MACROMEDIA®

Lingo™ Dictionary

Director®

Version 4

First Edition: February 1994
Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

This dictionary presents a listing of non-alphabetical symbols, followed by an alphabetical listing of Lingo words. Every Lingo language element is included.

Where useful, notes about proper uses for an element are included.

Four appendixes are included:

- Appendix A, “Lingo Changes,” lists new Lingo elements in Director 4 and outdated Lingo that is no longer fully supported in this version of Director.
- Appendix B, “ASCII Character Chart,” shows the ASCII equivalents of characters on the Macintosh keyboard.
- Appendix C, “Lingo Quick Reference,” organizes Lingo elements into usage categories.
- Appendix D, “Octal to Decimal Converter,” gives the decimal equivalents for octal cast identifiers, which are no longer supported in Director 4.

Typographic conventions

This dictionary uses the following conventions:

- 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 that you type `open`, but that you substitute an actual filename for the word *fileName*.
- Curly brackets ({ }) enclose optional elements, which you may include if you need them. Some commands have optional arguments that you use in particular circumstances. An optional element may or may not change what the statement does. Don't type the brackets.
- Square brackets ([ ]) enclose lists.
- Optional words are sometimes included to make a statement easier to read (`go to frame 23` is the same as `go 23`).
- The continuation symbol (¬) indicates where Lingo text has wrapped to the next line. Lines broken in this way are actually one line of Lingo.
- In the Description 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`). This is a common convention used to make the components of these words easier to read.

Symbols

# symbol definition operator

Syntax *#symbolName*

Description This symbol definition 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.

Example This statement sets the variable named `state` to the symbol `#Playing`:

```
put #Paused into state
```

– arithmetic operator

Syntax `-`*expression*

Syntax *expression1* `-` *expression2*

Description This arithmetic operator reverses the sign of the value of an expression, or performs arithmetic subtraction on two numerical expressions.

- The usage `-`*expression* reverses the sign of the value of the expression. This is an arithmetic operator with a precedence level of 5.
- The usage *expression1* `-` *expression2* subtracts *expression2* from *expression1*. When both expressions are integers, the difference is an integer. When either or both expressions are floating-point numbers, the difference is a floating-point number. This is an arithmetic operator with a precedence level of 3.

Example This statement reverses the sign of the expression 2 + 3.

```
put -(2 + 3)
```

The result is –5.

This statement subtracts the integer 2 from 5, and then displays the result in the message window:

```
put 5 - 2
```

The result is 3, which is an integer.

This statement subtracts the floating-point number 1.5 from 3.25, and then displays the result in the message window:

```
put 3.25 - 1.5
```

The result is 1.75, which is a floating-point number.

– – (double hyphen) comment delimiter

Syntax `--` {*comment*}

Description This comment delimiter symbol indicates the beginning of a script comment. On any line, what is between the comment symbol (double hyphen) and the end-of-line return character is interpreted as a comment instead of a Lingo statement.

Example This handler uses a double hyphen to make the second line a comment:

```
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
```

& text operator

Syntax *expression1* `&` *expression2*

Description This text operator concatenates two expressions. If either *expression1* or *expression2* is 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.

Example This statement concatenates the strings “abra” and “cadabra”:

```
put "abra" & "cadabra"
```

The result is the string “abracadabra”.

This statement concatenates the strings “$” and the content of the variable named `price`. The concatenated string is then assigned to the text cast member Price:

```
put "$" & price into field "Price"
```

&& — text operator

Syntax *expression1* `&&` *expression2*

Description This text operator concatenates 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.

Example This statement concatenates the strings “abra” and “cadabra”, and inserts a space between the two:

```
put "abra" && "cadabra"
```

The result is the string “abra cadabra”.

This statement concatenates the strings “Today is” and today’s date in the long format, and inserts a space between the two:

```
put "Today is" && the long date
```

If today’s date were Tuesday, March 15, 1994, the result would be Tuesday, March 15, 1994.

( ) — grouping operator

Syntax `(expression)`

Description This grouping 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.

Example These statements use the grouping operator to change the order in which operations occur. The result appears below each statement:

```
put (2 + 3) * (4 + 5)
-- 45
```

```
put 2 + (3 * (4 + 5))
-- 29
```

```
put 2 + 3 * 4 + 5
-- 19
```

* arithmetic operator

Syntax *expression1* `*` *expression2*

Description This arithmetic operator performs an arithmetic multiplication on two numerical expressions. If both expressions are integers, the product is an integer. If either or both expressions are floating-point numbers, the product is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

Example This statement multiplies the integers 2 and 3, and then displays the result in the message window:

```
put 2 * 3
```

The result is 6, which is an integer.

This statement multiplies the floating-point numbers 2.0 and 3.1414, and then displays the result in the message window:

```
put 2.0 * 3.1416
```

The result is 6.2832, which is a floating-point number.

+   arithmetic operator

Syntax *expression1* `+` *expression2*

Description This arithmetic operator performs an arithmetic sum on two numerical expressions. If both expressions are integers, the sum is an integer. If either or both expressions are floating-point numbers, the sum is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

Example This statement adds the integers 2 and 3, and then displays the result in the message window:

```
put 2 + 3
```

The result is 5, which is an integer.

This statement adds the floating-point number 2.5 and 3.25, and then displays the result in the message window:

```
put 2.5 + 3.25
```

The result is 5.75, which is a floating-point number.

/   arithmetic operator

Syntax *expression1* `/` *expression2*

Description This arithmetic operator performs an arithmetic division on two numerical expressions, dividing *expression1* by *expression2*. If both expressions are integers, the quotient is an integer. If either or both expressions are floating-point numbers, the quotient is a floating -point number.

This is an arithmetic operator with a precedence level of 4.

Example This statement divides the integer 22 by 7, and then displays the result in the message window:

```
put 22 / 7
```

The result is 3, which is an integer.

This statement divides the floating-point number 22.0 by 7.0, and then displays the result in the message window:

```
put 22.0 / 7.0
```

The result is 3.1429, which is a floating-point number.

< comparison operator

Syntax *expression1* < *expression2*

Description This comparison 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 comparison 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 comparison 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 comparison 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.

> comparison operator

Syntax *expression1* > *expression2*

Description This comparison 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 comparison 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.

[ ] list operator

Syntax [*entry1*, *entry2*, *entry3*, …]

Description This list operator specifies that the entries within the square brackets are a list.

There are four types of lists:

- Unsorted linear lists
- Sorted linear lists
- Unsorted property lists
- Sorted property lists.

Each entry in a linear list is a single value that has no other property associated with it. Each entry in a property list consists of a value and a property. The property appears before the value and is separated from the value by a colon. You cannot store a property in a linear list. When using strings as entries in a list, enclose the string in quotation marks.

For example, [6, 3, 8] is a linear list. The numbers have no properties associated with them. However, [#gears:6, #balls:3, #ramps:8] is a property list. Each number has a property, in this case a piece of machinery, associated with it. This property list could be useful for tracking how many of each piece of machinery are currently on the stage in the mechanical simulation. Properties can appear more than once in a property list.

Lists can be sorted in alphanumeric order. A sorted linear list is ordered by the values in the list. A sorted property list is in order by the properties in the list. You sort a list by using the appropriate command for a linear list or property list.

A linear list or a property list can contain no values at all. An empty list consists of two square brackets (`[]`). To clear a list, set the list to `[]`.

You can modify, test, or read items in a list.

Lists are an alternative to factories and `mGet` and `mPut`, which were supported in previous versions of Director. In Director 4, it is recommended that you use lists; they are a simpler way of accomplishing the same result.

You do not have to worry about explicitly disposing of lists. Lists are automatically disposed of when they are no longer referred to by any variable. When the list is held within a global variable, it persists from movie to movie.

Example This statement defines a list by making the variable `machinery` equal to the list:

```
set machinery = [#gears:6, #balls:3, #ramps:8]
```

This handler sorts the list machinery, and then displays the result in the message window:

```
on sortList machinery
  sort machinery
  put machinery
end sortList
```

The result is [#balls:3, #gears:6, #ramps:8].

This statement creates an empty linear list:

```
set x = []
```

This statement creates an empty property list:

```
set x = [:]
```

See also `add`, `addAt`, `append`, `count`, `findPos`, `findPosNear`, `getaPropAt`, `getAt`, `getLast`, `getPos` functions; `deleteAt`, `deleteProp`, `setAt`, `setaProp`, `sort` commands; `ilk`, `list`, `max`, `min` functions

¬ (continuation symbol) 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 This special character, when used as the last character in a line, makes the statement continue on the next line. Lingo then interprets the lines as one continuous statement. You can do this on several successive lines.

Create this character by pressing Option-Return.

Example This statement uses the ¬ character to wrap the statement onto several lines:

```
set the castNum of sprite mySprite ¬
  to the number of cast ¬
  "This is a long cast name."
```

A

abbreviated

See the `date` and `time` functions.

abort — command

Syntax

```
abort
```

Description

This command has Lingo exit the current handler and any handler that called it without executing any of the remaining statements in the handler. This differs from the `exit` keyword, which returns to the handler from which the current handler was called.

The `abort` command does not quit Director.

Example

This statement has Lingo exit the handler and any handler that called it when the amount of free memory is less than 50K:

```
if the freeBytes < 50*1024 then abort
```

abs — function

Syntax

`abs (`*numericExpression*`)`

Description

This function calculates the absolute value of a numerical expression. If *numericExpression* is an integer, its absolute value is also an integer. If *numericExpression* is 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).

Example This statement calculates the absolute value of –2.2 and displays the result in the message window:

```
put abs(–2.2)
```

Example This statement determines whether the absolute value of the difference between the current mouse position and the value of the variable `startV` is greater than 30. If it is, the foreground color of sprite 6 is changed.

```
if abs (the mouseV - startV) > 30 then ¬
set the forecolor of sprite 6 to 95
```

the actorList — property

Syntax

```
the actorList
```

Description All objects in `the actorList` receive a `stepFrame` message when the playback head advances to a new frame. This makes using `the actorList` a more powerful alternative to the `perFrameHook` property used in previous versions of Lingo.

You can clear child objects from `the actorList` by setting `the actorList` to `[]`, which is an empty list.

Example This statement creates a child object from the parent script Moving Ball. All three values are parameters that the script requires:

```
add the actorList, birth(script "MovingBall", 1, ¬
  200,200)
```

This statement displays the contents of `the actorList` in the message window:

```
put the actorList
```

This statement clears `the actorList`:

```
set the actorList = []
```

See also `birth` command

add command

Syntax

`add` *linearList, value*

Description

This command adds the value specified by *value* to the linear list specified by *linearList*. For a sorted list, the value is placed in its proper order. For an unsorted list, the value is added to the end of the list.

Example

This statement adds the value 2 to the list named bids. The resulting list is [3, 4, 1, 2]:

```
set bids = [3, 4, 1]
add bids, 2
```

This statement adds 2 to the sorted linear list [1, 4, 5]. The new item stays in alphanumeric order because the list is sorted:

```
add bids, 2
```

addAt command

Syntax

`addAt` *list*, *position*, *value*

Description

This command adds a value to the list at the position specified by *position*. Use this command when you need to add an item at a specific location in a list.

Example

This statement adds the value 8 to the fourth position in the list named bids, which is [3, 2, 4, 3, 6, 7]:

```
set bids = [3, 2, 4, 5, 6, 7]
addAt bids, 4, 8
```

The resulting value of bids is [3, 2, 4, 8, 3, 6, 7].

addProp command

Syntax `addProp` *list, property, value*

Description This command adds the property specified by *property* and its value specified by *value* to the property list specified by *list*. For an unsorted list, the value is added to the end of the list. For a sorted list, the value is placed in its proper order.

When the property already exists in the list, Lingo creates a duplicate property. You can avoid duplicate properties by using the `setaProp` command to change the new entry's property.

Example This statement adds the property named kayne and its assigned value 3 to the property list named bids, which contains [#gee: 4, #ohasi: 1]. Because the list is sorted, the new entry is placed in alphabetical order:

```
addProp bids, #kayne, 3
```

The result is the list is [#gee: 4, #kayne: 3, #ohasi: 1].

This statement adds the entry kayne: 7 to the list named bids, which now contains [#gee: 4, #kayne: 3, #ohasi: 1]. Because the list already contains the property `kayne`, Lingo creates a duplicate property:

```
addProp bids, #kayne, 7
```

The result is the list [#gee: 4, #kayne: 3, #kayne:7, #ohasi: 1].

See also `add`, `addAt`, and `setaProp` commands; `[]` list operator

after

See the `put…after` command.

alert — command

Syntax `alert` *message*

Description This command causes a system beep and displays an alert dialog box containing the string specified by *message*, and an OK button. This command is useful for providing error messages in your movie. The message can contain up to 255 characters.

Example The following statement produces an alert stating that there is no CD–ROM drive connected:

```
alert "There is no CD-ROM drive connected."
```

This statement produces an alert stating that a file was not found:

```
alert "The file" && QUOTE & filename & QUOTE ¬
  && "was not found."
```

ancestor — property

Syntax

```
property ancestor
```

Description The `ancestor` property allows child objects to use handlers that are not contained within the parent script. The `ancestor` property is typically used with two or more parent scripts. This is useful when you want child objects to share certain behaviors that are inherited from an ancestor, while differing in other behaviors that are inherited from the parents.

The `ancestor` property is typically assigned in the child object's birth handler within the parent script. When you send a message to a child object that does not have a defined handler, that message is forwarded to the script defined by the ancestor property.

For a complete discussion of this topic, see Chapter 10, "Parent Scripts and Child Objects," in *Using Lingo*.

The ancestor script can contain independent property variables that can be accessed by child objects. To refer to property variables within the ancestor script, you must use this syntax.

This statement changes the property variable `legCount` within an ancestor script to 4:

```
set the legCount of me to 4
```

Use the syntax `the` *variableName* `of` *scriptName* to access property variables that are not contained within the current object. This statement allows the variable `myLegCount` within the child object to access the property variable `legCount` within the ancestor script:

```
put the legCount of me into myLegCount
```

Example The following four scripts present an example of using the `ancestor` property. Each of these scripts is a cast member. Using the ancestor script Animal and the parent scripts Dog and Man, they interact with one another to define objects.

The first script Dog sets the property variable `breed` to Mutt; sets the ancestor of Dog to the Animal script; and sets the `legCount` variable that is stored in the ancestor script to 4:

```
property breed, ancestor
on birth me
  set  breed = "Mutt"
  set ancestor of me to birth(script "Animal")
  set the legCount of me to 4
  return me
end birth
```

The second script Man sets the property variable `race` to African; sets the ancestor of Man to the Animal script; and sets the `legCount` variable that is stored in the ancestor script to 2:

```
property race, ancestor
on birth me
  set race to "African"
  set ancestor to birth(script "Animal")
  set the legCount of me to 2
  return me
end birth
```

The third script Animal stores the property variable `legCount` for each child object and defines the `eat` handler:

```
property legCount
on birth me
  return me
end birth
on eat me, what
  put "Eating " & what
end
```

The fourth script creates a child object of Man, displays its variables in the message window, and calls the `eat` handler and displays it in the message window. Since the `eat` handler is not in the parent script Man, Lingo finds the `eat` handler in the ancestor script Animal:

```
set manChild to birth(script "man")
put the legCount of manChild
-- 2
put the race of manChild
-- "African"
eat manChild, "apple"
-- "Eating apple"
```

See also `birth` function; `me` and `property` keywords

and — logical operator

Syntax *logicalExpression1* `and` *logicalExpression2*

Description This logical operator determines whether two logical expressions are both TRUE. Only when both *logicalExpression1* and *logicalExpression2* are TRUE, the result is TRUE (1). When either or both expressions are FALSE, the result is FALSE (0).

The precedence level of this logical operator is 4.

Example This statement determines whether both logical expressions are TRUE and displays the result in the message window:

```
put 1 < 2 and 2 < 3
```

The result is 1, which is the numerical equivalent of TRUE.

The first logical expression in this statement is TRUE; the second logical expression is FALSE. Because both logical expressions are not TRUE, the logical operator gives the result 0, which is the numerical equivalent of FALSE:

```
put 1 < 2 and 2 < 1
-- 0
```

See also `not` and `or` logical operators

append — command

Syntax `append` *list*, *value*

Description This command adds the specified value to the end of the list, regardless of the list's type. This differs from the `add` command, which adds a value to a sorted list in accordance with the list's order.

Example This statement adds the value 2 at the end of the sorted list named bids, which contains [1, 3, 4] even though this is not according to the list's sorted order:

```
set bids = [1, 3, 4]
append bids, 2
```

The resulting value of bids is [1, 3, 4, 2].

See also `add` command

atan — function

Syntax `atan (`*number*`)`

Description This function calculates the arctangent of the specified number. The result is between -pi/2 and +pi/2.

Example This expression gives the arctangent of pi`/4` radians:

```
atan (pi()/4.0)
```

Note that the π symbol cannot be used in a Lingo expression.

the backColor of cast — cast property

Syntax

`set the backColor of cast` *castName* `to` *colorNumber*

Description

This cast property sets the background color of a text cast member.

Example

This statement changes the background color of the text in cast member 1 to the color in palette entry 250:

```
set the backColor of cast 1 to 250
```

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 *whichSprite*. The sprite must be a puppet before you can set its background color using Lingo. Setting the `backColor` using a Lingo script is the same as choosing the background color from the tools window when the sprite is selected on the stage.

