mod) arithmetic operator.
  - See mod arithmetic operator
- monitoring time, 76-78
- monitors
  - color, determining the color depth, 191
  - and Stage coordinates, 160-61
  - and video output, 157
- Motion Picture Experts Group (MPEG). See MPEG (Motion Picture Experts Group)
- mouse
  - pointer position, 162-63, 247, 249-50
  - quick reference summary of vocabulary and syntax, 341
  - tracking movement, 178
- mouseCast function, 245, 341
- mouseChar function, 245, 341
- mouse coordinates, 161. *See also* coordinates
- mouseDown function, 246, 341
- mouseDown handler, 101-2
  - in Cast Scripts, 263
  - and the dontPassEvent command, 200
  - effect on buttons, 83
  - and event scripts, 74-76, 98-99
  - turning off, 78
  - See also on mouseDown event handler; when mouseDown then command
- mouseDownScript property, 246-47, 341
- mouseH function, mouse pointer position, 162-63, 247, 341
- mouseItem function, 247-48, 341
- mouseLine function, 248, 341
- mouseOption handler, and event scripts, 75-76
- mouse pointer, quick reference
  - summary of vocabulary and syntax, 341
- mouse pointer position, 162-63, 247, 249-50
- mouseUp function, 248, 341
- mouseUp handler, 26, 94, 115-17
  - and Cast Scripts, 264
  - changing the castmember displayed, 100-101
  - effect on buttons, 83, 122
  - and event scripts, 74-76, 98-99
  - saving, 35
  - script statement for, 28
  - and sprite scripts, 170
  - turning off, 78
  - See also on mouseUp event handler; when mouseUp then command
- mouseUpScript property, 249, 341
- mouseV function, mouse pointer position, 162-63, 341
- mouseWord function, 250, 341
- moveableSprite command, 80-81, 95, 96, 250, 336
- moveSprite handler, 120
- movie. *See* go command; play command
- movie files, closing, 12
- movie function, 251, 342
- Movie Info dialog box
  - accessing Movie Scripts, 8
  - opening, 11
  - opening Movie Scripts, 31
- movies
  - color, 150-54
  - controlling the action with loops, 24
  - creating animation sequences, 22
  - debugging, 12, 49-51
  - demonstration movies, 21, 89-103
  - expanding the functionality of, 1
  - interactive, 6
  - movement from frame to frame, 119-20
  - Movie Scripts, 8
  - playing, 22-24, 33, 38-40
  - playing several from a single script, 40
  - quick reference summary of vocabulary and syntax, 342-43
  - repeating with the Loop button, 22-23
  - rewinding, 28
  - sample, 89-103
  - stopping playback, 23, 33, 36-37
  - suggested additional reading, 359
  - troubleshooting, 12, 49-51
- Movie Script editing window
  - illustrated, 9
  - opening, 11
- Movie Scripts, 7, 8
  - accessing, 8
  - calling custom handlers, 116
  - defined, 369
  - defining factories, 171
  - defining handlers, 8, 171



## Movie Scripts (*continued*)

- editing window
  - illustrated, 9
  - opening, 11
- handlers in, 63, 119
- opening, 31
- scripting guidelines, 169
- writing, 14
- See also* scripts
- MPEG (Motion Picture Experts Group), 370
- mPerform predefined method, 251-52, 343
- mPut predefined method, placing values in arrays, 111, 116, 135, 252, 343
- mRelease method, 123
- mRespondsTo predefined method, 253, 343
- multiline statements, 4
  - formatting, 54
  - See also* statements
- multimedia, defined, 370
- Multiple Devices example XObject, 138
- multiplication (\*) arithmetic operator, 47, 68, 328, 341
- Musical Instrument Digital Interface (MIDI). *See* MIDI (Musical Instrument Digital Interface)
- mySpeed instance variable, 118
- mySprite instance variable, 118

## N

- name cast property, 253-54, 336
- name menu item property, 254-55, 340
- name menu property, 254, 340
- naming conventions, 172-73
- negation (-), arithmetic operator, 68, 327, 341
- nesting level, of scripts, 51

## newsletter related to movie design, 359

- noFlush option, of the playAccel command, 274
- noSound option, of the playAccel command, 274
- not equal to (<>) comparison operator, 70, 332, 342
- nothing command, 255-56, 337, 339
- nothing value, 60
- not logical operator, 68, 70, 255, 340
- noUpdate option, of the playAccel command, 274
- NTSC palette, for NTSC video, 152
- NuBus cards, and XObjects, 126
- null string, defined, 370
- number of castmembers property, 256-57, 336
- number cast property, 256, 336
- number of chars in chunk function, 257, 345
- number of items in chunk function, 257-58, 345
- number of lines in chunk function, 258, 345
- number menu item property, 258-59, 340
- number menu property, 259, 340
- number of words in chunk function, 259-60, 345
- numeric, defined, 370
- numToChar function, 260, 345