The background color applies only to 1-bit bitmap and shape cast members. It does not affect how a text or button cast member is displayed. An 8-bit bitmap is affected, but generally not in a useful way.

The `backColor of sprite` value ranges from 0 to 255 for 8-bit color, and from 0 to 15 for 4-bit color. The numbers correspond to the index number of the background color in the current palette. (A color's index number appears in the color palette's lower left corner when you click the color.)

When 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 puppet sprites—you only have to use one `updateStage` command at the end of all the changes.

One use of the `backColor of sprite` property that works consistently with 8-bit bitmap sprites is specifying which color is to be made transparent using the score ink effect Background Transparent. This is particularly useful when creating or importing anti-aliased graphics or objects from a 3-D rendering program for use over video.

Using a black stage color that is defined as the overlay color by the video card, as well as having images that are anti-aliased against a black background, usually works best. This will produce a dark gray shadow behind the graphic when used over a video source. This is the least objectionable shadow color.

The `backColor of sprite` property can be tested and set.

Example The following statement sets the variable `oldColor` to the background color of sprite 5:

```
put the backColor of sprite 5 into oldColor
```

The following statement randomly changes the background color of a random sprite from sprite 11 to sprite 13 to color number 36:

```
set the backColor of sprite (10 + random(3)) to 36
```

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 This `on keyDown` handler checks whether the backspace key was pressed and, if it was, calls the author-defined handler `clearField`:

```
on keyDown
  if the key = BACKSPACE then clearField
  dontPassEvent
end keyDown
```

beep — command

Syntax `beep [`*numberOfTimes*`]`

Description This command causes the Macintosh computer's speaker to beep the number of times specified by *numberOfTimes*. If *numberOfTimes* is missing, the beep occurs once.

The beep sound is the Alert Sound selected in the Sound control panel. If the Speaker Volume in the Sound control panel is set to 0, the menu bar flashes instead.

Example This statement causes two beeps if the text field Answer is empty:

```
if field "Answer" = EMPTY then beep 2
```

This handler causes up to three beeps whenever a key is pressed:

```
on keyDown
  beep random(3)
end
```

the beepOn — property

Syntax `the beepOn`

Description This property determines whether the Macintosh speaker beeps when the user clicks outside an active sprite—a sprite that has a script associated with it. If the `beepOn` property is set to TRUE, clicking outside active sprites results in a beep.

The `beepOn` property can be tested and set. The default value is FALSE.

Example This statement displays an alert telling the user to click again when the user clicks outside active sprites:

```
if the beepOn = TRUE then alert "Click again."
```

This statement sets the `beepOn` property to TRUE:

```
set the beepOn to TRUE
```

This statement sets the `beepOn` to the opposite of its current setting:

```
set the beepOn to (not the beepOn)
```

before

See the `put`...`before` command.

birth function

Syntax `birth (script` *parentScriptName, value1, value2, ...*`)`

Description This predefined function is used to create child objects from parent scripts. You may define a birth handler within a parent script that creates child objects. The child object shares all the handler definitions of the parent script. The child object has the same property variable names that are declared in the parent script, but each child object has its own values for these properties.

Because the child object is a value, it can be assigned to variables, placed in lists, and passed as parameter.

Being able to assign individual property values to child objects is the primary advantage of using birth handlers. A birth handler must be named `birth`, and must accept `me` as a parameter and return `me`.

Example These statements use a birth handler to create a child object of a parent script. The parent script is a movie script cast member named Bird, which contains these handlers:

```
on birth me
  return me
end
on fly me
  put "I am flying"
end fly
```

These statements create a child object called `myBird`, and make it fly by calling the `fly` handler in the Bird parent script:

```
set myBird to birth (script "Bird")
fly myBird
```

Example This example uses a new Bird parent script, which contains the property variable `speed`:

```
property speed
on birth me, initSpeed
  set speed to initSpeed
  return me
end
on fly me
  put "I am flying at " & speed & "mph"
end
```

Example The following statements create 2 child objects called `myBird1` and `myBird2`. When the `fly` handler is called from the child object, the speed of the object is displayed in the message window:

```
set myBird1 to birth (script "Bird", 15)
set myBird2 to birth (script "Bird", 25)
fly myBird1
fly myBird2
```

This text would appear in the message window:

```
-- "I am flying at 15 mph"
-- "I am flying at 25 mph"
```

See also `ancestor` property; `me` keyword

the blend of sprite — sprite property

Syntax

```
the blend of sprite
```

Description Using this sprite property, you can set or determine the puppet sprite's blend value. Blend values can be from 0 to 100, which correspond to the blend values in the Set Blend dialog box.

Example This statement sets the blend value of sprite 3 to 40 percent:

```
set the blend of sprite 3 to 40
```

This statement displays the blend value of sprite 3 in the message window:

```
put the blend of sprite 3
```

the bottom of sprite — sprite property

Syntax `the bottom of sprite` *whichSprite*

Description This sprite property is the bottom vertical coordinate of the bounding rectangle of the sprite specified by *whichSprite*.

The `bottom of sprite` property can be tested, but not set directly. Use the `spriteBox` command to set the bottom vertical coordinate of a sprite.

Example This statement assigns the vertical coordinate of the bottom of sprite numbered (i + 1) to the variable named `lowest`:

```
put the bottom of sprite (i + 1) into lowest
```

Note *Sprite coordinates are measured in numbers of pixels, starting with (0,0) at the upperleft corner of the Stage. Stage coordinates are measured in numbers of pixels, starting with (0,0) at the upperleft corner of the monitor.*

See also `height`, `left`, `locH`, `locV`, `right`, `top`, and `width` sprite properties; `spriteBox` command

the buttonStyle — property

Syntax `the buttonStyle`

Description This property determines the visual response of buttons when a user clicks a button, and then moves the pointer over other buttons without releasing the mouse button.

The `buttonStyle` property can have these values:

0	list style
1	dialog style

- When `the buttonStyle` property is set to 0 (list style), subsequent buttons highlight when the pointer passes over them. 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 highlights. 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). You can use this property in any type of script.

Example The following statement sets `the buttonStyle` property to 1 (dialog style):

```
set the buttonStyle to 1
```

This statement remembers the current setting of the `buttonStyle` property by putting the current `buttonStyle` in the variable `buttonStyleValue`:

```
put the buttonStyle into buttonStyleValue
```

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 specific cast member.

If *whichCastmember* is a string, it is used as the cast name. If *whichCastmember* is an integer, it is used as the cast number.

Example The following statement sets the hilite of the button cast member named Enter Bid to TRUE:

```
set the hilite of cast "Enter Bid" to TRUE
```

This statement puts the name of sound cast member 132 into the variable `soundName`:

```
put the name of cast 132 into soundName
```

This statement determines whether cast member 9 has a name assigned:

```
if stringP(the name of cast 9) then exit
```

cast backColor

See `the backColor of cast` cast property.

cast castType

See `the castType of cast` cast property.

cast depth

See `the depth of cast` cast property.

cast fileName

See `the fileName of cast` cast property.

cast foreColor

See `the foreColor of cast` cast property.

cast height

See `the height of cast` cast property.

cast hilite

See `the hilite of cast` button property.

cast loaded

See `the loaded of cast` cast property.

cast name

See `the name of cast` cast property.

cast number

See `the number of cast` cast property.

cast palette

See `the palette of cast` cast property.

cast picture

See `the picture of cast` cast property.

cast purgePriority

See `the purgePriority of cast` cast property.

cast rect

See `the rect of cast` cast property.

cast regPoint

See `the regPoint of cast` cast property.

cast scriptText

See `the scriptText of cast` cast property.

cast text

See `the text of cast` cast property.

cast width

See `the width of cast` cast property.

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 cast member associated with the sprite specified by *whichSprite*.

Setting this property lets you switch the cast member assigned to a sprite. The sprite must be a puppet to be able to do this.

A typical use of this is exchanging cast members when a sprite is clicked to simulate the reversed image that appears when a standard button is clicked. You can also make some action in the movie depend on which cast member is assigned to a sprite.

When you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. If you are changing several sprite properties—or several puppet sprites—you only have to use one `updateStage` command after making all of the changes.

The `castNum of sprite` property can be tested and set.

Example The following statement switches the cast member assigned to sprite 3 to cast member number 35:

```
set the castNum of sprite 3 to 35
```

This statement assigns the cast member Narrator to sprite 10 by setting `the castNum of sprite` to Narrator's cast number:

```
set the castNum of sprite 10 = cast "Narrator"
```

See also `number of cast` property

the castType of cast — cast property

Syntax `the castType of cast` *cast member*

Description This property determines the type of the cast member specified by *cast member*. The result is given as a symbol—a Lingo element that starts with the symbol operator (#).

The possible cast types are:

#bitmap	#palette
#button	#picture
#digitalVideo	#script
#empty*	#shape
#filmLoop	#sound
#movie	#text

* No cast member is in the specified position.

Example The following statement displays the type of cast member number 45, which is a PICT image, in the message window:

```
put the castType of cast 45
-- #picture
```

the center of cast — digital video cast property

Syntax `the center of cast` *castName*

Description This cast property interacts with `the crop of cast` cast property. It can be tested and set.

- When `the crop of cast` is FALSE, `the center of cast` has no effect.
- When `the crop of cast` is TRUE and `the center of cast` is TRUE, cropping occurs around the center of the digital video cast member.
- When `the crop of cast` is TRUE and `the center of cast` is FALSE, cropping starts at the top left corner of the sprite that refers to the digital video cast member.

Example This statement causes the digital video cast member Interview to be displayed in the top left corner of the sprite:

```
set the center of cast "Interview" to FALSE
```

See also `the crop of cast` digital video cast property

the centerStage — property

Syntax `the centerStage`

Description This property determines whether the stage is centered on the monitor when the movie is loaded. The statement that includes this property is placed in the movie that preceeds the movie you want it to affect.

- If this property is TRUE, the stage is centered.
- If this property is FALSE, the stage is not centered.

This property is useful for checking stage location before a movie plays from a projector. Place handlers that use this property in the preceeding movie before using the `go to movie` command.

The `centerStage` property can be tested and set. The default value is TRUE.

Example This statement sends the movie to a specific frame if the stage is not centered:

```
if the centerStage = FALSE then ¬
  go to frame "off center"
```

This statement changes the `centerStage` property to the opposite of its current value:

```
set the centerStage to (not the centerStage)
```

See also `fixStageSize` property

char...of — chunk expression keyword

Syntax `char` *whichCharacter* `of` *chunkExpression*

Syntax `char` *firstCharacter* `to` *lastCharacter* `of` *chunkExpression*

This chunk expression keyword identifies a character or a range of characters in a chunk expression. Chunk expressions are used to refer to any character, word, item, or line in any source of text such as text cast members and variables that hold strings.

- The expression *whichCharacter* identifies a specific character.
- The expressions *firstCharacter* and *lastCharacter* identify a range of characters.

The expressions must be integers that specify a character or range of characters in the chunk. Characters include letters, numbers, punctuation marks, spaces, and control characters like Tab and Return.

You can test and set the `char...of` keyword.

Example This statement displays the first character of the string $9.00:

```
put char 1 of "$9.00"
-- "$"
```

This statement displays the entire string $9.00:

```
put char 1 to 5 of "$9.00"
-- "$9.00"
```

Example This statement changes the first five characters of the second word of the third line of a text cast member:

```
put "?????" into char 1 to 5 of word 2 of line 3 ¬
  of field "quiz"
```

See also `the mouseCast`, `the mouseItem`, `the mouseLine`, `the mouseWord` integer functions; `word...of`, `item...of`, and `line...of` chunk expression keywords; `the number of chars in` chunk function; `chars`, `length`, and `offset` functions

chars — function

Syntax `chars(`*stringExpression*`,` *firstCharacter*`,` *lastCharacter*`)`

Description This function identifies a substring of characters in *stringExpression*. The substring starts at *firstCharacter* and ends at *lastCharacter*. The expressions *firstCharacter* and *lastCharacter* must specify a 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 identified. If *lastCharacter* is before *firstCharacter*, the function gives the value EMPTY.

Example This statement identifies the sixth character in the word Macromedia:

```
put chars("Macromedia", 6, 6)
-- "m"
```

This statement identifies the sixth through tenth characters of the word Macromedia

```
put chars("Macromedia", 6, 10)
-- "media"
```

This statement tries to identify the sixth through twentieth characters of the word Macromedia. Because the word has only ten characters, the result includes only the sixth to tenth characters.

```
put chars ("Macromedia", 6, 20)
-- "media"
```

This statement tests whether the word Macromedia has a twentieth character:

```
if chars ("Macromedia", 20, 20) = EMPTY then put
"TRUE"
-- 1
```

See also `char...of` chunk expression keyword; `length` and `offset` functions; `number of chars in` chunk function

charToNum function

Syntax `charToNum(`*stringExpression*`)`

Description This function identifies the ASCII code number that corresponds to the first character of *stringExpression*.

You can use the Lingo `charToNum` function to test for the ASCII value of characters created with the combination of the Control key and one other alphanumeric key. (When the Control key is pressed, it modifies the ASCII value of the key.)

Example This statement displays the ASCII code number for the letter A:

```
put charToNum("A")
-- 65
```

This statement checks whether 0 is the ASCII code number of the character assigned to the variable `nextChar`:

```
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 checkbox or radio button created with button tools in the tools window. There are three possible results:

0	lets the user set checkboxes and radio buttons on and off (this is the default)
1	lets the user set checkboxes and radio buttons on but not off
2	prevents the user from setting checkboxes and radio buttons at all; the buttons can only be set by scripts

The `checkBoxAccess` property can be tested and set. The default value is 0.

Example This statement sets the `checkBoxAccess` property to 1, which lets the user click checkboxes and radio buttons on but not off:

```
set the checkBoxAccess to 1
```

This statement records the current setting of the `checkBoxAccess` property by putting the value in the variable `oldAccess`:

```
put the checkBoxAccess into oldAccess
```

See also `hilite` button property; `checkBoxType` property

the checkBoxType — property

Syntax

```
the checkBoxType
```

Description

This property determines what is inserted in checkboxes to indicate they are selected. There are three possible styles:

0	inserts an "x." This is the default
1	inserts a black rectangle
2	inserts a filled black rectangle

The `checkBoxType` property can be tested and set. The default value is 0.

Example

This statement sets the `checkBoxType` property to 1, which shows a black rectangle in checkboxes when the user clicks them:

```
set the checkBoxType to 1
```

This statement records the current setting of the `checkBoxType` property by putting the value in the variable `oldBoxType`:

```
put the checkBoxType into oldBoxType
```

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 custom menu item is displayed with a checkmark.

- When it is TRUE, a checkmark appears next to the custom menu item.
- When it is FALSE, no checkmark appears.

The *whichItem* expression can be either a menu item name or a menu item number. The *whichMenu* expression can be either a menu name or a menu number.

The `checkMark of menuItem` property can be tested and set. The default value is FALSE.

Example This handler unchecks any items that are checked in the custom menu specified by the argument `theMenu`. 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 `installMenu` command; `enabled`, `name`, `number`, and `script of menuItem` menu item properties; `name of menu` and `number of menus` menu item properties; `menu:` keyword

clearGlobals command

Syntax `clearGlobals`

Description This command sets all user-defined global variables to 0.

Example If you initialize a global variable with a string or value,

```
global foo
put "Director" into foo
```

The global variable *foo* contains the string Director until another string or value is put into the global, or until the `clearGlobals` command is issued. This can be useful when initializing global variables, or when opening a new movie that requires a new set of global variables.

When this command is issued, the global variable *foo* contains 0.

```
clearGlobals
```

the clickLoc — function

Syntax `the clickLoc`

Description This function identifies the last place on the screen where the mouse was clicked. The location is given as a point.

Example The following `on mouseDown` handler displays the last mouse click location:

```
on mouseDown
  put the clickLoc
end mouseDown
```

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.

When you want to detect whether 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 whether the `clickOn` equals 0.

Example This statement checks whether sprite 7 was the last active sprite clicked:

```
if the clickOn = 7 then alert "Sorry — wrong choice."
```

Example This statement sets the forecolor of the last active sprite that was clicked to a random color:

```
set the foreColor of sprite (the clickOn) to ¬
  random(256)-1
```

See also `doubleClick`, `mouseDown`, and `mouseUp` functions

close window — command

Syntax `close window` *windowName*

Description This command closes the window specified by *windowName*.

- To specify a window by name, use the syntax `close window "`*name*`"`, where you replace *name* with the name of a window. Be sure to use the complete path name.
- To specify a window by its number in the `windowList`, use the syntax `close window` *number*, where you replace *number* with the window's number in the window list.

Lingo permits you to attempt to close a window that is already closed.

Example This statement closes the window named Panel:

```
close window "Panel"
```

This statement closes the window that is number 5 in the window list:

```
close window 5
```

See also `open window` command; `windowList` function

closeDA — command

Syntax `closeDA`

Description This command closes all open desk accessories under System 6.x. Under System 7.0 and later, this command has no effect since there are no desk accessories.

See also `openDA` command

closeResFile command

Syntax `closeResFile {`*whichFile*`}`

Description This command closes the resource file specified by the string expression *whichFile*. When the resource file is in a different folder than the current movie, *whichFile* must specify a pathname. When no file is specified, all open resource files are closed.

It is good practice to close any file you have opened as soon as you are finished using it.

Example This statement closes all open resource files:

```
closeResFile
```

This statement closes the file Special Fonts when it is in the same folder as the movie:

```
closeResFile "Special Fonts"
```

This statement closes the file Special Fonts in the folder Special Tools on the same disk as the movie. The disk is identified by the variable currentDrive:

```
closeResFile currentDrive & ¬
  "Special Tools:Special Fonts"
```

See also `openResFile` and `showResFile` commands

closeXlib command

Syntax `closeXlib {`*whichFile*`}`

Description This command closes the Xlibrary file specified by the string *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. HyperCard XCMDs and XFCNs can also be stored in Xlibrary files.

It is good practice to close any file you have opened as soon as you are finished using it.

Example This statement closes all open Xlibrary files:

```
closeXlib
```

This statement closes the Xlibrary VideoDisc Xlibrary when it is in the same foler as the movie:

```
closeXlib "VideoDisc Xlibrary"
```

This statement closes the Xlibrary Transporter XObjects in the folder New XObjects on the same disk as the movie. The disk is identified by the variable `currentDrive`:

```
closeXlib currentDrive & ¬
  "New XObjects:Transporter XObjects"
```

See also `openXlib` and `showXlib` commands

the colorDepth — property

Syntax `the colorDepth`

Description This property determines the color depth of the monitor on a color Macintosh. Using this property lets you adapt your movie to different color depths of different computer monitors.

Possible values are the following:

1	black-and-white
2	4 colors
4	16 colors
8	256 colors
16	32,768 colors
32	16,777,216 colors

When you assign a monitor a `colorDepth` higher than the monitor's color depth, the monitor becomes set to its maximum color depth.

For more than one monitor, `the colorDepth` property refers to the monitor that the stage is on. If the stage spans more than one monitor, `the colorDepth` indicates the greatest depth of those monitors; setting `the colorDepth` attempts to put all those monitors to the specified depth.

The `colorDepth` property can be tested and set. The default value is the value set in the Monitors control panel.

Example This statement makes playing the segment "Full color" dependent on whether the monitor color depth is set to 256 colors:

```
if the colorDepth = 8 then play movie "Full color"
```

Example This statement uses the `colorQD` function to check whether the monitor can display color, and then sets the color depth to 256 colors if it is:

```
if the colorQD = TRUE then set the colorDepth to 8
```

See also `colorQD` function; `switchColorDepth` property

---

the colorQD — function

Syntax

```
the colorQD
```

Description This function indicates whether the Color QuickDraw software is available.

- ◆ The `colorQD` is TRUE (1) for a color-capable Macintosh.
- ◆ The `colorQD` is FALSE (0) for a black-and-white-only machine.

Note *A machine capable of displaying color may not have it switched on. This command is best used in conjunction with colorDepth.*

Example This statement checks whether the Macintosh is color capable and plays the movie “Color Movie” if it is:

```
if the colorQD = TRUE then play movie "Color Movie"
```

This statement checks whether the Macintosh is color capable and sets the color depth to 256 colors if it is:

```
if the colorQD = TRUE then set the colorDepth to 8
```

See also `colorDepth` and `switchColorDepth` properties

the commandDown function

Syntax

```
the commandDown
```

Description

This function determines whether the Command key is being pressed.

- The `commandDown` function is TRUE when the Command key is being pressed.
- The `commandDown` function is FALSE when the Command key is not being pressed.

You can use `the commandDown` together with the element `the key` to determine when the Command key is pressed in combination with another key. This lets you create handlers that are executed when the user presses specified command-key combinations.

Note that Command key equivalents for Director's authoring menus take precedence while playing the movie, unless you have installed custom Lingo menus, or are playing a projector version of the movie.

Example

These statements have Lingo pause the movie whenever the user presses Command-p. By setting the `keyDownScript` property to `doCommandKey`, the `on startMovie` handler makes the `doCommandKey` handler the first event handler executed when a key is pressed. The `doCommandKey` handler checks whether the Command and p keys are pressed at the same time and pauses the movie if it is.

```
on startMovie
  set the keyDownScript to "doCommandKey"
end startMovie
on doCommandKey
  if (the commandDown) and (the key = "p") then pause
end
```

See also

`controlDown`, `the key`, `the keyCode`, `optionDown`, and `shiftDown` functions

constrainH function

Syntax `constrainH (`*whichSprite*`, `*integerExpression*`)`

Description This function evaluates *integerExpression*, and then gives a value that depends on the horizontal coordinates of the left and right edges of *whichSprite*.

- When the value is between the left and right coordinates, the value doesn't change.
- When the value is less than the left horizontal coordinate, the value is changed to the value of the left coordinate.
- When the value is greater than the right horizontal coordinate, the value is changed to the value of the right coordinate.

The `constrainH` and `constrainV` functions constrain only one axis each; the `constraint of sprite` property limits both. Note that this function does not change the sprite's properties.

Example These statements check the `constrainH` for sprite 1 when it has left and right coordinates of 40 and 60:

```
put constrainH(1, 20)
-- 40
put constrainH(1, 55)
-- 55
put constrainH(1, 100)
-- 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer goes past the edge of the gauge:

```
set the locH of sprite 1 to constrainH(2, the mouseH)
```

See also `constrainV` function; `constraint`, `left`, and `right` 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 *whichSprite*. When the `constraint of sprite` property is turned on, the sprite specified by *whichSprite* is constrained to the bounding rectangle of another sprite.

The `constraint of sprite` property affects moveable sprites, and the `locH` and `locV` sprite properties. The constraint point of a moveable sprite cannot be moved outside the bounding rectangle of the constraining sprite. (The constraint point for a bitmap sprite is the registration point. The constraint point for a shape sprite is the top left corner of the shape sprite.) When a sprite has a constraint set, the constraint limits override any `locH` and `locV` sprite property settings.

To remove a `constraint of sprite` property, set it to 0:

`set the constraint of sprite` *whichSprite* `to 0`

The `constraint of sprite` property can be tested and set. The default value is 0.

The `constraint of sprite` property is useful for constraining a moveable sprite to the bounding rectangle of another sprite. This is a way to simulate a “track” for a slider control or restrict where on the screen a user can drag an object in a game.

Example This statement constrains sprite (i + 1) to the boundary of sprite 14.

```
set the constraint of sprite (i + 1) to 14
```

This statement checks whether sprite 3 is constrained and activates the handler `showConstraint` if it is. (The operator `<>` performs the same operation as “not equal to.”)

```
if the constraint of sprite 3 <> 0 then ¬
  showConstraint
```

See also `constrainH` and `constrainV` functions; `locH` and `locV` sprite properties

constrainV function

Syntax `constrainV (`*whichSprite*`,` *integerExpression*`)`

Description This function evaluates *integerExpression*, and then gives a value that depends on the vertical coordinates of the top and bottom edges of the sprite specified by *whichSprite*.

- When the value is between the top and bottom coordinates, the value doesn’t change.
- When the value is less than the top coordinate, the value is changed to the value of the top coordinate.
- When the value is greater than the bottom coordinate, the value is changed to the value of the bottom coordinate.

Note that this function does not change the sprite properties.

Example These statements check the `constrainV` for sprite 1 when it has top and bottom coordinates of 40 and 60:

```
put constrainV(1, 20)
-- 40
put constrainV(1, 55)
-- 55
put constrainV(1, 100)
-- 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer goes past the edge of the gauge:

```
set the locV of sprite 1 to ¬
  constrainV(2, the mouseV)
```

See also `bottom of sprite`, `the constraint of sprite`, and `top of sprite` properties; `constrainH` function

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).

The `contains` comparison operator has a precedence level of 1.

The `contains` comparison operator is useful for checking whether the user types a specific character or string of characters. You can also use the `contains` operator to search one or more text fields for specific strings of text.

Note *The string comparison is not sensitive to case or diacritical marks; "a" and "Å" are treated the same.*

Example This statement determines whether a string from a text field contains only numeric input before converting it using `the value()` function. You can use this handler to test it:

```
on isNumber aLetter
  put "1234567890." into digits
  if digits contains aLetter then
    return TRUE
  else
    return FALSE
  end if
end isNumber
```

See also `offset` function; `starts` comparison operator

continue — command

Syntax `continue`

Description The `continue` command resumes playing the movie after a pause.

Example This statement has the movie resume playing when it is paused and the Return key is pressed:

```
set the keydownScript to "if the key = RETURN ¬
  then continue"
```

See also `delay` and `pause` commands; `pauseState` function

the controlDown — function

Syntax `the controlDown`

Description This function determines whether the Control key is being pressed.

- The `controlDown` function is TRUE when the Control key is being pressed.
- The `controlDown` function is FALSE when the Control key is not being pressed.

You can use `the controlDown` function together with the element `the key` to check for combinations of the Control key and another key.

Example This `keyDown` handler checks whether the key that is pressed is the Control key and activates the `doControlKey` handler if it is. The argument (the key) identifies which key was pressed in addition to the Control key.

```
on keyDown
  if the controlDown then doControlKey (the key)
end
```

See also `charToNum`, `commandDown`, `the key`, `the keyCode`, `optionDown`, and `shiftDown` functions

the controller of cast — digital video cast property

Syntax

`the controller of cast` *castName*

Description

A digital video movie cast member can be made to show or hide its controller with this cast property. Setting this property to 1 shows the controller; setting it to 0 hides it.

The digital video movie must be in `directToStage` playback mode in order to display the controller.

Example

This statement has the digital video cast member Demo show its controller:

```
set the controller of cast "Demo" to 1
```

See also

`the directToStage` cast property

copyToClipBoard — command

Syntax

`copyToClipBoard cast` *castMember*

Description

This command copies the specified cast member to the Clipboard. You can use this command to copy cast members between movies or applications. The cast window does not need to be the active window when you use the `copyToClipBoard` command.

Example

This statement copies the cast member named chair to the Clipboard:

```
copyToClipBoard cast "chair"
```

This statement copies cast member number 5 to the Clipboard:

```
copyToClipBoard cast 5
```

cos — function

Syntax `cos (`*angle*`)`

Description This function calculates the cosine of the specified angle. The angle must be expressed in radians.

Example The following statement calculates the cosine of pi ()/2 and displays it in the message window:

```
put cos (pi ()/2)
```

Note that the π symbol cannot be used in a Lingo expression.

count — function

Syntax `count (`*list*`)`

Description This function returns the number of entries in the specified list.

Example This statement displays the number 3, the number of entries:

```
put count ( [10, 20, 30] )
-- 3
```

the crop of cast — digital video cast property

Syntax `the crop of cast`

Description This cast property affects how the digital video cast member is displayed inside a sprite. It can be tested and set.

- When `the crop of cast` is FALSE the cast member is scaled—either stretched or shrunk—to fit inside the sprite rectangle.
- When `the crop of cast` is TRUE, the cast member is not scaled. It is cropped to fit inside the sprite rectangle.

Example This statement instructs Lingo to crop any sprite that refers to the digital video cast member Interview.

```
set the crop of cast "Interview" to TRUE
```

See also `the center of cast` digital video cast property

---

cursor — command

Syntax `cursor [`*castNumber, maskCastNumber*`]`

Syntax `cursor` *whichCursor*

Description This command changes the cast member that is used for a cursor. The `cursor` command stays in effect until you turn it off by setting the cursor to zero.

- Use the syntax `cursor [`*castNumber, maskCastNumber*`]` to specify the number of a cast member to use as a cursor and its optional mask. The hot spot of the cursor is the registration point of the cast member.

  The cast member that you specify must be a 1-bit cast member; it must be at least 16 by 16 pixels. If the cast member is larger, Director crops it to a 16 x 16 square, starting in the upper left corner of the image. If the cast member is smaller, draw a 17 x 17 square around it to achieve the proper dimensions when the image is cropped.

- Use the syntax `cursor` *whichCursor* to use the default cursors that are supplied by the system. The term *whichCursor* must be an integer that specifies the appearance of the cursor. The following values specify cursors:

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).

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 overrides the `cursor` command settings from the previous movie. Therefore, in a presentation using a custom cursor for multiple movies, store any special cursor resource number as a global variable. Global Lingo variables stay in memory between movies. This allows you to use the `cursor` command at the beginning of any new movie that is loaded.

Example This statement changes the cursor to a watch cursor whenever the value in the variable named `status` equals 1:

```
if status = 1 then cursor 4
```

See also `cursor of sprite` property; `openResFile` command

the cursor of sprite — sprite property

Syntax `the cursor of sprite` *whichSprite* `to [`*castNumber, maskCastNumber*`]`

Syntax `the cursor of sprite` *whichSprite* `to` *whichCursor*

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 that 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 of sprite` property is useful for changing the cursor when the mouse pointer is over specific regions of the screen. You can use this to indicate regions where certain actions are possible when the user clicks.

See the `cursor` command for information about using custom cursors.

The `cursor of sprite` property can be tested and set.

Example This statement executes the handler `setCursor` when no cursor is set for sprite 3:

```
if the cursor of sprite 3 = 0 then setCursor
```

This statement sets the cursor to an I-beam when the cursor is over sprite 3:

```
if rollover(3) then set the cursor of sprite 3 to 1
```

This statement sets the cursor to a custom cursor named grabber:

```
set the cursor of sprite (i + 1) to grabber
```

See also `cursor` and `openResFile` commands; `rollover` function

the date — function

Syntax `the abbr date`

Syntax `the abbrev date`

Syntax `the abbreviated date`

Syntax `the date`

Syntax `the long date`

Syntax `the short date`

Description This function gives the current date in the system clock in one of three formats: `abbreviated`, `long`, or `short`. If no format is specified, the default is short. The abbreviated format can also be referred to as `abbrev` and `abbr`.

Example This statement gives the abbreviated date:

```
put the abbreviated date
-- "Sat, Sep 7, 1991"
```

This statement gives the long date:

```
put the long date
-- "Saturday, September 7, 1991"
```

This statement gives the short date:

```
put the short date
-- "9/7/91"
```

Example This statement tests whether the current date is January 1 by checking whether the first four characters of the date are 1/1. If it is January 1, the alert “Happy New Year!” appears:

```
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. These examples are for the United States.*

See also `time` function

delay command

Syntax `delay` *numberOfTicks*

This command halts the movie for a given amount of time. The integer expression *numberOfTicks* specifies the number of ticks to wait. (There are 60 ticks per second.) The only interactivity possible during this time is halting Lingo entirely by pressing Command-Period (for example, mouse clicks are ignored).

The `delay` command works only when the playback head is moving. Place scripts using the `delay` command in either an `on enterFrame` or `on exitFrame` handler.

To mimic the behavior of a halt in a handler when the playback head is not moving, use the `startTimer` command and test for the `timer` property within a `repeat... while` loop.

Because it increases the time of individual frames, the `delay` command is useful for controlling the playback rate of a sequence of frames.

Example This handler delays the movie for 2 seconds when the playback head exits the current frame:

```
on exitFrame
  delay 2 * 60
end exitFrame
```

This handler, which could be placed in a frame script, delays the movie a random number of ticks:

```
on keydown
  if the key = RETURN then delay random(180)
end keyDown
```

See also `startTimer` command; `timer` property

delete — command

Syntax `delete` *chunkExpression*

Description This command deletes the specified chunk expression (character, word, item, or line) in any text container. Sources of text include fields (text cast members) and variables that hold strings.

Example This statement deletes the first word of line 3 in the text cast member Address:

```
delete word 1 of line 3 of field "Address"
```

This statement deletes the first character of the string in the variable `bidAmount`:

```
if char 1 of bidAmount = "$" then delete char 1 ¬
  of bidAmount
```

See also `char…of`, `field` keyword; `item…of`, `line…of`, `word…of` chunk expression keywords; `hilite` text property

deleteAt command

Syntax `deleteAt` *list*, *number*

Description This command deletes the item in the position specified by *number* from the list specified by *list*. The value *number* is the item's position in the order of the list.

Example This statement deletes the second item from the list named designers, which contains ["gee", "kayne", "ohashi"]:

```
set designers = ["gee", "kayne", "ohashi"]
deleteAt designers, 2
```

The result is the list ["gee", "ohashi"].

See also `addAt` command

deleteProp command

Syntax `deleteProp` *list*, *property*

Description This command deletes the item that has the specified property from the specified list. For linear lists, this is the same as the `deleteAt` command. When there are more than one of the same property, only the first property in the list is deleted.

Example This statement deletes the property color from the list [#height: 100, #width: 200, #color: 34, #ink: 15], which is called `spriteAttributes`:

```
deleteProp spriteAttributes, #color
```

The result is the list [#height: 100, #width: 200, #ink: 15].

See also `deleteAt` command

the depth of cast — cast property

Syntax `the depth of cast` *cast member*

Description This cast property gives the color depth of the bitmap cast member specified by *cast member*. Black and white is 1-bit color depth; 256 colors is 8-bit color depth; thousands of colors is 16-bit color depth; and millions of colors is 32-bit color depth.

This property can be tested, but not set from Lingo.