## O

- object arrays. *See* arrays
- objectCount variable, 116, 117-18
- objectName variable, 131
- object oriented programming (OOP), defined, 370
- objectP function, 241, 261, 340

## objects

- calling methods, 112
- communicating with scripts, 107-8
- created by factories, 239
- creating from factories, 50, 110, 117-18, 243-44
- defining in factories, 105, 107-8
- deleting from memory, 111-12, 131
- destroying, 235-36
- external, 125-38, 239
- instances of, 105, 106
- instance variables of, 109-10
- internal, 239
- manipulating, 121
- messages of, 107-8
- properties of, 67
  - changing, 62
  - types of, 239
- See also* XObjects
- offset function, 261-62, 345
- of word, 261
- on checkKey handler, naming castmembers, 45
- on command, 54
  - defining handlers, 43, 64, 107
- on handlerName definition line, 173
- on idle movie handler, 263, 338
- on keyword, 262-63, 337, 338
- on mouseDown event handler, 263-64, 338
  - and cast scripts, 170
  - See also* mouseDown handler
- on mouseUp event handler, 264, 338
  - and cast scripts, 170
  - See also* mouseUp handler
- on startMovie movie handler, 264-65, 338
- on stepMovie movie handler, 265, 338. *See also* stepMovie handler
- on stopMovie movie handler, 266, 338. *See also* stopMovie handler



- open command, 266, 339
- openDA command, 267, 339
- openResFile command, 267, 339
- openXlib command, 268, 339
  - opening XCMDs and XFCNs, 140
  - opening XObjects, 130, 131
  - and XObjects, 127, 129
- operand, defined, 370
- operators, 68-70
  - arithmetic, 68, 168
  - comparison, 68, 69-70, 168
  - defined, 370
  - described, 168
  - Lingo, dictionary of, 175-333
  - logical, 68, 70, 168
  - order of precedence, 68-70, 168
  - quick reference summary of
    - vocabulary and syntax, 341-42
  - string, 70
  - text, 168
  - See also* individual operators by name
- optional elements, in scripts, 64-65
- optional keywords, in scripts, 55
- optionDown function, 268-69, 339
- Option-L (continuation symbol), 4, 29
- Option-Return character (↵) special
  - symbol, continuation
  - symbol, 4, 29, 327-28
- Option-X, creating menu items, 48, 79, 238
- order of precedence, of operators, 68-70, 168
- or logical operator, 68, 70, 269, 340
- output, quick reference summary of
  - vocabulary and syntax, 342.
  - See also* video output

## P

- Paint window, creating graphic
  - castmembers, 10
- palette, defined, 370

- Palette channel, 8, 154
  - as a puppet, 277-78
- Panel XObject, 129, 135
- parameters
  - of arguments, 64
  - defined, 370
  - of handlers, 64
  - required with commands, 15
- parentheses
  - as arithmetic operators, 68
  - calling functions from other
    - scripts, 65
  - as grouping operator, 326, 341
  - in the Message window, 55
  - optional and required, 55
  - overriding precedence with, 55, 68
- Pascal programming language
  - defined, 370
  - and writing XObjects, 128, 168
- pass, defined, 370
- Paste command (Edit menu), using in
  - the script editor, 16
- pathName function, 269, 339, 342
- pattern, defined, 371
- pause command, 33, 36-37, 270, 342
  - in event scripts, 66-67
- pauseState function, 67, 270, 342
- perFrameHook property, 271-72, 339, 342
  - in factories, 110, 272
  - and the mAtFrame special
    - message, 234, 271-72
  - recording movies to videotape, 137
  - and the stepMovie handler, 171, 265
  - turning off, 78
- PICT graphic file format, defined, 371
- picture cast property, 272, 336
- PioneerLaserdisk XObject, 137
- pixel, defined, 371
- pixel point sizes, and video output, 156
- placeholder names, showing
  - required parameters, 15

- placeholders
  - examples of, 54
  - placeholder names, 15
  - uses of, 17
- playAccel command, 274-75, 343
  - options available, 274
- playback head
  - controlling, 33, 38-40, 45, 213-14, 272-74
  - defined, 371
  - stopping from advancing, 67, 270
  - watching, 22, 23
- play command, 33, 38-40, 95, 272-73, 342, 343
  - and the go to command, 39
  - starting secondary documents, 91, 92
  - when to use, 40
- Play command (Control menu),
  - starting movies, 30
- play done command, 33, 38-40, 95, 273-74, 343
  - returning to the main movie, 91-92
- playFile. *See* sound playFile
  - command
- playRect option, of the playAccel
  - command, 274
- plus sign (+)
  - addition arithmetic operator, 68, 329, 341
  - indicating castmembers which
    - have Cast Scripts, 14, 20
- pointer. *See* mouse pointer
- PopupMenu XObject, 136
- PopUp Menu example XObject, 135
- pound sign (#) symbol definition
  - operator, indicating
  - symbols, 62, 333
- precedence order
  - defined, 371
  - of operators, 68-70, 168
  - overriding with parentheses, 55, 68