Example This statement determines the color depth of the cast member Shrine:

```
put the depth of cast "Shrine"
```

digitalVideo cast center

See `the center of cast` digital video cast property.

digitalVideo cast controller

See `the controller of cast` digital video cast property.

digitalVideo cast crop

See `the crop of cast` digital video cast property.

digitalVideo cast directToStage

See `the directToStage of cast` digital video cast property.

digitalVideo cast duration

See `the duration of cast` digital video cast property.

digitalVideo cast frameRate

See `the frameRate of cast` digital video cast property.

digitalVideo cast loop

See `the loop of cast` digital video cast property.

digitalVideo cast pausedAtStart

See `the pausedAtStart of cast` digital video cast property.

digitalVideo cast preload

See `the preload of cast` digital video cast property.

digitalVideo cast sound

See `the sound of cast` digital video cast property.

digitalVideo cast video

See `the video of cast` digital video cast property.

digitalVideo sprite movieRate

See `the movieRate of sprite` digital video sprite property.

digitalVideo sprite movieTime

See `the movieTime of sprite` digital video sprite property.

digitalVideo sprite startTime

See `the startTime of sprite` digital video sprite property.

digitalVideo sprite stopTime

See `the stopTime of sprite` digital video sprite property.

digitalVideo sprite volume

See `the volume of sprite` digital video sprite property.

the directToStage of cast — digital video cast property

Syntax `the directToStage of cast` *castName*

Description This property determines whether a digital video cast member plays in front of all other layers on the stage.

- When this property is set to 1, a digital video movie plays in front of all other layers.
- When this property is set to 0, a digital video movie cast member can appear in any layer of the stage's animation planes.

No cast members appear in front of a `directToStage` digital video movie. Also, ink effects do not affect the appearance of a `directToStage` digital video movie. Using this property may improve the playback performance of a digital video movie cast member.

Example This statement makes the digital video movie "The Residents" always play in the top layer of the stage:

```
set the directToStage of cast "The Residents" to 1
```

do — command

Syntax `do` *stringExpression*

Description This command evaluates *stringExpression* and executes the result 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.

Note *This command does not allow global variables to be declared. In earlier versions of Director, the statement would work. In Director 4, they do not.*

Example This statement performs the statement contained within quotes:

```
do "beep 2"
do getAt(commandList, 3)
```

done

See the `play done` command.

dontPassEvent — command

Syntax

```
dontPassEvent
```

Description This command prevents Lingo from passing an event message to subsequent locations in the message hierarchy.

The `dontPassEvent` command applies only to the current event being handled. It does not affect future events.

The `dontPassEvent` command applies only within primary event handlers or handlers that they call. It has no effect elsewhere.

Example This handler has the computer beep when the Tab or Enter key is pressed and keeps the message from passing on to subsequent locations in the message hierarchy:

```
on myKey
  if the key = TAB or the key = ENTER then beep
  dontPassEvent
end myKey
```

Example This statement makes `myKey` the primary event handler:

```
set the keyDownScript to "myKey"
```

When these are in effect at the same time, pressing the Tab or Enter key any time the movie is playing has the computer beep but not pass the keyDown message on to anywhere else in the movie.

See also `keyDownScript`, `keyUpScript`, `mouseDownScript`, `mouseUpScript`, and `timeOutScript` properties

the doubleClick — function

Syntax

```
the doubleClick
```

Description This function determines whether the last two mouse clicks were considered a double-click.

- The `doubleClick` function is TRUE if the last two mouse clicks were a double-click.
- The `doubleClick` function is FALSE if the last two mouse clicks were not a double-click.

Example This statement sends the playback head to the frame EnterBid when the user double-clicks the mouse button.

```
if the doubleClick then go to frame "EnterBid"
```

See also `the clickOn`, `the mouseDown`, and `the mouseUp` functions

down — keyword

See the `repeat with...down to` keyword.

the drawRect of window — window property

Syntax `the drawRect of window` *windowName*

Description This window property identifies the rectangular coordinates of a movie's window. The coordinates are given as a rect, with entries in the order left, top, right, and bottom.

This can be useful for scaling or panning movies.

The `drawRect of window` property can be tested or set.

Example This statement displays the current coordinates of the movie window called Control Panel.

```
put the drawRect of window "Control Panel"
-- rect(10, 20, 200, 300)
```

This statement sets the coordinates of the movie window to the values of the rect `movieRectangle`:

```
set the drawRect of window "Control Panel" ¬
  to movieRectangle
```

duplicate cast — command

Syntax `duplicate cast` *original*`{,` *new*`}`

Description This command makes a copy of the cast member specified by *original*. The optional *new* parameter specifies a specific cast window location for the duplicate cast member. If the *new* parameter isn't included, the duplicate cast member is placed in the first open cast window position.

Example This statement makes a copy of cast member Desk and places it in the first empty cast window position:

```
duplicate cast "Desk"
```

This statement makes a copy of cast member Desk and places it in cast window position 125:

```
duplicate cast "Desk", cast 125
```

the duration of cast — cast property

Syntax `the duration of cast` *castName*

Description This cast property is used with digital video cast members to determine the duration of the movie. This property cannot be set. The value returned for the movie's duration is in ticks (60ths of a second).

Example This statement sets the variable `HowLong` to the duration of the digital video cast member Demo:

```
put the duration of cast "Demo" into HowLong
```

the editableText of sprite — sprite property

Syntax `the editableText of sprite` *whichSprite*

Description This sprite property indicates whether a text sprite is editable.

- When text can be edited by the user, `the editableText of sprite` is TRUE.
- When the text cannot be edited by the user, `the editableText of sprite` is FALSE.

To use Lingo to make a text sprite editable, the sprite must first be a puppet sprite.

The `editableText of sprite` property lets you change whether text field can be edited as the movie plays. This lets you turn editable text on and off depending on current conditions in the movie.

You can also make a text cast member editable by using the Editable Text option in the Text Cast Member Info dialog box. You can make a text sprite editable by using the Edit Text option in the score.

The `editableText of sprite` property can be tested and set.

Example This handler first makes the text sprite a puppet and then makes it editable:

```
on myNotes
  puppetSprite 5, TRUE
  set the editableText of sprite 5 to TRUE
end
```

This statement checks whether a text sprite is editable and displays a message if it is:

```
if the editableText of sprite 13 = TRUE ¬
  then set the text of cast "Notice" to  "Please ¬
  enter your answer below."
```

else — keyword

See the `if...then` keyword.

EMPTY — character constant

Syntax

```
EMPTY
```

Description

This character constant represents the empty string, `" "`, a string with no characters.

You can scroll to a specific line in a scrolling text field by inserting `EMPTY` before the line.

Example

This statement erases all characters in the text cast member Notice by setting the field to `EMPTY`:

```
set the text of cast "Notice" to EMPTY
```

This statement does the same as the previous statement but uses a different form:

```
put EMPTY into field "Notice"
```

the enabled of menuItem — menu property

Syntax

`the enabled of menuItem` *whichItem* `of menu` *whichMenu*

Description

This menu item property determines whether the menu item specified by *whichItem* is displayed in plain text or dimmed, and whether it is selectable. The term *whichMenu* specifies the menu that contains the menu item.

- If `the enabled of menuItem` is TRUE, the menu item appears in plain text and is selectable.
- If `the enabled of menuItem` is FALSE, the menu item appears dimmed and is not selectable.

The expression *whichItem* can be either a menu item name or a menu item number. The expression *whichMenu* can be either a menu name or a menu number.

The enabled property can be tested and set. The default value is TRUE.

Example This handler enables or disables all the items in the specified menu. The argument `theMenu` specifies the menu; the argument `Setting` specifies TRUE or FALSE. For example, `ableMenu` (“Special”, FALSE) disables all the items in the Special menu:

```
on ableMenu theMenu, vSetting
  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 vSetting
    end repeat
end ableMenu
```

See also `checkMark`, `name`, `number`, and `script of menuItem` menu item properties; `name` and `number` menu properties

end — keyword

This keyword marks the end of handlers, methods, and multi-line control structures.

See also `if…then`, `method`, `on`, `repeat while`, and `repeat with` keywords; `on idle`, `on keyDown`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

ENTER — character constant

Syntax `ENTER`

Description This character constant represents the Enter key, which is marked "enter" on the Macintosh keyboard.

Example This statement checks whether the Enter key is pressed and sends the playback head to the frame addSum if it is:

```
on keyDown
  if the key = ENTER then go to frame "addSum"
end
```

enterFrame — event handler

See the `on enterFrame` event handler; `stepMovie` keyword.

erase cast — command

Syntax `erase cast` *whichCastmember*

Description This command deletes the specified cast member and leaves its cast window location empty.

Example This statement deletes the cast member named Gear:

```
erase cast "Gear"
```

This handler deletes cast members start through finish:

```
on deleteCast start, finish
  repeat with i = start to finish
    erase cast i
  end repeat
end on deleteCast
```

exit — keyword

Syntax `exit`

Description This keyword has Lingo leave a handler or factory method, and return to the place from where the handler or factory was called. When the handler is nested within another handler, Lingo returns to the main handler.

Example The first statement of this script checks whether the monitor is set to black and white, and exits if it is:

```
on setColors
  if the colorDepth = 1 then exit
  set the foreColor of sprite 1 to 35
end setColors
```

See also `abort` command; `on` and `method` keywords

exit repeat — keyword

Syntax `exit repeat`

Description This keyword has Lingo leave a repeat loop and go to the statement following the `end repeat` statement, but remain within the current handler or method.

The `exit repeat` keyword is useful for breaking out of a repeat loop when a specified condition—such as two values being equal or a variable being a certain value—exists.

Example This handler looks for the position of the first vowel in a string represented by the variable `testString`. As soon as the first vowel is found, the `exit repeat` command has Lingo leave the repeat loop and go to the statement `return i`:

```
on findVowel testString
  repeat with i = 1 to the number of chars ¬
    in testString
    if "aeiou" contains letter then exit repeat
  end repeat
  return i
end findVowel
repeat while and repeat with keywords
```

exitFrame — event handler

See the `on exitFrame` event handler.

the exitLock — property

Syntax `the exitLock`

Description This property determines whether the user can quit to the Finder from projectors.

- When `the exitLock` is FALSE, the user can quit to the Finder by pressing Command-period, Command-q or Command-w.
- When `the exitLock` is TRUE, the user cannot quit to the Finder by pressing Command-period or Command-w.

The `exitLock` property can be tested and set. The default value is FALSE.

Example This statement sets the `exitLock` property to TRUE:

```
set the exitLock to TRUE
```

This handler checks whether Command-period or Command-q was pressed and whether `the exitLock` is set so that the user cannot exit to the Finder. When this is the case, the playback head goes to the frame "quit sequence," which could provide an alternative way to exit the movie:

```
on checkExit
  if the commandDown and ¬
  (the key = "." or the key = "q") and ¬
  the exitLock = TRUE then go to frame "quit
sequence"
end checkExit
```

exp — function

Syntax `exp(`*integer*`)`

Description This function calculates e, the natural logarithm base, to the power specified by *integer*.

Example The following statement calculates the value of e to the exponent 5:

```
put exp(5)
-- 148.4132
```

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 have the same message syntax as XObjects.

In Director 4, it is recommended that you use lists and parent scripts rather than factories. Lists and parent scripts are a simpler way of achieving the same result.

See Appendix C in *Using Lingo* for more information about factories, objects, methods, and instance variables.

See also

`factory` function; `instance`, `me`, and `method` keywords; `mDispose`, `mGet`, `mInstanceRespondsTo`, `mNew`, `mPerform`, `mPut`, and `mRespondsTo` predefined methods

factory function

Syntax `factory(`*factoryName*`)`

Description This function identifies the factory or XObject specified by *factoryName*. If no factory or XObject with the given name is found, the factory function value is 0. If the factory is found, the object is returned. You can use the `objectP()` function to test the return value.

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 the `sound fadeIn` command.

fadeOut

See the `sound fadeOut` command.

FALSE — logical constant

Syntax `FALSE`

Description This logical constant applies to an expression that is logically FALSE, such as 2 > 3. When treated as a number value, FALSE has the numerical value of 0.

Example This statement turns off the `soundEnabled` property by setting it to FALSE:

```
set the soundEnabled to FALSE
```

See also `TRUE` logical constant

field — keyword

Syntax `field` *whichField*

Description This keyword refers to the text in a text cast member. It is equivalent to the `text of cast` property.

The text cast member is specified by *whichField*.

- When *whichField* is a string, it is used as the cast name.
- When *whichField* is an integer, it is used as the cast number.

Text can be read from or put into the field. You can also use chunk expressions with text fields.

Example This statement puts the characters 5 through 10 of the field name entry into the variable `myKeyword`:

```
put char 5 to 10 of field "entry" into myKeyword
```

This statement checks whether the user entered the word "desk" and goes to the frame "deskBid" if he or she did:

```
if field "bid" contains "desk" then go to "deskBid"
```

See also `cast` keyword; `char...of`, `item...of`, `line...of`, and `word...of` chunk expression keywords

the fileName of cast — cast property

Syntax `the fileName of cast` *cast member*

Description This cast property refers to the name of the file assigned to the linked cast member specified by *cast member*. This is useful for switching which external linked file is assigned to a cast member while the movie plays, similar to the way you can switch cast members. When the linked file is in a different folder than the movie, you must include the file's pathname.

The `fileName of cast` property can be tested and set. After the filename is set, Director uses that file the next time the cast member is used.

Example This statement makes the digital video movie "ChairAnimation" the linked file assigned to cast member 40:

```
set the fileName of cast 40 = ChairAnimation
```

the fileName of window — window property

Syntax `the fileName of window` *whichWindow*

Description This window property refers to the filename of the movie assigned to the window specified by *whichWindow*. When the linked file is in a different folder than the movie, you must include the file's pathname.

You assign a movie to a window by setting `the fileName of window` for the window to the movie's filename. This is required before you can play the movie in the window.

The `fileName of window` property can be tested and set.

Example This statement assigns the file named Control Panel to the window named Tool Box:

```
set the fileName of window "Tool Box" = ¬
  "Control Panel"
```

This statement displays the filename of the file assigned to the window named Navigator:

```
put the fileName of window "Navigator"
```

findEmpty — function

Syntax `findEmpty(cast` *castNum*`)`

Description This function returns the next empty cast position after and including the specified *castNum*.

Example This statement finds the first empty cast member on or after cast member 100:

```
put findEmpty(cast 100)
```

findPos — function

Syntax `findPos(`*list*`,` *prop*`)`

Description This function identifies which position the property specified by *property* holds in the property list specified by *list*.

The `findPos` command does the same thing as the `findPosNear` command, except that the result of the `findPos` command is <VOID> when the specified property is not in the list.

Example This statement identifies the position of the property c in the list Answers, which consists of [a:#10, b:#12, c:#15, d:#22]:

```
findPos(Answers, #c)
```

The result is 3, because c is the third property in the list.

See also `findPosNear` function

findPosNear — function

Syntax

`findPosNear(`*list*`,` *prop*`)`

Description

This function identifies which position the property specified by *property* holds in the property list specified by *list*.

The `findPosNear` command does the same thing as the `findPos` command, except that when the specified property is not in the list, the `findPosNear` command identifies the postion of the closest property in the list, based on the sort order. This would be useful in finding the closest name in a sorted directory of names.

Example

This statement identifies the position of a property in the sorted list Answers, which consists of [#Nile:2, #Pharaoh:4, #Raja:0]:

```
findPosNear(Answers, #Ni)
```

The result is 1, because Ni is closest to Nile, the first property in the list.

See also

`findPos` command

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.

The `fixStageSize` property cannot change the stage size for a movie that is currently playing. This property is primarily used for movies played back with the player.

The `fixStageSize` property can be tested and set. The default value is TRUE.

Example This statement determines if the `fixStageSize` property is turned on, and sends the playback head to a specified frame if it is:

```
if the fixStageSize = FALSE then ¬
  go to frame "proper size"
```

This statement sets the `fixStageSize` property to the opposite of its current setting:

```
set the fixStageSize to (not the fixStageSize)
```

See also `centerStage` property

float — function

Syntax `float(`*expression*`)`

Description This function converts an expression to a floating-point number. The number of digits that follow the decimal point is set using the `the floatPrecision` property.

Example This statement converts the integer 1 to floating-point 1.

```
put float(1)
-- 1.0
```

See also `the floatPrecision` property

floatP — function

Syntax `floatP(`*expression*`)`

Description This function indicates whether the value specified by *expression* is a floating-point number.

- The `floatP` function is TRUE (1) if *expression* is a floating-point number.
- The `floatP` function is FALSE (1) if *expression* is not a floating-point number.

The “P” in `floatP` stands for “predicate.”

Example This statement tests whether 3.0 is a floating-point number. The message window displays the number 1, indicating that it is TRUE:

```
put floatP(3.0)
-- 1
```

This statement tests whether 3 is a floating-point number. The message window displays the number 0, indicating that it is FALSE:

```
put floatP(3)
-- 0
```

See also `float`, `integerP`, `objectP`, `stringP`, and `symbolP` functions

the floatPrecision — property

Syntax `the floatPrecision to` *integer*

Description This system property rounds off the display of floating-point numbers to the number of decimal places specified by *integer*. The maximum is 19 significant digits.

The `floatPrecision` property determines only the number of digits used to display floating-point numbers. The number of digits used to perform calculations doesn't change.

The `floatPrecision` property can be tested and set. The default value is 4.

Example This statement rounds off the square root of 3.0 to three decimal places:

```
set the floatPrecision to 3
put sqrt(3.0) into x
put x
-- 1.732
```

Example This statement rounds off the square root of 3.0 to eight decimal places:

```
set the floatPrecision to 8
put x
-- 1.73205081
```

the foreColor of cast — cast property

Syntax `set the foreColor of cast` *castName* `to` *colorNumber*

Description This cast property sets the foreground color of a text cast member.

Example This statement changes the color of the text in cast member 1 to the color in palette entry 250.

```
set the foreColor of cast 1 to 250
```

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 *whichSprite*. Setting `the foreColor of sprite` sprite property in a Lingo script is equivalent to choosing the foreground color from the tools window when the sprite is selected on the stage.

The foreground color applies only to 1-bit bitmap and shape cast members. It does not affect the display of a text or button cast member. An 8-bit, 16-bit, or 24-bit bitmap is affected, but generally not in a useful way.

The value of a sprite's background color ranges from 0 to 255 for 8-bit color, and from 0 to 15 for 4-bit color. The numbers correspond to the index number of the background color in the current palette. (A color's index number appears in the color palette's lower left corner when you click the color.)

Changing a sprite's foreground color during a `mouseDown` is a useful way to indicate when a sprite is clicked.

When you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command 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.

The `foreColor of sprite` property can be tested and set, although in order to set it with Lingo the sprite must be a puppet.

Example The following statement sets the variable `oldColor` to the foreground color of sprite 5:

```
put the foreColor of sprite 5 into oldColor
```

The following statement makes 36 the number for the foreground color of a random sprite from sprite 11 to sprite 13:

```
set the foreColor of sprite (10 + random(3)) to 36
```

---

forget window — command

Syntax `forget window` *whichWindow*

Description This command tells Lingo to close and delete the window specified by *window* when the window is no longer in use and no other variables refer to it.

Example This statement has Lingo delete the window Control Panel when the movie no longer uses the window:

```
forget window "Control Panel"
```

the frame — function

Syntax

```
the frame
```

Description This function refers to the current frame in the current movie.

Example This statement sends the playback head to the frame before the current frame:

```
go to (the frame - 1)
```

See also `label` and `marker` functions

the frameLabel — frame property

Syntax

```
the frameLabel
```

Description This frame property identifies the label assigned to the current frame. When the current frame has no label, the value of the `frameLabel` property is an empty string (" ").

The `frameLabel` property can be tested. However, you cannot use the `frameLabel` to assign a label.

Example This statement checks the label of the current frame. In this case, the current `frameLabel` is Start:

```
put the frameLabel
-- "Start"
```

See also `the movieRate of sprite`, `the movieTime of sprite` sprite properties

the framePalette — frame property

Syntax `the framePalette`

Description This frame property identifies the cast member number of the palette used in the current frame.

The `framePalette` property can be tested. However, you cannot assign a palette by using the `framePalette` property.

Example This statement checks the palette used in the current frame. In this case, the palette is cast member 45:

```
put the framePalette
-- 45
```

See also `puppetPalette` command

the frameRate of cast — digital video cast property

Syntax `the frameRate of cast` *DVcast member*

Description This digital video cast property specifies the frame rate that the digital video movie specified by *DVcast member* is played. The possible values for `the frameRate of cast` correspond to the radio buttons for selecting digital video playback options.

- When `the frameRate of cast` is between 0 and 255, the digital video movie plays every frame at that frame rate. The `frameRate of cast` property cannot be greater than 255.
- When `the frameRate of cast` is set to 0, the digital video movie plays at its normal setting, as if you had the Play Every Frame checkbox unchecked in the Cast Info dialog box.
- When `the frameRate` is set to –1, the digital video movie plays every frame at its normal rate.
- When `the frameRate` is set to –2, the digital video movie plays every frame as fast as possible.

Example This statement sets the frame rate of the digital video cast member Rotating Chair to 30 frames per second:

```
set the frame rate of cast "Rotating Chair" to 30
```

This statement has the digital video cast member Rotating Chair play every frame as fast as possible:

```
set the frame rate of cast "Rotating Chair" to -2
```

See also `movieRate of sprite`, `movieTime of sprite` digital video sprite properties

---

the frameScript — frame property

Syntax

```
the frameScript
```

Description This frame property identifies the number of the frame script assigned to the current frame.

The `frameScript` property can be tested. However, you cannot assign a script by using the `frameScript` property.

Example This statement dispalys the number of the script assigned to the current frame. In this case, the script number is 25:

```
put the frameScript
-- 25
```

---

the framesToHMS — function

Syntax `the framesToHMS(`*frames*`,` *tempo*`,` *dropFrame*`,` *fractionalSeconds*`)`

Description This function converts the specified number of frames to their equivalent length in hours, minutes, and seconds. This is useful for predicting the actual playtime of a movie or controlling a video playback device.

- 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. Normally, this is FALSE. This argument is meaningful only if FPS is set to 30 frames per second. (Drop frame is a method of compensating for the color NTSC frame rate which is not exactly 30 frames per second.)
- The *fractionalSeconds* argument determines what happens to residual frames. When TRUE replaces *fractionalSeconds*, the residual frames are converted to the nearest hundredth of a second. When FALSE replaces *fractionalSeconds*, 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 This statement converts a 2710-frame, 30 frame-per-second movie. The *dropFrame* and *fractionalSeconds* arguments are both turned off:

```
put framesToHMS(2710, 30, FALSE, FALSE)
-- " 00:01:30.10 "
```

See also `HMStoFrames` function

the frameTempo — frame property

Syntax `the frameTempo`

Description This frame property indicates the tempo assigned to the current frame.

The `frameTempo` property can be tested. However, you cannot set the tempo by using the `framePalette` property.

Example This statement checks the tempo used in the current frame. In this case, the tempo is 15 frames per second:

```
put the frameTempo
-- 15
```

See also `puppetTempo` command

the freeBlock — function

Syntax `the freeBlock`

Description This function indicates the size of the largest free contiguous block of memory, in bytes. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes. In order to load a cast member, you need a free block at least as large as the cast member.

Example This statement determines whether the largest contiguous free block is smaller than 10K, and displays an alert if it is:

```
if the freeBlock < 10 * 1024 then ¬
  alert "Not enough memory!"
```

See also `freeBytes` and `memorySize`, and `ramNeeded` functions; `the size of cast` cast property

the freeBytes — function

Syntax `the freeBytes`

Description This function indicates the total number of bytes of free memory, which may not be contiguous. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

This function differs from `freeBlock`, because it reports all free memory, not just contiguous memory.

Example This statement checks whether more than 200K of memory is available, and plays a color movie if it is:

```
if the freeBytes > 200 * 1024 then ¬
  play movie "colorMovie"
```

See also `freeBlock`, `memorySize`, and `RamNeeded` functions; `the size of cast` cast property

getaProp function

Syntax `getaProp(`*list*, *positionOrProperty*`)`

Description This function identifies the value associated with the position or value specified by *positionOrProperty* in the list specified by *list*.

- When the list is a linear list, the result is the value at the position specifed by *positionOrProperty*.
- When the list is a property list, the result is the value associated with the property specified by *positionOrProperty*.

The `getaProp` command gives the result <VOID> when the specified value is not in the list.

When used with linear lists, the `getaProp` command does the same as the `getAt` command.

Example This statement identifies the value in the third position of the linear list Answers, which consists of [10, 12, 15, 22]:

```
getaProp(Answers, 3)
```

The result is 15, because 15 is the third value in the list.

This statement identifies the value associated with the property 15 in the property list Answers, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
getaProp(Answers, #c)
```

The result is 15, which is the value associated with property c.

See also `getOne` and `getProp` functions

getAt function

Syntax `getAt(`*list*`, `*position*`)`

Description This function identifies the value in the position specified by *position* in the list specified by *list*.

This function does the same as the `getaProp` command for linear lists.

Example This statement has the message window display the third item in the list Answers, which consists of [10, 12, 15, 22]:

```
getAt(Answers, 3)
```

The result is `15`.

See also `getaProp` function

getLast function

Syntax `getLast(`*list*`)`

Description This function identifies the last value in the list specified by *list*.

Example This statement identifies the last item in the list Answers, which consists of [10, 12, 15, 22]:

```
put getLast(Answers)
```

The result is 22.

This statement identifies the last item in the list Bids, which consists of [#Gee:750, #Kayne:600, #Ohashi:850]:

```
put getLast(Bids)
```

The result is 850.

getNthFileNameInFolder function

Syntax `getNthFileNameInFolder(`*folderPath, fileNumber*`)`

Description This function returns a fileName from the directory folder at the specified path and number within the folder. If the function returns an EMPTY string, you have specified a higher number than there are files in the folder.

Example The following handler returns a list of fileNames in the folder at the current path. To call the function, use parentheses after the handler name in the calling statement, as in `put currentFolder()`:

```
on currentFolder
  put [] into fileList
  repeat with i = 1 to the maxInteger
    put getNthFileNameInFolder(the pathName, i) ¬
      into n
    if n = EMPTY then exit repeat
    append(fileList, n)
  end repeat
  return fileList
end currentFolder
```

Note *To specify other folder names, use the full path, ending with a colon. For example:*

`"HardDisk:myOuterFolder:myInnerFolder:"`

To look for files on the DeskTop, use the following path:

`"HardDisk:Desktop Folder:"`

getOne function

Syntax `getOne(`*list, value*`)`

Description This function identifies the position or property associated with the value specified by *value* in the list specified by *list*.

- When the list is a linear list, the result is the value's position in the list.
- When the list is a property list, the result is the property associated with the value in the list.

For values in the list more than once, only the first occurrence is displayed. The `getOne` function gives the result 0 when the specified value is not in the list.

When used with linear lists, the `getOne` function does the same as the `getPos` command.

Example This statement identifies the position of the value 12 in the linear list Answers, which consists of [10, 12, 15, 22]:

```
getOne(Answers, 12)
```

The result is 2, because 12 is the second value in the list.

This statement identifies the property associated with the value 12 in the property list Answers, which consists of [#a:10, # b:12, # c:15, #d:22]:

```
getOne(Answers, 12)
```

The result is b, which is the property associated with the value 12.

See also `getPos` function

getPos function

Syntax `getPos(`*list, value*`)`

Description This function identifies the position of the value specified by *value* in the list specified by *list*. When the specified value is not in the list, the `getPos` command gives the value 0.

For values in the list more than once, only the first occurrence is displayed. This command does the same as the `getOne` command when used for linear lists.

Example This statement identifies the position of the value 12 in the list Answers, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
getPos(Answers, 12)
```

The result is 2, because 12 is the second value in the list.

See also `getOne` function

getProp function

Syntax `getProp(`*list, property)*

Description This function identifies the value associated with the property specified by *property* in the property list specified by *list*.

The `getProp` function is identical to the `getaProp` command, except that the `getProp` function displays an error message when the specified property is not in the list.

Example This statement identifies the value in the third position of the linear list Answers, which consists of [10, 12, 15, 22]:

```
getProp (Answers, 3)
```

The result is 15, because 15 is the third value in the list.

See also `getOne` function

getPropAt function

Syntax `getPropAt`*(list, index)*

Description This function identifies the property name associated with the position specified by *index* in the property list specified by *list*.

Example This statement displays the second property in the given list.

```
put getPropAt ([#a:10, #b:20],2)
-- #b
```

global keyword

Syntax `global` *variable1*`{,` *variable2*`}{,` *variable3*`}`...

Description This keyword identifies a variable as a global variable so that it can be shared by other handlers or movies.

A global variable can be declared by a script, a handler, or a method, and its value can be used by other scripts, handlers, and methods.

To use a global variable inside a handler, you must declare it to be a global variable; otherwise the handler assumes it is a local variable.

It is important to declare all the global variables within each handler or method that uses the global variable. Otherwise, any variable the handler uses is a local variable, even if it has been declared a global variable by another handler.

For further information, see Variables in Chapter 3 in *Using Lingo*.

Example

```
global startingPoint
set startingPoint = whichMenu
```

See also `showGlobals`, `property` command

go — command

Syntax `go {to} {frame}` *whichFrame*

Syntax `go {to} movie` *whichMovie*

Syntax `go {to} {frame}` *whichFrame* `of movie` *whichMovie*

Description This command causes the playback head to jump to the frame specified by *whichFrame* of the movie specified by *whichMovie*. The expression *whichFrame* can be a marker label or an integer frame number. The expression *whichMovie* must specify a movie file. (If the movie is in another folder, *whichMovie* must specify the pathname.)

It's better to refer to marker labels instead of frame numbers, because editing a movie 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.

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 to suspend the handler while playing the movie, use the `play` command.

The following are reset when loading a movie: `the beepOn`, the `constraint` properties, `the keyDownScript`, `the mouseDownScript`, `the mouseUpScript`; the `cursor of sprite` and `immediate of sprite` properties; the `cursor` and `puppetSprite` commands; and custom menus. However, `the timeoutScript` is not reset when loading a movie.

Example This statement sends the playback head to the marker named start:

```
go to "start"
```

This statement sends the playback head to the marker named Memory in the movie named *Noh Tale to Tell*:

```
go to frame "Memory" of movie "Noh Tale to Tell"
```

See also `label`, `marker`, and `the pathName` functions; `play` command

go loop — command

Syntax `go loop`

Description This command has the movie loop marker. It is equivalent to the statement `go to the marker(0)` that was used in earlier versions of Lingo.

Example This statement has the movie loop in the marker:

```
go loop
```

See also `go`, `go next, go previous` commands

go next — command

Syntax `go next`

Description This command sends the playback head to the next marker in the movie. It is equivalent to the statement `go marker(1)` that was used in earlier versions of Lingo.

Example This statement sends the playback head to the next marker in the movie:

```
go next
```

See also `go`, `go loop`, `go previous` commands

go previous — command

Syntax

```
go previous
```

Description

This command sends the playback head to the previous marker in the movie. It is equivalent to the statement `go marker(-1)` that was used in earlier versions of Lingo.

Example

This statement sends the playback head to the previous marker in the movie:

```
go previous
```

See also

`go`, `go loop`, `go next` commands

H

halt — command

Syntax `halt`

Description This command has Lingo exit the current handler and any handler that called it. After exiting all handlers, the `halt` command then stops the movie.

Example This statement checks whether the amount of free memory is less than 50K, and if it is, exits all handlers that called it, and then stops the movie:

```
if the freeBytes < 50*1024 then halt
```

See also `abort` and `pass` commands; `exit` keyword

the height of cast — cast property

Syntax `the height of cast` *whichCastmember*

Description This cast property determines the height in pixels of the cast member specified by *whichCastmember*.

The `height of cast` property can be tested, but not set.

Example This statement assigns the height of cast member 50 to the variable `vHeight`:

```
put the height of cast 50 into vHeight
```

See also `spriteBox` command; `the width of cast` property; `height of sprite` and `width of sprite` sprite properties

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 *whichSprite*. The height applies only to bitmap and shape cast members. It does not affect text or button cast members.

Setting this property does not have any effect on bitmap sprites unless the sprite's stretch property is set to TRUE. In order to set this property with Lingo, the sprite must be a puppet.

When you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. When you are changing several sprite properties—or several sprites—you only have to use the `updateStage` command once at the end of all the changes.

The `height of sprite` property can be tested and set.

Example This statement sets the height of sprite 10 to 26 pixels:

```
set the height of sprite 10 to 26
```

This statement assigns the height of sprite (i + 1) to the variable `vHeight`:

```
put the height of sprite (i + 1) into vHeight
```

See also `height of cast`, `stretch of sprite`, `width of sprite`, and `width of cast` sprite properties; `spriteBox` command

hilite — command

Syntax `hilite` *chunkExpression*

Description This command highlights (selects) the specified chunk in a text sprite. You can select any chunk that Lingo lets you define, such as a character, word, or line. The color that highlights text is the highlight color set in Color in the Control Panels.

Example

This statement highlights the fourth word in the text cast member Comments, which contains the string “Thought for the Day”:

```
hilite word 4 of field Comments
```

See also

`char…of`, `item…of`, `line…of`, and `word…of` chunk expression keywords; `delete` command; `mouseChar`, `mouseLine`, and `mouseWord` integer functions; `field` keyword; `selEnd` and `selStart` text properties

the hilite of cast — button property

Syntax

`the hilite of cast` *whichCastmember*

Description

This button property determines whether a checkbox or radio button sprite is selected.

- When `the hilite of cast` is TRUE, the checkbox or radio button is selected.
- When `the hilite of cast` is FALSE, the checkbox or radio button is deselected.

When *whichCastmember* is a string, it is used as the cast name. When *whichCastmember* is an integer, it is used as the cast number.

The `hilite of cast` button property can be tested and set. The default value is FALSE.