- predefined methods, 168
  - quick reference summary of vocabulary and syntax, 343
  - See also* methods
- preLoadCast command, 276, 343
  - and the virtual cast facility, 159
- preLoad command, 275, 343
  - and the virtual cast facility, 158-59
- presentations, controlling the sequence, 6
- printFrom command, 276-77, 342
- printing, at reduced size, 277
- properties
  - of buttons, 25
  - defined, 371
  - described, 167
  - Lingo, dictionary of, 175-326
  - of the Macintosh system, 62, 67
  - of objects, 62, 67
  - of puppets, 82-83
  - of sprites, 81
    - controlling, 80
  - super-global, 167
  - See also* individual properties by name
- prototyping, 6
- pull-down menus, creating, 47-49
- puppetPalette command, 83, 277-78, 343
- puppets
  - declaring sprites to be, 81-82, 279-80
  - defined, 80, 371
  - properties of, 82-83
  - puppet sounds, 103
  - puppet sprites, 100-103, 279-80
  - quick reference summary of vocabulary and syntax, 343-44
  - sample movie illustrating, 100-103
  - transitions between frames, 280-82
  - turning off, 78, 82
  - using, 81-82
- puppetSound command, 83, 103, 278-79, 343, 344

- puppet sounds, 103
- puppetSprite command, 81-82, 100, 103, 279-80, 344
- puppet sprite property, 277, 343
- puppetTempo command, 83, 280, 344
- puppetTransition command, 83, 280-82, 344
- put...after command, 282-83, 346
- put...before command, 283-84, 346
- put command, 282, 347
  - assigning values to variables, 60, 62-63
  - calculating the value of an expression, 62
  - changing properties of objects, 62-63, 67
  - for debugging, 62
  - defining callback factories, 145-46
  - naming XObjects, 131
- put...into command, 284, 346

## Q

- quick reference summary of Lingo vocabulary, 335-48
- quit command, 285, 340, 343, 344
- quotation marks, as used in this manual, 3
- QUOTE character constant, 62, 99, 285, 338
- quoted literal, defined, 371

## R

- radio buttons
  - creating, 24, 84
  - defined, 83
  - determining the action of, 187-88
  - reacting to clicked, 85
  - selecting, 85-86
  - See also* buttons
- random function, 102, 285-86, 342
- recorder formats, and video output, 155
- registration point properties, of sprites, 164

- regular buttons, creating, 24-26. *See also* buttons
- relational operation/operator, defined, 371
- repeat control structure, 172
- repeat option, of the playAccel command, 274
- repeat while keyword, 73, 286, 337
  - specifying a delay during, 77
- repeat while loops, and the delay command, 67
- repeat with keyword, 73, 119, 123, 286-87, 337
- Replace All command (Script menu), 19
- Replace And Find command (Script menu), 19
- replacing, with searching, 19
- ResEDIT
  - adding or removing XObjects, 129
  - viewing XObjects, 126
- reserved words, 58
- resource, defined, 371
- resource files
  - closing, 190
  - displaying resources in, 294
  - opening, 267
- resource fork, defined, 371
- restart command, 287, 344
- result function, 287-88, 337
- return, defined, 371
- RETURN character constant, 62, 289, 338
- Return key
  - automatic formatting of code, 16
  - testing for, 44
- return keyword, 288, 337
- right sprite property, 81, 163, 164, 193, 289, 336
- rnd handler, 101, 102
- rollOver function, 97-98, 289-90, 341
  - monitoring mouse motion, 98-99
- rollOver sample movie, 96-98



## S

- saturation, defined, 372
- Saving Text example XObject, 135
- scan converters, and video output, 156
- scan line, defined, 372
- Score, defined, 372
- Score Script editing window,
  - opening, 10
- Score Scripts, 7, 8
  - accessing, 8
  - creating loops with, 23-24
  - editing window
    - illustrated, 9
    - opening, 10
  - global variables, 61
  - replacing, 16
  - scripting guidelines, 169-70
  - writing, 14
  - See also* scripts
- Score Script window, accessing Score Scripts, 8
- Score window
  - defined, 372
  - displaying script notation, 14
  - illustrated, 10
  - Palette channel, 154
  - removing scripts, 19-20
  - Score Scripts, 8
  - script numbers, 16-17, 20
    - displaying, 14
  - script pop-up menu, illustrated, 17
  - viewing scripts of cells, 32
  - watching the playback head cycle through the frames, 22
- screen coordinate system, illustrated, 161. *See also* coordinates
- Script button, accessing Movie Scripts, 8
- Script channels
  - and the alpha channel, 153
  - displaying script numbers, 17
  - operation of, 8
  - scripts placed in, 8
- Script channel scripts, 8
  - creating loops with, 23-24

- script editing windows, 7
  - editing text, 16
  - entering text, 16
  - illustrated, 9
  - opening, 10-12
  - opening multiple, 11-12
  - See also* scripts
- script editor
  - closing, 16
  - using, 14-20
  - using Edit menu commands, 16
  - writing scripts, 14
  - See also* scripts
- script entry area, pasting commands to, 15
- scripting environment, 9-14
  - elements of, illustrated, 9
  - Message window, 12-14
  - Script editing windows, 9-12
  - See also* scripts
- scripting guidelines, 169-73
- Script menu, 9, 18-19
  - Find Again command, 19
  - Find command, 18
  - Find In Next Script command, 19
  - illustrated, 18
  - Replace All command, 19
  - Replace And Find command, 19
  - utilities, 11
- script menu item property, 290
- script notation, displaying in the Score window, 14
- script numbers, 16-17, 20
  - defined, 372
  - displaying in the Score window, 14
  - removing, 20
  - viewing, 34
  - See also* scripts
- script pop-up menu
  - illustrated, 17
  - removing unwanted scripts, 20
- scripts
  - abbreviated commands, 55
  - attached to sprite cells, 8
  - attaching to buttons, 26
  - boolean expressions, 58, 362
  - case sensitivity, 56, 172

- scripts (*continued*)
  - Cast Scripts, 7, 8
  - character spaces in, 55-56
  - clean-up-and-exit, 122-23
  - comments, 56, 172-73
  - communicating with objects, 107-8
  - components of, 53-87
  - constants, 62, 167, 364
  - control structures, 70-79
  - defined, 54, 372
  - displaying, 30-32
  - displaying notation in the Score window, 14
  - displaying numbers in the Score window, 14, 16-17
  - duplicate names in, 87
  - editing, 7, 9, 14, 16-17
  - entering, 7, 9
  - event scripts, 7-8, 9, 66-67
  - frame markers, 27-28, 171
  - functions, 64, 67-68
  - guidelines for script placement, 166-73
  - indicating location, 51
  - labels, 27-28, 171, 368
  - line breaks in, 29
  - literals in, 57-58, 368
  - long, 29
  - loops, 22-29, 369
  - message hierarchy, 87
  - monitoring time, 76-78
  - Movie Scripts, 7, 8
  - moving to new locations, 17
  - naming conventions, 172-73
  - nesting level, 51
  - operators, 68-70, 168, 370
  - optional elements, 64-65, 109
  - optional keywords, 55
  - parentheses in, 55, 68, 326, 341
  - preventing from executing, 200-201
  - puppets, 80, 81-83, 100-103
  - removing, 19-20
  - Score Scripts, 7, 8
  - scripting guidelines, 169-73
  - script numbers, 14, 16-17, 20
  - script writing style, 172
  - searching, 18-19
  - sprites, 80-86
  - testing with the Message window, 12



- scripts (*continued*)
  - testing structures, 70-79
  - tracing, 12, 49-51
  - turning off event states, 78
  - types, 7-9
  - uppercase and lowercase letters
    - in, 56
  - uses of, 6
  - using label names, 27
  - using the script editor, 14-20
  - values, 59-63, 373
  - variables, 59-62, 373
  - writing, 14
  - writing conventions, 56
  - See also* Cast Scripts; event scripts; Movie Scripts; Score Scripts; statements
- script window, 7
- script writing conventions, 56
- script writing style, 172
- searching scripts, 18-19
  - and replacing, 19
  - search order, 19
- select, defined, 372
- Select All command (Edit menu),
  - using in the script editor, 16
- selection, defined, 372
- selection function, 290-91, 346
- selEnd text property, 291, 346
- selStart text property, 291-92, 346
- sequences
  - adding labels to, 42
  - nesting, 40
- serialPort XObject, 50, 129
- set...= command, 292, 347
- setCallback command, 142, 292-93, 339
  - specifying callback handlers, 146
- set command
  - assigning values to variables, 60, 62-63
  - changing properties of objects, 62-63, 67
  - creating callback objects, 146
- set command (*continued*)
  - creating instances of XObjects, 131
  - creating objects from factories, 110
- setPuppets handler, 100
- set...to command, 292, 347
- shake handler, 101-2
- shiftDown function, 293, 339
- short. *See* date function; time function
- showGlobals command, 293, 347
  - displaying current global variables, 61
- showLocals command, 294, 347
  - displaying current local variables, 61
- showResFile command, 294, 339
- showXlib command, 294-95, 339
  - viewing XCMDs and XFCNs, 129, 140
  - and XLibraries, 129, 294-95
  - and XObjects, 127, 294-95
- shutDown command, 295, 344
- Simple Factories example movie, 113-23
- Simple Puppets sample movie, 100-103
- 16-bit display mode, 151
- software simulations, 6
- soundBusy function, 297, 340, 344
- Sound channel, 8, 136
  - as a puppet, 278
- soundEnabled property, 297, 344
- sound fadeIn command, 295-96, 344
- sound fadeOut command, 296, 344
- soundLevel property, 297-98, 344
- sound playFile command, 296, 339, 344
- sounds
  - fading in, 295-96
  - fading out, 296
  - playing under script control, 103, 136, 278-79
  - quick reference summary of vocabulary and syntax, 344
- sounds (*continued*)
  - stopping, 297
  - volume level of, 297-98, 319
- sound stop command, 297, 344
- spaces. *See* character spaces
- special messages, quick reference
  - summary of vocabulary and syntax, 339
- speedChange argument, 121
- spriteBox command, for puppet
  - sprites, 163, 182, 299-300, 337
- sprite cells
  - plus sign (+) displayed in, 14
  - scripts attached to, 8, 80-81
- sprite...intersects comparison
  - operator, 69-70, 298-99, 340
- sprite keyword, 298
- sprite properties, dictionary of, 175-326. *See also* sprites
- sprites, 80-86
  - animated moveable, 96
  - background colors, 179-80
  - border properties, 163
  - clickable, 95
  - constraining the position of, 96, 164, 193-94
  - controlling properties of, 80
  - controlling vertical and horizontal location, 82
  - declaring to be puppets, 81-82
  - defined, 80, 372
  - foreground color of, 209-10
  - handling, 80-81
  - horizontal position of, 230
  - ink effects, 219-20
  - line size of, 230
  - moveable, 95, 250
    - animated, 96
    - constrained, 96
  - as puppets, 81-83, 100-103, 279-80
  - quick reference summary of vocabulary and syntax, 336-37
  - registration point properties, 164
  - sprite coordinates, 161
  - sprite positions, 163-64, 230, 231