Example

This statement checks whether the button named 2400 baud is selected and sets the baud rate to 2400 if it is:

```
if the hilite of cast "2400 baud" = TRUE then ¬
  setBaudRate(2400)
```

This statement uses Lingo to select the button cast member powerSwitch by setting `the hilite of cast` for the cast member to TRUE:

```
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 movies measured in 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	is a dash (–) if the time is less than zero, or a space if the time is greater than or equal to zero
HH	represents number of hours
MM	represents number of minutes
SS	represents number of seconds
FF	represents fraction of a second if *fractionalSeconds* is TRUE. FF represents frames if *fractionalSeconds* is FALSE
D	is the letter "d" if *dropFrame* is TRUE. D is a space if *dropFrame* is FALSE

- The expression *tempo* specifies the tempo in frames per second.
- The *dropFrame* argument is a logical expression. When TRUE replaces *dropFrame*, it is a drop-frame. When FALSE replaces *dropFrame*, it is not. When the string *hms* ends in a "d", the time is treated as a drop-frame, regardless of the value of *dropFrame*.
- The *fractionalSeconds* argument determines the meaning of the fractional seconds. When it is set to TRUE, the numbers after the seconds specify a fraction of a second, to the nearest hundredth of a second. When it is set to FALSE, the numbers after the seconds specify the number of residual frames.

Example This statement determines the number of frames in a 1-minute, 30.1-second movie when the tempo is 30 frames per second. The *dropFrame* and *fractionalSeconds* arguments are both turned off:

```
put HMStoFrames(" 00:01:30.10 ", 30, FALSE, FALSE)
-- 2710
```

See also `framesToHMS` function

idle — event handler

See the `on idle` event handler.

if — keyword

Syntax `if` *logicalExpression* `then` *then-statement*

Syntax `if` *logicalExpression* `then` *then-statement*
`else` *else-statement*
`end if`

Syntax `if` *logicalExpression* `then`
    statement(s)
`end if`

Syntax `if` *logicalExpression* `then`
    statement(s)
`else`
    statement(s)
`end if`

Syntax `if` *logicalExpression*`1 then`
    statement(s)
`else if` *logicalExpression*`2 then`
    statement(s)
`else if` *logicalExpression*`3 then`
    statement(s)
`end if`

Description The if…then structure evaluates the *logicalExpression* specified by *logicalExpression*.

- When the condition is TRUE, Lingo executes the statement(s) that follow `then`.
- When it is FALSE, Lingo executes the statement(s) following `else`. If no statements follow `else`, Lingo exits the `if...then` structure.

The `else` portion of the statement is optional. If you need to have more than one *then-statement* or *else-statement*, you must end with the form `end if`.

When you use `else`, it always corresponds to the previous `if` statement. This means that sometimes you need to include an `else nothing` statement to associate an `else` keyword with the proper `if` keyword.

Example This statement checks whether the Return key was pressed, and then continues if it was:

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

This statement checks whether the color QuickDraw software is available. If it is available, Lingo plays the movie “Color Movie.” If the color QuickDraw software isn’t available, Lingo plays the movie “Black & White Movie”:

```
if the colorQD = TRUE then play "Color Movie"
else play "Black & White Movie"
```

This handler checks whether the “q” key was pressed, and then executes the subsequent statements if it was:

```
on keyDown
  if (the commandDown) and (the key = "q") then
    cleanUp
    quit
  end if
end keyDown
```

ilk function

Syntax `ilk(`*item*`,` *type*`)`

Description This function checks the type of lists, rects, and points by matching *item* with *type* and returning TRUE (1) or FALSE (0).

item	possible type				
	#list	#linearlist	#propertylist	#point	#rect
linear list	1	1	0	0	0
property list	1	0	1	0	0
point	1	0	0	1	0
rect	1	0	0	0	1

Example This statement identifies whether the list named bids is a property list and displays the result in the message window:

```
put ilk(bids, #property)
```

Because the list is a property list the message window displays 1, which is the numeric equivalent of TRUE.

This statement identifies whether the variable `vTotal` is a list and displays the result in the message window:

```
put ilk(vTotal, #list)
```

Because the variable is not a list, the message window displays 0, which is the numeric equivalent of FALSE.

ilk list

See the `ilk` function.

ilk point

See the `ilk` function.

ilk rect

See the `ilk` function.

importFileInto — command

Syntax `importFileInto` *castMember*, *fileName*

Description This command replaces the content of the cast member specified by *cast member* with the file specified by *fileName*.

Example This statement replaces the content of the sound cast member Memory with the sound file Wind:

```
importFileInto cast "Memory", "Wind"
```

in

See the `number of chars in`, `number of items in`, `number of lines in`, and `number of words in` functions.

inflate rect — function

Syntax `inflate (rectangle,` *widthChange*`,` *heightChange*`)`

Description This function changes the dimensions of the rectangle specified by *rectangle*. The change is relative to the center of the rectangle.

- The *widthChange* parameter specifies the number of the rectangle changes horizontally.
- *The heightChange* parameter specifies how much the rectangle changes vertically.

Values less than 0 for horizontal or vertical reduce the rectangle's size.

Example This statement increases the width of the rectangle by 4 pixels and the height by 2 pixels:

```
put inflate (Rect(10, 10, 20, 20), 2, 1)
-- Rect (8, 9, 22, 21)
```

This statement increases both the height and width of the rectangle by 20 pixels:

```
put inflate (Rect(0, 0, 100, 100), 10, 10)
-- Rect (-10, -10, 110, 110)
```

the ink of sprite — sprite property

Syntax `the ink of sprite` *whichSprite*

Description This sprite property determines the ink effect applied to the sprite specified by *whichSprite*.

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 background transparent (ink effect 36), you set the color that becomes transparent by selecting the color from the background color chip in the tools window while the sprite is selected in the score. You can do the same thing by using Lingo to set the `backColor` property, but this is unpredictable when the sprite has more than 1-bit color.

If you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. If you change several sprite properties—or several sprites—you need to use only one `updateStage` command at the end of all the changes.

For further information about ink effects, see *Using Director*.

The `ink` sprite property can be tested and set. To change any sprite property using Lingo, the sprite must be a puppet.

Example This statement changes the variable `currentInk` to the value for the ink effect of sprite 3:

```
put the ink of sprite 3 into currentInk
```

This statement gives sprite (i + 1) a matte ink effect by setting the `ink effect of sprite` property to 8, which specifies matte ink:

```
set the ink of sprite (i + 1) to 8
```

See also `the backColor of sprite` and `the foreColor of sprite` sprite properties

---

inside — function

Syntax `inside(`*point, rectangle*`)`

Description This function indicates whether the point specified by *point* is within the rectangle specified by *rectangle*.

- When the point is within the rectangle, the `inside` function is TRUE.
- When the point is outside the rectangle, the `inside` function is FALSE.

Example This statement indicates whether the point Center is within the rectangle Zone and displays the result in the message window:

```
put inside(Center, Zone)
```

See also `map`, `mouseH`, `mouseV`, and `point` functions

inside point

See the `inside` function.

installMenu command

Syntax `installMenu` *cast member*

Description This command installs the menu defined in the text cast member specified by *cast member*. These custom menus appear only while the movie is playing. To remove the custom menus, use the `installMenu` command with no argument, or with 0 as the argument.

For an explanation of how menu items are defined in a text cast member, refer to the `menu:` keyword.

Example This statement installs the menu defined in text cast member 37:

```
installMenu 37
```

This statement installs the menu defined in the text cast member named Menubar:

```
installMenu cast "Menubar"
```

This statement disables menus that were installed by the `installMenu` command:

```
installMenu 0
```

See also `menu:` keyword

instance — keyword

Syntax `instance` *variable1*`{,` *variable2*`}{,` *variable3*`}` ...

Description This keyword makes a variable an instance variable—a special kind of variable used with factories. A factory can assign instance variables to specific objects. Instance variables contain unique 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. An instance variable's value 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 that persist as long as the object persists.

In Director 4, it is recommended that you use lists and parent scripts rather than factories. Lists and parent scripts are a simpler way of achieving the same result.

To use an instance variable within a factory method, you must declare it an instance variable by using the `instance` keyword; otherwise the factory assumes it is a local 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.

For further information, see the section "Variables" in Chapter 3 in *Using Lingo*.

Example This handler uses the `instance` keyword in the `mNew` method to create instance variables in a factory:

```
method mNew parameter1, parameter2
  instance variable1, variable2
  put parameter1 into variable1
  put parameter2 into variable2
end mNew
```

This handler also creates instance variables:

```
method mNew
  global counter
  instance mySpeed, mySprite
  put 0 into mySpeed
  put counter into mySprite
end mNew
```

See also `factory`, `global`, and `method` keywords

integer — function

Syntax `integer(`*numericExpression*`)`

Description This function rounds the value of *numericExpression* to the nearest whole integer.

You can force an integer to be a string by using the `string()` function.

Example This statement rounds off the number 3.75 to the nearest whole integer:

```
put integer(3.75)
-- 4
```

Example This statement rounds off the value in parentheses. This provides a usable value for `the locH of sprite`, which requires an integer:

```
set the locH of sprite 1 ¬
  to integer(0.333 * stageWidth)
```

See also `float` and `string` functions

---

integerP function

Syntax `integerP(`*expression*`)`

Description This function indicates whether the expression specified by *expression* is an integer:

- When *expression* can be evaluated to an integer, `integerP` is TRUE (1).
- When *expression* cannot be evaluated to an integer, `integerP` is FALSE (0).

The “P” in `integerP` stands for “predicate.”

Example This statement checks whether 3 can be evaluated to an integer. Because it is an integer, the message window displays the number 1, which is the numeric equivalent of TRUE:

```
put integerP(3)
-- 1
```

This statement checks whether 3 can be evaluated to an integer. Because 3 surrounded by quotes cannot be evaluated to an integer, the message window displays the number 0, which is the numeric equivalent of FALSE:

```
put integerP("3")
-- 0
```

Example This statement checks whether the numerical value of the string in text cast member Entry is an integer, and displays an alert if it isn't.

```
if integerP(value(field "Entry")) = FALSE then ¬
  alert "Please enter an integer."
```

See also `floatP`, `objectP`, `stringP`, and `symbolP` functions

intersect — function

Syntax `intersect(`*rectangle1*`,` *rectangle2*`)`

Description This function determines the rectangle formed where *rectangle1* and *rectangle2* intersect.

Example This statement assigns the variable newRectangle the rectangle formed where rectangle toolKit intersects rectangle Ramp:

```
set newRectangle = intersect(vToolKit, vRamp)
```

See also `map`, `offset`, and `rect` functions

intersect rect

See the `intersect` function.

intersects

See the `sprite…intersects` comparison operator.

into

This code fragment occurs in a number of Lingo constructs.

item...of — chunk expression keyword

Syntax `item` *whichItem* `of` *chunkExpression*

Syntax `item` *firstItem* `to` *lastItem* `of` *chunkExpression*

Description This chunk expression keyword specifies an item or a range of items in a chunk expression. An item in this case is any sequence of characters delimited by commas.

The terms *whichItem*, *firstItem*, and *lastItem* must be integers or integer expressions that refer to the position of items in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of text. Sources of text include text cast members and variables that hold strings.

When the number that specifies the last item is greater than the item's position in the chunk expression, the actual last item is specified instead.

Example This statement determines the third item in the chunk expression that consists of names of colors and displays the result in the message window:

```
put item 3 of "red, yellow, blue green, orange"
-- "blue green"
```

The result is the entire chunk blue green because this is the entire chunk within the commas.

This statement attempts to determine the third through tenth items in the chunk expression. Because there are only four items in the chunk expression, the fourth item is used instead of the tenth item. The result appears in the message window:

```
put item 3 to 10 of "red, yellow, blue green, orange"
-- " blue green, orange"
put item 10 of "red, yellow, blue green, orange"
-- ""
```

Example This statement inserts the item Desk as the fourth item in the second line of the text cast member All Bids:

```
put "Desk" into item 4 of line 2 ¬
  of field "All Bids"
```

See also `char…of`, `line…of`, and `word…of` chunk expression keywords; `number of items in` chunk function

the itemDelimiter — property

Syntax `the itemDelimiter`

Description This property indicates the special character used to separate items.

You can use the `itemDelimiter` function to parse filenames by setting `itemDelimiter` to “:”. Be sure to restore it to “,” for normal operation.

The `itemDelimiter` function can be tested and set.

Example This handler determines the last component in a Macintosh pathname. The handler first records what the current delimiter is, and then changes the delimiter to a colon (:). When a colon is the delimiter, Lingo can use the `last item of` to determine the last item in the chunk that makes up a Macintosh pathname. Before exiting, the delimiter is reset to its original value.

```
on getLastComponent pathName
  set save = the itemDelimiter
  set the itemDelimiter = ":"
  set f = the  last item of pathName
  set the itemDelimiter = save
  return f
end
```

items

See the `number of items in` chunk function.

the key — function

Syntax `the key`

Description This function indicates the last key that was pressed.

This function can be used for testing keys within event script and for navigation/keyboard shortcuts.

You can use `the key` to write handlers that perform certain actions when the user presses specific keys. This is a way to provide keyboard shortcuts and other forms of interactivity for the user. When used in a primary event handler, the actions you specify are the first to be executed.

Example These statements have the movie pause when the user presses Return. By setting the `keyDownScript` property to checkKey, the `on Start` movie handler makes the checkKey handler the first event handler executed when a key is pressed. The checkKey handler checks whether the Return key is pressed and pauses the movie if it is:

```
on startMovie
  set the keyDownScript to "checkKey"
end startMovie
on checkKey
  if the key = RETURN then pause
end
```

This `keyDown` handler checks whether the last key pressed is the Enter key, and then calls the handler addNumbers if it is:

```
on keyDown
  if the key = ENTER then addThem
end keyDown
```

See also `commandDown`, `controlDown`, `keyCode`, and `optionDown` functions

the keyCode — function

Syntax `the keyCode`

Description This function gives the numerical code for the last key pressed. (This keyboard code is the key's numerical value, not the ASCII value.)

You can use the `keyCode` function to detect when the user has pressed the arrow or function keys, which cannot be specified by the `key` function. The value of `keyCode` varies on international keyboards.

The `keyCode` function can be tested but not set.

Example This handler uses the message window to display the appropriate key code each time a key is pressed:

```
on enterFrame
  set the KeyDownScript = "put the keyCode"
end
```

This statement checks whether the Up arrow (whose key code is 126) is pressed, and goes to the previous marker if it is:

```
if the keyCode = 126 then go to marker(-1)
```

See also `commandDown`, `controlDown`, `key`, and `the optionDown` functions

keyDown — event handler

See the `on keyDown` event handler, `the keyDownScript` property, and `when keyDown then` command.

the keyDownScript — property

Syntax `the keyDownScript`

Description This property specifies the Lingo that is executed when a key is pressed. The Lingo can be a simple statement or a calling script for a handler.

When a key is pressed and `the keyDownScript` is defined, Lingo executes the instructions specified for `the keyDownScript` first. If you do not want the `keyDown` message to pass on to other objects in the movie, include the `dontPassEvent` command in the `keyDownScript`.

Setting the `keyDownScript` property does the same as using the `when keyDown then` command that appeared in earlier versions of Director.

When the instructions you specify for the `keyDownScript` property are no longer appropriate, turn them off by using the statement `set the keyDownScript to EMPTY`.

Example This statement sets `the keyDownScript` to `if the key = RETURN then continue`. When this is in effect and the movie is paused, the movie always continues whenever the user presses the Return key.

```
set the keyDownScript ¬
  to "if the key = RETURN then continue"
```

See also `the keyUpScript`, `the mouseDownScript`, and `the mouseUpScript` properties

keyUp — function

See the `on keyUp` event handler.

the keyUpScript property

Syntax `the keyUpScript`

Description This property specifies the Lingo that is executed when a key is released. The Lingo can be a simple statement or a calling script for a handler.

When a key is released and `the keyUpScript` is defined, Lingo executes the instructions specified for `the keyUpScript` first. If you do not want the `keyUp` message to pass on to other objects in the movie, include the `dontPassEvent` command in the `keyUpScript`.

When the instructions you've specified for `the keyUpScript` property are no longer appropriate, turn them off by using the statement `set the keyUpScript to empty`.

Example This statement sets `the keyUpScript` to `if the key = RETURN then continue`. When this is in effect and the movie is paused, the movie always continues whenever the user presses the Return key.

```
set the keyUpScript ¬
  to "if the key = RETURN then continue"
```

label function

Syntax `label(`*expression*`)`

Description This function indicates the frame associated with the marker label specified by *expression*. The term *expression* should be a label in the current movie; if it is not, this function returns 0.

Example This statement sends the playback head to the tenth frame after the frame labeled Start:

```
go to label("Start") + 10
```

This statement assigns the frame number of the fourth item in the label list to the variable whichFrame:

```
put label(line 4 of the labelList) into whichFrame
```

See also `go` and `play` commands; `labelList` and `marker` functions

the labelList function

Syntax `the labelList`

Description This function gives a listing of the frame labels in the current movie, one label per line.

Example This statement makes a listing of frame labels the content of the text cast member Key Frames:

```
put the labelList into field "Key Frames"
```

See also `label` and `marker` functions

the last — function

Syntax `the last` *chunk* `in (`*chunkExpression*`)`

Description This function identifies the last chunk specified by *chunk* of the chunk expression specified by *chunkExpression*.

Chunk expressions refer to any character, word, item, or line in any container of text. Containers include the contents of text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges within containers.

Example This statement identifies the last word of the string “Macromedia, the multimedia company” and displays the result in the message window:

```
put the last word of "Macromedia,¬
  the multimedia company"
```

The result is the word “company”.

This statement identifies the last character of the string “Macromedia, the multimedia company” and displays the result in the message window:

```
put the last char of "Macromedia,¬
  the multimedia company"
```

The result is the letter “y”.

See also `char...of` and `word...of` chunk expression keywords

the lastClick — function

Syntax `the lastClick`

Description This function gives the time in ticks (60ths of a second) since the mouse button was last pressed.

The `lastClick` can be tested, but not set.

Example This statement checks whether it has been ten seconds since the last mouse click, and sends the playback head to the marker No Click if it has:

```
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 gives the time in ticks (60ths of a second) since the last mouse click, mouse roll, or key press occurred.