- sprites (*continued*)
    - sprite properties, 81, 163-64
      - dictionary of, 175-326
    - text, 81, 93-94
    - tracking movement, 178
    - types, listed, 316
    - vertical position of, 231
    - vertical size of, 214
  - sprite scripts, 8
    - defined, 372
  - sprite...within comparison operator, 69-70, 299, 340
  - sqrt function, 300, 342
  - square brackets ([ ]), indicating optional elements, 64-65, 109, 176
  - square roots of numbers, 300
  - Stage
    - centering, 185
    - constraining, 162
    - defined, 372
    - position of, 160-61, 300-302
    - redrawing, 317
  - stageBottom function, 300-301, 345
    - and Stage coordinates, 160-61
  - stageColor property, 301, 345
  - Stage coordinates, 160-64. *See also* coordinates
  - Stage coordinate system, illustrated, 162. *See also* coordinates
  - stageLeft function, 301, 345
    - and Stage coordinates, 160-61
  - stageRight function, 302, 345
    - and Stage coordinates, 160-61
  - stageTop function, 302, 345
    - and Stage coordinates, 160-61
  - standard file dialog, defined, 372
  - \*Standard.xlib
    - automatic opening of, 140
    - built-in XObjects, 129
  - startMovie. *See* on startMovie movie handler
  - startMovie handler, 31
    - defining custom menus, 79
    - in Movie scripts, 119, 169, 264-65
  - starts comparison operator, 69-70, 303, 340, 346
  - startTimer command, 76, 303, 346
  - statement list, defined, 373
  - statements
    - commenting out, 90
    - defined, 7, 54, 373
    - editing, 16
    - entering, 16
    - error messages for invalid, 13
    - multiline, 4, 29, 54
    - optional elements, 64-65, 109
    - syntax, 7
    - testing in the Message window, 13, 50
    - See also* scripts; syntax
  - state tests, 71
  - stepMovie handler
    - in Movie Scripts, 119, 169, 263, 265
    - in Score Scripts, 170
    - when to use, 171
    - See also* on stepMovie movie handler
  - stillDown function, 304, 339, 341
  - stop. *See* sound stop command
  - stopMovie handler, 119, 169
    - in Movie Scripts, 266
    - See also* on stopMovie movie handler
  - stretch sprite property, 304-5, 317, 337
  - string concatenator symbol (&&), 56, 99
  - string function, 305, 346
  - string literals, character spaces in, 56
  - string operators, 70
  - stringP function, 241, 305-6, 340
  - strings
    - comparing two, 195
    - defined, 373
  - subtraction (-) arithmetic operator, 68, 329, 341
  - switchColorDepth property, 306, 343, 345
  - symbol definition operator (#), 62, 333, 342
  - symbolP function, 241, 306-7, 340
  - symbols, 62
    - defined, 373
    - Lingo, dictionary of, 326-33
    - See also* individual symbols
  - sync option, of the playAccel command, 274
  - syntax
    - for all Lingo language elements, 175-333
    - defined, 373
    - getting help with, 15-16
    - for Lingo objects to communicate with Lingo scripts, 128
    - of Lingo statements, 7
    - quick reference summary of, 335-48
    - as represented in this manual, 3
    - typographic conventions used, 336
  - system (Macintosh), quick reference summary of vocabulary and syntax, 344-45. *See also* Macintosh system
  - System Palette, for color, 152
- ## T
- TAB character constant, 62, 307, 338
  - Tab key, automatic formatting of code, 16
  - Tempo channels, 8
  - tempo (in frames per second), setting, 280
  - tempo option, of the playAccel command, 274
  - testing structures, 70-79



- tests
  - of scripts, 70-79
  - types, 71
- text
  - clearing from the Message window, 14
  - deleting, 199
  - editable text fields, 41-42
    - creating, 26-27
  - editable text sprites, creating, 81, 93
  - fonts, 308
  - font size, 157, 309
  - font style, 310
  - line spacing, 309
  - quick reference summary of vocabulary and syntax, 345-46
  - searching, 18-19
  - testing strings, 44
- textAlign text property, 308, 346
- Text cast window
  - defining factories, 171
  - defining macros, 171
- text field castmembers, creating, 10
- text field input, for loop control, 26-27
- text fields, editable, 26-27, 41-42, 93-94
- textFont text property, 308, 346, 356
- textHeight text property, 309, 346
- text operators, 168
  - dictionary of, 326-33
- textSize text property, 309, 346
- text sprites
  - creating, 81
  - making editable, 27, 81, 93, 201
  - See also* sprites
- text strings
  - as literals, 57
  - putting spaces in, 63
  - testing, 44
- textStyle text property, 310, 346
- text text property, 307, 337, 346
- Text window, pasting commands to, 15
- the backColor of sprite sprite property, 179-80
- the beepOn property, 181-82
- the bottom of sprite sprite property, 182
- the buttonStyle property, 183
- the castNum of sprite sprite property, 184-85
- the centerStage property, 185
- the checkBoxAccess property, 187-88
- the checkBoxType property, 188
- the checkMark of menuItem property, 188-89
- the clickOn function, 189
- the colorDepth property, 191
- the colorQD function, 192
- the commandDown function, 192
- the constraint of sprite sprite property, 193-94
- the controlDown function, 195-96
- the cursor of sprite sprite property, 197
- the date function, 198
- the doubleClick function, 201
- the enabled of menuItem menu item property, 202-3
- the exitLock property, 204-5
- the fixStageSize property, 207-8
- the floatPrecision property, 209
- the foreColor of sprite sprite property, 209-10
- the frame function, 210
- the freeBlock function, 211
- the freeBytes function, 212
- the height of sprite sprite property, 214
- the hilite of cast button property, 215
- the immediate of sprite sprite property, 218
- the ink of sprite sprite property, 219-20
- the keyCode function, 225
- the keyDownScript property, 225-26
- the labelList function, 226
- the lastClick function, 227
- the lastEvent function, 227
- the lastKey function, 227
- the lastRoll function, 227-28
- the left of sprite sprite property, 228
- the lineSize of sprite sprite property, 230
- the locH of sprite sprite property, 230
- the locV of sprite sprite property, 231
- the machineType function, 231-32
- the mouseCast function, 245
- the mouseChar function, 245
- the mouseDown function, 246
- the mouseDownScript property, 246-47
- the mouseH function, 247
- the mouseItem function, 247-48
- the mouseLine function, 248
- the mouseUp function, 248
- the mouseUpScript property, 249
- the mouseV function, 249-50
- the mouseWord function, 250
- the movie function, 251
- then. *See* if...then keywords; when keyDown then command; when mouseDown then command; when mouseUp then command; when timeout then command
- the name of cast cast property, 253-54
- the name of menuItem menu item property, 254-55



- the name of menu **menu** property, 254
- then keyword, and event scripts, 74-76
- the number of cast **cast** property, 256
- the number of castmembers **property**, 256-57
- the number of chars in **chunk** function, 257
- the number of items in **chunk** function, 257-58
- the number of lines in **chunk** function, 258
- the number of menuItems **menu** property, 258-59
- the number of menus **menu** property, 259
- the number of words in **chunk** function, 259-60
- theObjects variable, 116, 123
- the optionDown function, 268-69
- the pathName function, 269
- the pauseState function, 270
- the perFrameHook property, 271-72
- the picture of cast **cast** property, 272
- the puppet of sprite **sprite** property, 277
- the result function, 287-88
- the right of sprite **sprite** property, 289
- the script of menuItem **menu** item property, 290
- the selEnd text property, 291
- the selStart text property, 291-92
- the shiftDown function, 293
- the soundEnabled property, 297
- the soundLevel property, 297-98
- the sqrt function, 300
- the stageBottom function, 300-301
- the stageColor property, 301
- the stageLeft function, 301
- the stageRight function, 302
- the stageTop function, 302
- the stillDown function, 304
- the stretch of sprite **sprite** property, 304-5, 337
- the switchColorDepth property, 306
- the textAlign of field text **property**, 308
- the text of cast **cast** property, 307
- the textFont of field text **property**, 308
- the textHeight of field text **property**, 309
- the textSize of field text **property**, 309
- the textStyle of field text **property**, 310
- the ticks function, 311
- the time function, 311-12
- the timeoutKeyDown property, 312
- the timeoutLapsed property, 312
- the timeoutLength property, 313
- the timeoutMouse property, 313
- the timeoutPlay property, 313-14
- the timeoutScript property, 314
- the timer property, measuring time, 76-77, 315
- the top of sprite **sprite** property, 315
- the type of sprite **sprite** property, 316-17
- the volume of sound **sound** property, 319
- the width of sprite **sprite** property, 323
- the word, designating properties, 310
- 32-bit display mode, 151
- 32-bit QuickDraw color, 150
  - and the alpha channel, 153
- ticks
  - defined, 373
  - setting timeout length, 31, 47
- ticks function, 311, 346
- time
  - monitoring, 76-78, 311-12
  - quick reference summary of vocabulary and syntax, 346
- time function, 311-12, 346
- timeOut event
  - and event scripts, 74
  - setting, 46-47
  - setting length, 31
  - turning off, 78
  - See also when timeOut then command
- timeoutKeyDown property, 312, 346
- timeoutLapsed property, 77-78, 312, 346
  - and the when timeOut then command, 322
- timeoutLength property, 31, 76, 77, 313, 346
  - and the when timeOut then command, 322
- timeoutMouse property, 313, 346
- timeoutPlay property, 313-14, 346
- timeouts, 312-14, 321-22
  - creating, 46-47
- timeoutScript property, 314, 346
- timer
  - resetting, 76, 303
  - using, 76-77
- timer property, measuring time, 76-77, 315, 346
- Tool window
  - creating button castmembers, 10
  - creating buttons, check boxes, and radio buttons, 84
  - creating regular buttons, 24-25
  - creating text field castmembers, 10
- top sprite property, 81, 163, 164, 194, 315
- to word, 315
- trace, defined, 373