Example This statement checks whether it has been ten seconds since the last mouse click, mouse roll, or key press, and sends the playback head to the marker Help if it has:

```
if the lastEvent > 10 * 60 then go to "Help"
```

See also `lastClick`, `lastKey`, and `lastRoll` functions; `startTimer` command

the lastFrame — property

Syntax `the lastFrame`

Description This property is the number of the last frame in the movie.

The `lastFrame` property can be tested but not set.

Example This statement displays the number of the last frame of the movie in the message window:

```
put the lastFrame
```

the lastKey — function

Syntax

```
the lastKey
```

Description This function gives the time in ticks (60ths of a second) since the last key was pressed.

Example This statement checks whether it has been 10 seconds since the last key was pressed, and sends the playback head to the marker “No Key” if it has:

```
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 gives the time in ticks (60ths of a second) since the mouse was last moved.

Example This statement checks whether it has been 45 seconds since the mouse was last moved, and sends the playback head to the marker “No Roll” if it has:

```
if the lastRoll > 45 * 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 is the left horizontal coordinate of the bounding rectangle of the sprite specified by *whichSprite*.

Sprite coordinates are measured in numbers of pixels, starting with (0,0) at the upper left corner of the stage.

The `left of sprite` property can be tested, but not set. Use the `spriteBox` command to set the left horizontal coordinate of a sprite.

Example

The following statement determines whether the sprite's left edge is to the left of the stage's left edge. If the sprite's left edge is to the stage's left edge, the script runs the handler `offLeftEdge`:

```
if the left of sprite 3 < 0 then offLeftEdge
```

This statement measures the left horizontal coordinate of the sprite numbered (i + 1) and assigns the value to the variable named `vLowest`:

```
put the left of sprite (i + 1) into vLowest
```

See also

`bottom`, `height`, `locH`, `locV`, `right`, `top`, and `width` sprite properties; `spriteBox` command

length — function

Syntax

`length(`*string*`)`

Description

This function gives the number of characters in the string specified by *string*. Spaces and control characters like Tab and Return count as characters.

The length function can be tested, but not set.

Example

This statement displays the number of characters in the string "Macro"&"media":

```
put length("Macro" & "media")
-- 10
```

Example This statement checks whether the content of the text cast member File Name has more than 31 characters and displays an alert if it does:

```
if length(field "File Name") > 31 then ¬
  alert "That file name is too long."
```

See also `chars` and `offset` functions

line...of — chunk expression keyword

Syntax `line` *whichLine* `of` *chunkExpression*

`line` *firstLine* `to` *lastLine* `of` *chunkExpression*

Description This chunk expression keyword specifies 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 be integers that specify a line in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of text. Sources of text include text cast members and variables that hold strings.

Example This statement assigns the first four lines of the variable `vAction` to the text cast member To Do:

```
set the text of cast "To Do" =  lines 1 to 4 ¬
  of vAction
```

This statement inserts the word “and” after the second word of the third line of the text assigned to the variable vNotes:

```
put "and" after word 2 of line 3 of vNotes
```

See also `char...of`, `item...of`, and `word...of` chunk expression keywords; `number of words in` chunk function

lines — chunk keyword

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 *whichSprite*. The `lineSize of sprite` property applies only to shape sprites. For non-rectangular shapes the border is the edge of the shape, not its bounding rectangle.

The `lineSize of sprite` property can be tested and set. For a sprite property to be set using Lingo, the sprite must be a puppet.

Example This statement displays the thickness of the border of sprite 4:

```
put the lineSize of sprite 4 into thickness
```

This statement sets the thickness of the border of sprite 4 to 3 pixels:

```
set the lineSize of sprite 4 to 3
```

list — function

Syntax `list(`*value1*`,` *value2*`,` *value3...*`)`

Description This function defines a linear list made up of the values specified by *value1*, *value2*, *value3*.... This is an alternative way to create a list.

Example This statement sets the variable named designers equal to a linear list that contains the names Gee, Kayne, and Ohashi:

```
set designers = list("Gee", "Kayne", "Ohashi")
```

The result is the list [“Gee”, “Kayne”, “Ohashi”].

listP — function

Syntax `listP(`*item*`)`

Description This function indicates whether the item specified by *item* is a list.

- When listP is TRUE (1), the item specified by *item* is a list.
- When listP is FALSE (0), the item specified by *item* is not a list.

Example This statement checks whether the list named designers is a list and displays the result in the message window:

```
put listP(designers)
```

The result is 1, which is the numerical equivalent of TRUE.

See also `ilk` function

the loaded of cast — cast property

Syntax `the loaded of cast` *whichCastmember*

Description This cast property specifies whether the cast member specified by *whichCastmember* is loaded into memory.

- When `the loaded of cast` is TRUE, the cast member is loaded into memory.
- When `the loaded of cast` is FALSE, the cast member is not loaded into memory.

The `loaded of cast` property can be tested but not set.

Example This statement checks whether cast member Demo Movie is loaded in memory, and goes to an alternate movie if it isn't:

```
if the loaded of cast "Demo Movie" = FALSE then ¬
go to "Waiting"
```

See also `the size of cast` cast property; `preLoad` and `unLoad` commands; `ramNeeded` function

the locH of sprite — sprite property

Syntax `the locH of sprite` *whichSprite*

Description This sprite property is the horizontal position of the specified sprite's registration point. Sprite coordinates are relative to the upper left corner of the stage. See *Using Director* for information about registration points.

The `locH of sprite` property can be tested and set. For a sprite property to be set using Lingo, the sprite must be a puppet.

If you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. If you are changing several sprite properties—or several sprites—you need only one `updateStage` command at the end of all the changes.

Example This statement checks whether the horizontal position of sprite 9's registration point is to the right of the right edge of the stage, and moves the sprite's right edge to the edge of the stage if it is:

```
if the locH of sprite 9 > the stageRight then ¬
  set the locH of sprite 9 to the stageRight
```

This statement puts sprite 15 at the same horizontal location as the mouse click:

```
set the locH of sprite (15) to the mouseH
```

See also `bottom`, `height`, `left`, `locV`, `right`, `top`, and `width` sprite properties; `spriteBox` and `updateStage` commands

the locV of sprite — sprite property

Syntax `the locV of sprite` *whichSprite*

Description This sprite property is the vertical position of the specified sprite's registration point. Sprite coordinates are relative to the upper left corner of the stage. See *Using Director* for information about registration points.

The `locV of sprite` property can be tested and set. For a sprite property to be set using Lingo, the sprite must be a puppet.

If you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. If you are changing several sprite properties—or several sprites—you need only one `updateStage` command at the end of all the changes.

Example This statement checks whether the vertical position of sprite 9's registration point is to the below the bottom of the stage, and moves the sprite's bottom edge to the bottom of the stage if it is:

```
if the locV of sprite 9 > the stageBottom then set
the locV of sprite 9 to the stageBottom
```

This statement puts sprite 15 at the same vertical location as the mouse click:

```
set the locV of sprite (15) to the mouseV
```

See also `bottom`, `height`, `left`, `locH`, `right`, `top`, and `width` sprite properties; `spriteBox` and `updateStage` commands

log — function

Syntax `log(`*number*`)`

Description This function calculates the natural logarithm of the number specified by *number*, which must be a decimal number.

Example This statement assigns the natural logarithm of 10.5 to the variable `vAnswer`. The result is calculated to two decimal places:

```
set vAnswer = log (10.5)
```

This statement calculates the natural logarithm of the square root of the value `vNumber`, and then assigns the result to the variable `vAnswer`:

```
set vAnswer = log (sqrt (vNumber))
```

long

See the `date` and `time` functions.

loop — keyword

Syntax `loop`

Description This keyword refers to the marker. The `loop` keyword with the `go to` command is equivalent to the statement `go to` marker.

Example This handler loops the movie in the current frame:

```
on exitFrame
  go loop
end exitFrame
```

See also `go loop` keyword

the loop of cast — digital video cast property

Syntax `the loop of cast` *castName*

Description This cast property specifies whether digital video movie cast members are set to loop.

- When loop is set to 1, the digital video movie cast member loops.
- When loop is set to 0, the digital video movie cast member does not loop.

Example This statement sets the digital video movie cast member Demo to loop:

```
set the loop of cast Demo to 1
```

the machineType — function

Syntax `the machineType`

Description This function indicates the kind of computer that is currently being used. These codes indicate the type of Macintosh computer:

1	Macintosh 512Ke
2	Macintosh Plus
3	Macintosh SE
4	Macintosh II
5	Macintosh IIx
6	Macintosh IIcx
7	Macintosh SE/30
8	Macintosh Portable
9	Macintosh IIci
11	Macintosh IIfx
15	Macintosh Classic
16	Macintosh IIsi
17	Macintosh LC
18	Macintosh Quadra 900
19	PowerBook 170
20	Macintosh Quadra 700
21	Classic II
22	PowerBook 100
23	PowerBook 140

24	Macintosh Quadra 950
25	Macintosh LCIII
27	PowerBook Duo 210
28	Macintosh Centris 650
30	PowerBook Duo 230
31	PowerBook 180
32	PowerBook 160
33	Macintosh Quadra 800
35	Macintosh LC II
42	Macintosh IIvi
46	Macintosh II vx
47	Macintosh Color Classic
48	PowerBook 165c
50	Macintosh Centris 610
52	PowerBook 145
76	Macintosh Quadra 840av
256	IBM PC-type machine

Note *These codes are for general classification purposes only. It is unwise to use them to make assumptions about the performance or screen size of the computer your movie is running on.*

Example This statement checks whether the computer is a Macintosh Classic and plays the movie Classic Movie if it is:

```
if the machineType = 15 then play "Classic Movie"
```

See also `colorDepth` property; `colorQD` function

map function

Syntax `map(`*targetRect, sourceRect, destination Rect*`)`

Description This function is used to position and size a rectangle, based on the relationship of a second rectangle to a third.

Example This handler modifies te rectangle of sprite n so that it has the same relationship to the dimensions of the stage that sprite 2 has:

```
on scaleMySprite n
  set the stretch of sprite to TRUE
  set the rect of sprite n = ¬
    map(the rect of sprite n, ¬
    the rect of sprite 2, ¬
    the rect of the stage)
  updateStage
end scaleMySprite
```

map point

See the `map` function.

map rect

See the `map` function.

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

Example This statement sends the playback head to the beginning of the current frame:

```
go to marker(0)
```

This statement sets the variable `nextMarker` equal to the next marker in the score:

```
put marker(1) into nextMarker
```

See also `go` command; `frame`, `label`, and `labelList` 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 that 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 for use with XObjects that need to be called at every subframe, such as frame-by-frame video recorders. Scripts should generally use an `on exitFrame` handler if they need to be called at every frame.

See also

`factory` and `method` keywords; `perFrameHook` property; `on exitFrame` handler

max — function

Syntax

`max(`*list*`)`

Syntax

`max (`*value1, value2, value3, …*`)`

Description

This function returns the highest value in the specified list, or the highest of a given series of values.

Example This handler assigns the variable `vWinner` the maximum value in the list `vBids`, which consists of [#Castle:600, #Schmitz:750, #Wang:230]. The result is then inserted in the content of the text cast member Congratulations:

```
on findWinner vBids
  set vWinner = max(vBids)
  set the text of "Congratulations" = ¬
  "You have won, with a bid of $" & vWinner &"!"
end
```

maxInteger function

Syntax `the maxInteger`

Description This function returns the largest whole number that is supported by the system. On most personal computers, this is 2147483647 (2 to the 31st power, minus 1).

This can be useful for initializing boundary variables before a loop or for limit testing.

Example This example generates a table in the message window, of the maximum decimal value that can be represented by a certain number of binary digits:

```
on showMaxValues
  put 31 into b
  put the maxInteger into v
  repeat while v > 0
    put b && "-" && v
    put b-1 into b
    put v/2 into v
  end repeat
end showMaxValues
```

Note *The minimum integer value can be found using the formula* `(the maxInteger*(-1))-1`. *Higher and lower values can be dealt with using floating-point numbers.*

mci — command

Syntax `mci "string"`

Description The multimedia extensions for Microsoft Windows respond to commands sent to the media control interface, or mci. If you plan to use Director Player for Windows to play your movie under Microsoft Windows, you can use the `mci` command to pass the strings specified by *strings* to the Windows media control interface.

Strings passed by the `mci` command play only under Windows; they are not executed on the Macintosh. Because the Macintosh does not support the mci interface, the `mci` command gives you a way to include commands intended for the Windows environment within a movie that you create and can play on the Macintosh.

Example This statement makes the command `play cdaudio from 200 to 600 track 7` play only when the movie plays back under Windows:

```
mci "play cdaudio from 200 to 600 track 7"
```

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 that document the functionality or syntax of these methods.

You only use this method for authoring. Do not include it in scripts within a movie.

Before using `mDescribe` to display an XObject's method, first open the appropriate library using the `openXlib` command. To display information about the XObject, enter the `showXlib` command followed by XobjectName(`mDescribe`) in the message window. A display of all open Xlibrary resource files and all XObjects contained in those Xlibraries.

Example This statement displays methods and comments assigned to the fileIO XObject:

```
fileIO(mDescribe)
```

See also `mMessageList` predefined method; `showXLib` command

mDispose — predefined method

Syntax *object*`(mDispose)`

Description This predefined method supports factories in earlier versions of Lingo. In Director 4, it is recommended that you use lists and parent scripts. They are a simpler way of achieving the same result.

This predefined method is used to destroy the object specified by *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.

You do not need to explicitly dispose of child objects created from parent scripts. Lingo disposes of these objects when they are no longer referenced by a variable within the movie.

It is best that you check for previous instances of an object with the same name, and dispose of it before creating new instances of an object using `mNew`. The initialize handler in the following example illustrates this. In this way, if the movie is aborted before the normal `mDispose`, you won't fill up memory by repeatedly creating new objects. 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`.

Example This handler determines whether the item assigned to the variable myObject is an XObject and disposes of it if it is:

```
on cleanUp
  global myObject
    if objectP(myObject) then myObject(mDispose)
end cleanUp
```

See also `mNew` predefined method

me keyword

Syntax `me`

Description This keyword is used within parent scripts as a shorthand means of referring to the script itself.

In earlier versions of Director, the `me` keyword supported factories. In Director 4, it is recommended that you use the `birth` function or lists. They are a simpler way of achieving the same result.

Example This statement sets the object `myBird1` to the script named Bird. The `me` keyword accepts the parameter `script "Bird"` and is used to return that parameter:

```
set myBird1 to birth (script "Bird")
```

This is the birth handler of the Bird script:

```
on birth me
return me
end
```

See also `birth` function; `ancestor` property

the 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 (K) is 1024 bytes. A megabyte (MB) is 1024K.

```
if the memorySize < 500 * 1024 then alert ¬
  "There is not enough memory to run this movie."
```

See also `freeBlock`, `freeBytes`, and `ramNeeded` functions; `the size of cast` cast property

menu

See `name of menu` property; `name`, `number`, `checkMark`, `enabled`, and `scriptof menuItem` 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 specify the actual content of custom menus, in conjunction with the `installMenu` command. Menu definitions are typed in the text cast members. You refer to a particular menu definition by its cast name or 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 create hierarchical menus by using XCMDs or simulate them by writing scripts that display a graphic cast member that mimics a submenu.

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 and enables Macintosh menu bar items 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 checkmark (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 This set of statements specifies the content of a custom File menu:

```
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; `the checkmark of menuItem`, and `the enabled of menuItem` properties

menuItem

See `the checkMark`, `enabled`, `name`, and `script of menuItem` menu item properties.

menuItems

See `the number of menuItems` menu property.

menus

See `the number of menus` menu property.

method keyword

Syntax `method` *methodName* `{`*argument1*`}{,` *argument2*`}` ...

Description This keyword supports factories in earlier versions of Lingo. In Director 4, it is recommended that you use lists and parent scripts. They are a simpler way of achieving the same result.

This keyword is used to define a method. A method is a special kind of handler that exists inside a factory script or XObject and that has its own special syntax. It uses Lingo to create expressions that are commands or functions. A method is a script, or series of scripts, that handle different messages (or processes) for objects created by a factory, or XObject.

There are two kinds of objects: internal (created by factories) and external (created by XObjects). Factories and XObjects use methods. The difference is that you define a factory's methods in the movie script or a cast member script, but 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 the value you substitute for *messageName* with a lowercase m.

See also `exit`, `factory`, `instance`, and `return` keywords; `mNew` predefined method

mGet — predefined method

Syntax *object*(`mGet`, *whichElement*)

Description This method was used for managing arrays in earlier versions of Director. In Director 4, it is recommended that you use lists and parent scripts. They are a simpler way of achieving the same result.

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 that has not been assigned a value with the `mPut` method, the element has the numerical value 0.

Different types of data can be stored in various elements of the same array. You can use the functions `floatP`, `integerP`, `objectP`, `stringP`, and `symbolP` to determine the data type of a particular element.

Example These first three statements use `mPut` to put data into an internal array. Using 3, 7, and 12 assigns these values to the third, seventh, and twelfth elements of the array:

```
put FactoryName (mNew) into myObject
myObject(mPut, 3, 2 + 2)
myObject(mPut, 7, sqrt(2.0))
myObject(mPut, 12, "hello" && "there")
```

This statement displays the value associated with the third element of the array:

```
put myObject(mGet, 3)
```

The result is 4, which is the equivalent of 2 + 2.

Example This statement displays the value associated with the seventh element of the array:

```
put myObject(mGet, 7)
```

The result is 1.4142, which is the equivalent of the square root of 2.

This statement displays the value associated with the twelfth element of the array:

```
put myObject(mGet, 12)
```

The result is “hello there”, which is the value that was assigned in the first example.

See also `mPut` predefined method

min — function

Syntax `min(`*list*`)`

Syntax `min(`*a1, a2, a3…*`)`

Description This function specifies the minimum value in the list specified by *list*.

Example This handler assigns the variable `vLowest` the minimum value in the list `vBids`, which consists of [#Castle:600, #Shields:750, #Wang:230]. The result is then inserted in the content of the text cast member Sorry:

```
on findLowest vBids
set vLowest = min(vBids)
set the text of "Sorry" = ¬
  "We're sorry, your bid of $" & vLowest && "is not a
  winner!"
end
```

See also `max` function

mInstanceRespondsTo — predefined method

Syntax *XObject*(`mInstanceRespondsTo,` *message*)

Description This predefined method can only be used with XObjects. It 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.

Example This statement checks whether the SerialPort XObject responds to the message string `mWrite`.