- Trace checkbox, obtaining a journal of statements, 49-50
- trace function, 49-51
- tracing scripts, with the Message window, 12, 49-51
- tracing symbols, in the Message window, 50-51
- Transition channels, 8
- transitions
  - audio, 6
  - between frames, of puppets, 280-82
  - graphic, 6, 280-82
  - table of, 281
- troubleshooting, movies, 12, 49-51
- TRUE logical constant, 58, 316, 338
- type sprite property, 316-17
- typewriter font
  - as used in the Lingo dictionary, 176
  - as used in this manual, 3
- typographic conventions
  - in the Lingo dictionary, 176
  - in the quick reference summary, 336

## U

- unconstrainFace handler, 96
- Undo command (Edit menu), using in the script editor, 16
- updateStage command, 317, 337, 344
- uppercase letters, in scripts, 56, 172, 176

## V

- value function, 318, 342, 346
- values
  - and arrays, 111, 240-41, 252
  - assigning, calculating, and communicating, 59-63
  - comparing two, 69-70
  - constants, 62
  - defined, 59, 373
  - establishing, 62-63

- values (*continued*)
  - variables, 59-63, 109-10
- variables, 59-62
  - for arguments, establishing, 128
  - assigning values to, 60
  - case conventions for names, 172
  - defined, 59, 373
  - defining, 62-63
  - global, 60, 61, 109, 212-13, 293, 367
  - initializing, 59
  - instance variables, 109-10, 221-22, 243-44
  - local, 60, 61, 109, 294, 369
  - naming, 172-73
  - quick reference summary of
    - vocabulary and syntax, 347
  - symbols, 62
  - variable numbers of, 111
  - See also* individual variables by name
- VCR (videocassette recorder), defined, 373
- version system variable, 318, 345
- VHS (S-VHS), defined, 373
- Video8, defined, 374
- video cameras, displaying images from, 137-38
- video cards, and video output, 156
- videocassette recorder (VCR). *See* VCR (videocassette recorder)
- Videodisk example XObject, 135
- videodisk players, controlling with XObjects, 135
- videodisks
  - defined, 374
  - displaying images from, 137-38
- video input card, defined, 373
- video-in-window, defined, 373
- Video-in-Window example XObject, 137-38
- video monitors
  - and Stage coordinates, 160-61
  - and video output, 157
- video output, 154-57
  - color saturation, 157

- font sizes, 157
- pixel point sizes, 156
- planning for, 154-55
- recorder format, 155
- scan converters, 156
- video cards, 156
- video monitors, 157
- video transfer method, 156
- Video Playback example XObject, 137
- video recorder formats, and video output, 155
- videotape
  - defined, 374
  - recording movies to, 136-37
- video tape recorder (VTR). *See* VTR (video tape recorder)
- video transfer method, and video output, 156
- virtual cast facility, 158-59
- vocabulary, Lingo
  - dictionary of, 175-333
  - quick reference summary of, 335-48
- volume sound property, 319
- vPlay handler, using built-in XObjects, 133
- VTR (video tape recorder), defined, 374
- vtrXObj XObject, controlling consumer videotape decks, 137

## W

- whatFits option, of the playAccel command, 274
- when. *See* when keyDown then command; when mouseDown then command; when mouseUp then command; when timeOut then command
- when keyDown, and event scripts, 9
- when keyDown statement, 44
- when keyDown then command, 319-20



- when keyword, and event scripts, 66, 74-76
- when mouseDown, and event scripts, 9
- when mouseDown then command, 320, 339. *See also* mouseDown handler
- when mouseUp, and event scripts, 9
- when mouseUp then command, 321, 339. *See also* mouseUp handler
- when...nothing command, for turning off event scripts, 78
- when...then keywords, and event scripts, 74-76, 98-99
- when...then tests, and the delay command, 67
- when timeOut command, and event scripts, 9, 77-78, 339
- when timeOut then command, 321-22, 346. *See also* timeOut event
- whichCursor parameter, 196
- whichItem expression, 189
- whichMenu expression, 189
- whichSprite expression, 194
- whichSprite handler, 102
- while. *See* repeat while keyword
- width sprite property, 323, 337
- windows, creating Macintosh-style, 135
- Windows example XObject, 135
- Window XObject, 129, 135
- with. *See* repeat with keyword
- within. *See* sprite...within comparison operator
- word...of chunk expression keyword, 324, 346
- words
  - defined, 374
  - deleting, 199

- highlighting, 215
- Lingo, dictionary of, 175-326
- misspelled, 24
- quick reference summary of Lingo vocabulary, 335-48
- reserved, 58
- searching, 18-19
- selecting in the script editor, 16
- summary of major word groups, 166-73
- types of Lingo words, 166
- See also* keywords
- words. *See* the number of words in chunk function

## X

- XCMDGlue example XObject, 136
- XCMDGlue XObject, 126, 129
  - using XCMDs and XFCNs with, 130, 136, 140-43
- XCMD resources (HyperTalk)
  - closing, 141, 190
  - defined, 374
  - opening, 130, 140
  - using, 141-48
  - viewing, 129, 140
  - and XObjects, 126-27
- XCOD resources
  - defined, 374
  - XObject code modules stored as, 126, 129
- xFactoryList function, 325, 339, 344
- XFCN resources (HyperTalk)
  - closing, 141, 190-91
  - defined, 374
  - opening, 130, 140
  - using, 141-48
  - viewing, 129, 140
  - and XObjects, 126-27

## XLibraries

- closing, 130, 190-91
- defined, 374
- displaying, 294-95
- opening, 268
- removing from memory, 130

## XLibraries (continued)

- of XObjects, 129
- XObject Developer Kit, writing XObjects, 128, 168
- XObjects, 125-38
  - adding, 129
  - built-in, 129, 132-38
  - characteristics of, 127-28
  - creating, 50
  - defined, 374
  - described, 126-27
  - displaying, 294-95
  - documents associated with, 130
  - drivers, 126
  - generic, 127
  - and Hypertalk XFCN and XCMD resources, 126-27, 129
  - instances of, 126
    - creating, 131, 243-44
  - location of, 129
  - methods, 127, 242
    - common, 130-31
    - viewing, 130
  - and NuBus cards, 126
  - opening, 130
  - predefined methods, 168, 241-43
  - removing, 129
  - removing from memory, 130, 132
  - uses of, 125
  - viewing, 126, 127, 129
  - writing your own, 128-32
  - XLibraries of, 129

## Z

- zoomBox command, 325-26, 337
- zoom effect, creating, 325-26







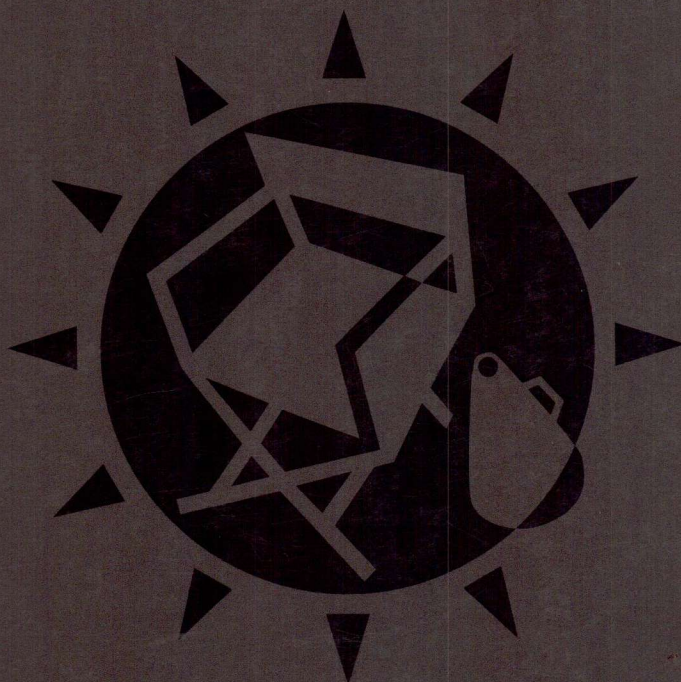












**MACROMIND**

*the multimedia company*

MacroMind, Inc. 600 Townsend Street, Suite 310 San Francisco, CA 94103 (415) 442-0200