```
put SerialPort(mInstanceRespondsTo, "mWrite")
```

The result is 2; one for the first parameter, plus one.

See also `mRespondsTo` predefined method

mMessageList — predefined method

Syntax *XObject*(`mMessageList`)

Description This predefined method can only be used with XObjects. It returns a string that describes the XObject and its methods. The string is the same string that the `mDescribe` method displays in the message window; however, it may be put into fields or variables.

Example This statement displays methods and comments assigned to the fileIO XObject:

```
put fileIO(mMessageList)
```

See also `mDescribe`, `mInstanceRespondsTo`, `mRespondsTo` predefined methods

mName — predefined method

Syntax *XObject*`(mName)`

Syntax *XObjectInstance* `(mName)`

Description This predefined method (which can be used only with XObjects and their instances) gives a string that contains the name of the XObject that created the instance.

Example These statements create an instance of the serial port XObject and places it in the variable `modemPort`. It then displays the name of the XObject instance:

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

The result is `SerialPort`, which is the name of the XObject.

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 you use the `mNew` method to assign instance variables used throughout the methods of a factory object.

In order to clear the object you create using `mNew` from RAM at the end of the movie, it is a good idea to use the predefined `mDispose` method for both factory objects and external XObjects.

Before creating new instances of an object using `mNew`, it is also a good idea to check for previous instances of an object that has 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 the movie before the end, and play it again from the beginning.

Example This statement creates a new instance of `myArrayFactory`. The new instance is named `myArray`:

```
put myArrayFactory(mNew) into myArray
```

This statement creates a new instance of `birdFac`. The new instance is named `bird` and has initial instance variables `wingCastNum` and `legCastNum`:

```
put birdFac(mNew, wingCastNum, legCastNum) into bird
```

This statement creates a new instance of the XObject `PioneerLaserDisc` `myArrayFactory`. The new instance is named `vDisc`:

```
put PioneerLaserDisc(mNew, 1, 9600, 0) into vDisc
```

This handler checks for existing instances of factories and XObjects and disposes of any it finds. It then creates a new instance of the `myArrayFactory`:

```
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
```

See also `factory`; `method` and `instance` keywords; and `mDescribe` and `mDispose` predefined methods

mod — arithmetic operator

Syntax *integerExpression1* `mod` *integerExpression2*

Description This arithmetic 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.

Example This statement divides 7 by 4 and then displays the remainder in the message window:

```
put 7 mod 4
```

The result is 3.

This handler sets the ink effect of all odd-numbered sprites to copy, which is the ink effect specified by the number 0. First, the handler checks whether the sprite that has the number in the variable `mySprite` is an odd-numbered sprite by dividing the sprite number by 2 and then checking whether the remainder is 1. When the remainder is 1, which is the result for an odd-numbered number, the ink effect is set to copy:

```
on setInk
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
end setInk
```

the modal of window — window property

Syntax `the modal of window "`*window*`"`

Description This window property specifies whether movies can respond to events that occur outside the window specified by *window*.

- When the `modal of window` property is TRUE, movies cannot respond to events outside the window.
- When the `modal of window` property is FALSE, movies can respond to events outside the window.

Setting the modal of window to TRUE lets you define that a movie that plays in a window is the only movie that the user can interact with.

Example This statement lets movies respond to events outside of the window Tool Panel:

```
set the modal of window "Tool Panel" to FALSE
```

the modified of cast — cast property

Syntax `the modified of cast` *castMember*

Description This cast property indicates whether the cast member specified by *castMember* has been modified since it was read in from the movie file.

- When `the modified of cast` is TRUE (1), the cast member has been modified since it was read from the movie file.
- When `the modified of cast` is FALSE (0), the cast member has not been modified since it was read from the movie file.

Example This statement tests whether the cast member Introduction Text has been modified since it was read from the movie file:

```
put the modified of cast "Introduction Text"
```

The result is 0, which is the numerical equivalent of FALSE.

the mouseCast — function

Syntax `the mouseCast`

Description This integer function gives the cast number of the sprite that is under the cursor when the function is called. When the cursor is not over a cast member, it gives the result -1.

This is useful for having the movie perform specific actions when the cursor rolls over a sprite and the sprite uses a certain cast member.

Example This statement checks whether the cast member `Off Limits` is the cast member assigned to the sprite under the cursor and displays an alert if it is. This is one example of how you can specify an action depending on which cast member is assigned to the sprite:

```
if the mouseCast = the number of cast "Off Limits"¬
  then alert "Stay away from there!"
```

This statement assigns the number of the sprite under the cursor to the variable `lastCast`:

```
put the mouseCast into lastCast
```

See also `castNum of sprite` sprite property; `mouseChar`, `mouseItem`, `mouseLine`, `mouseWord`, and `rollOver` functions; `number of cast` property

the mouseChar — function

Syntax `the mouseChar`

Description This integer function, used for text sprites, gives the number of the character that is under the cursor when the function is called. The count is from the beginning of the field. If the mouse is not over a field or is in the gutter of a field, the result is -1.

Example This statement determines whether the cursor is not over a text sprite and changes the content of the text cast member Instructions to "Please point to a character." when it is:

```
if the mouseChar = -1 then ¬
  put "Please point to a character." ¬
  into field "Instructions"
```

This statement assigns the character under the cursor in the specified text field to the variable `currentChar`:

```
put char (the mouseChar) of field (the mouseCast) ¬
  into currentChar
```

See also `mouseItem`, `mouseLine`, and `mouseWord` functions; `char...of` chunk expression keyword; `number of chars in` chunk function

mouseDown

See the `on mouseDown` event handler, `when mouseDown then` command.

the mouseDown function

Syntax `the mouseDown`

Description This function indicates whether the mouse button is currently being pressed.

- When `the mouseDown` is TRUE, the button is being pressed.
- When `the mouseDown` is FALSE, the button is not being pressed.

Example This handler has the movie beep until the user clicks the mouse button:

```
on enterFrame
  repeat while the mouseDown = FALSE
    beep
  end repeat
```

This statement has Lingo exit the repeat loop or handler it is in when the user clicks the mouse button:

```
if the mouseDown then exit
```

See also `mouseH`, `mouseUp`, and `mouseV` functions; `on mouseDown` and `on mouseUp` event handlers

the mouseDownScript — property

Syntax `the mouseDownScript`

Description This property specifies the Lingo that is executed when the mouse button is pressed. The Lingo can be a simple statement or a calling script for a handler.

When the mouse button is pressed and `the mouseDownScript` is defined, Lingo executes the instructions specified for `the mouseDownScript` first. If you do not want the `mouseDown` message to pass on to other objects in the movie, use the `dontPassEvent` command in the `mouseDownScript`.

Setting the `mouseDownScript` property does the same as using the `when mouseDown then` command that appeared in earlier versions of Director.

When the instructions you've specified for the `mouseDownScript` property are no longer appropriate, turn them off by using the statement `set the mouseDownScript to empty`.

The `mouseDownScript` property can be tested and set, and the default value is EMPTY, which means that `the mouseDownScript` has no Lingo at all assigned to it.

Example This statement sets `the mouseDownScript` to `if the mouseDown then go to next`. When this is in effect and the user clicks the mouse button, the playback head always jumps to the next marker in the movie:

```
set the mouseDownScript ¬
  to "if the mouseDown then go to next"
```

This statement sets `the mouseDownScript` so that if the user clicks anywhere on the stage, the computer beeps. When this is in effect and the user clicks anywhere on the stage, the computer beeps:

```
set the mouseDownScript ¬
  to "if the clickOn = 0 then beep"
```

See also `dontPassEvent` command; `mouseUpScript` property; and `on mouseDown` and `on mouseUp` event handlers

the mouseH function

Syntax `the mouseH`

Description This function indicates the horizontal position of the mouse cursor. The value of `mouseH` is the number of pixels the cursor is from the left edge of the stage.

The `mouseH` function is useful for moving sprites to the horizontal position of the mouse cursor and checking whether the cursor is within a region of the stage. Using `mouseH` and `mouseV` functions together, you can determine the cursor's exact location.

The `mouseH` function can be tested but not set.

Example This handler moves sprite 10 to the mouse cursor location and updates the stage when the user clicks the mouse button:

```
on mouseDown
  set the locH of sprite 1 to the mouseH
  set the locV of sprite 1 to the mouseV
  updateStage
end
```

Example This statement tests whether the cursor is more than ten pixels to the right or left of a starting point and sets the variable `Far` to TRUE if it is:

```
if abs(the mouseH - startH) > 10 then ¬
  put TRUE into draggedEnough
```

See also `locH` and `locV` sprite properties; `mouseV` function

the mouseItem function

Syntax `the mouseItem`

Description This integer function gives the number of the item that is under the pointer when the function is called and the cursor is over a text sprite. (An item is any sequence of characters delimited by commas.) Counting starts at the beginning of the field. If the mouse is not over a field, the result is –1.

Example This statement determines whether the cursor is over a text sprite and changes the content of the text cast member `Instructions` to "`Please point to an item.`" when it is not:

```
if the mouseItem = -1 then ¬
  put "Please point to an item." ¬
  into field "Instructions"
```

This statement assigns the item under the cursor in the specifed text field to the variable `currentItem`:

```
put item (the mouseItem ) of field (the mouseCast) ¬
  into currentItem
```

See also `item`…`of` chunk expression keyword; `mouseChar`, `mouseLine`, and `mouseWord` functions; `number of items in` chunk function

the mouseLine — function

Syntax `the mouseLine`

Description This integer function gives the number of the line under the pointer when the function is called and the cursor is over a text sprite. Counting starts at the beginning of the field. When the mouse is not over a text sprite, the result is -1.

Example This statement determines whether the cursor is over a text sprite and changes the content of the text cast member Instructions to "Please point to a line." when it is not:

```
if the mouseLine = -1 then ¬
  put "Please point to a line." ¬
  into field "Instructions"
```

This statement assigns the number of the item under the cursor in the specifed text field to the variable currentLine:

```
put line (the mouseLine) of field (the mouseCast) ¬
  into currentLine
```

See also `line…of` chunk expression keyword; `mouseChar`, `mouseItem`, and `mouseWord` functions; `number of lines in` chunk function

mouseUp

See the `on mouseUp` event handler and the `mouseUpScript` property.

the mouseUp — function

Syntax `the mouseUp`

Description This function indicates whether the mouse button is being pressed.

- The `mouseUp` function is TRUE when the mouse button is not being pressed.
- The `mouseUp` function is FALSE when the mouse button is being pressed.

Example This handler has the movie beep until the user clicks the mouse button:

```
on enterFrame
  repeat while the mouseUp = FALSE
    beep
  end repeat
end enterFrame
```

This statement has Lingo exit the repeat loop or handler it is in when the user clicks the mouse button:

```
if the mouseUp then exit
```

See also `mouseDown`, `mouseH`, and `mouseV` functions; `on mouseDown` and `on mouseUp` event handlers

the mouseUpScript property

Syntax

```
the mouseUpScript
```

Description

This property determines the Lingo that is executed when the mouse button is released. The Lingo can be a simple statement or a calling script for a handler.

When a key is released and `the mouseUpScript` is defined, Lingo executes the instructions specified for `the mouseUpScript` first. If you do not want the `mouseUp` message to pass on to other objects in the movie, use the `dontPassEvent` command in the `mouseUpScript`.

When the instructions you've specified for the `mouseUpScript` property are no longer appropriate, turn them off by using the statement `set the mouseUpScript to empty`.

```
when mouseUp then nothing
```

Setting the `mouseDownScript` property does the same as using the `when keyDown then` command that appeared in earlier versions of Director.

The `mouseUpScript` property can be tested and set. The default value is `EMPTY`.

Example

This statement sets `the mouseUpScript` to `continue`. When this is in effect and the movie is paused, the movie always continues whenever the user releases the mouse button:

```
set the mouseUpScript to "continue"
```

This statement has the movie beep when the user releases the mouse button after clicking anywhere on the stage:

```
set the mouseUpScript ¬
  to "if the clickOn = 0 then beep"
```

See also

`dontPassEvent` command; `mouseDownScript` property; `on mouseDown` and `on mouseUp event handlers`

the mouseV — function

Syntax

```
the mouseV
```

Description

This function indicates the vertical position of the mouse cursor. The value of `mouseV` is the number of pixels the cursor is from the top of the stage.

The `mouseV` function is useful for moving sprites to the vertical position of the mouse cursor and checking whether the cursor is within a region of the stage. Using `mouseH` and `mouseV` functions together, you can identify the cursor's exact location.

Example

This handler moves sprite ten to the mouse cursor location and updates the stage when the user clicks the mouse button:

```
on mouseDown
  set the locH of sprite 10 to the mouseH
  set the locV of sprite 10 to the mouseV
  updateStage
end
```

This statement tests whether the cursor is more than ten pixels above or below a starting point and sets the variable `vFar` to TRUE if it is:

```
if abs(the mouseV - startV) > 10 then ¬
  put TRUE into draggedEnough
```

See also

`mouseH` function; `locH` and `locV` sprite properties

the mouseWord — function

Syntax

```
the mouseWord
```

Description

This integer function gives the number of the word under the cursor when the function is called and when the cursor is over a text sprite. Counting starts from the beginning of the field. When the mouse is not over a field, the result is –1.

Example This statement determines whether the cursor is over a text sprite and changes the content of the text cast member Instructions to "`Please point to a word.`" when it is not:

```
if the mouseWord = -1 then ¬
  put "Please point to a word." ¬
  into field "Instructions"
```

This statement assigns the number of the word under the cursor in the specified text field to the variable `currentWord`:

```
put word (the mouseWord) of field (the mouseCast) ¬
  into currentWord
```

See also `mouseChar`, `mouseItem`, and `mouseLine` functions; `number of words in` chunk function; `word...of` chunk expression keyword

move cast command

Syntax `move cast` *whichCastmember*`{, cast` *whichLocation*`}`

Description This command moves the cast member specified by *whichCastmember* to a different location in the cast window.

- Using the `move cast` command without the optional parameter, the cast member moves to the first empty location in the cast window.
- Including the `cast` *whichLocation* parameter in the `move cast` command moves the cast member to the location specified by *whichLocation*.

Example This statement moves cast member Shrine to the first empty location in the cast window:

```
move cast "Shrine"
```

This statement moves cast member Shrine to location 20 in the cast window:

```
move cast "Shrine", cast 20
```

the moveableSprite of sprite — sprite property

Syntax `the moveableSprite of sprite` *whichSprite*

Description This sprite property indicates whether a sprite is moveable.

- When the sprite can be moved by the user, the `moveableSprite of sprite` is TRUE (1).
- When the sprite cannot be moved by the user, the `moveableSprite of sprite` is FALSE (0).

To use Lingo to make a sprite moveable, the sprite must first be a puppet sprite.

You can also make a sprite moveable by using the Moveable option in the score. However, controlling whether a sprite is moveable by using Lingo lets you turn this condition on and off as situations in the movie require. For example, referring to the "Mechanical Simulation" sample movie, you could let the user drag parts from the toolkit but make them unmoveable after they are on the pegboard by turning `moveableSprite of sprite` on and off at the appropriate times.

Setting the `moveableSprite of sprite` property lets you control whether sprites are moveable from other scripts.

The `moveableSprite of sprite` property can be tested and set.

Example This handler first makes the sprite a puppet and then makes it moveable:

```
on spriteMove
  puppetSprite 5, TRUE
  set the moveableSprite of sprite 5 to TRUE
end
```

This statement checks whether a sprite is moveable and displays a message if it isn't:

```
if the moveableSprite of sprite 13 = FALSE ¬
  then set the text of cast "Notice" to ¬
  "You can't drag this item by using the mouse."
```

See also `puppetSprite` command

moveToBack command

Syntax `moveToBack window "`*whichWindow*`"`

Description This command moves the window specified by *whichWindow* behind all other windows.

Example This statement moves the window Demo Window behind all other windows:

```
set myWind=getat(the windowList, 1,1)
moveToBack myWind
```

Note *Note that the first record of the windowList contains the text "Demo Window" so the long version of the moveToBack would read:*

```
moveToBack window “Demo Window”
```

moveToFront command

Syntax `moveToFront window "`*whichWindow*`"`

Description This command moves the window specified by *whichWindow* to the front of all other windows.

Example This statement moves the window Demo Window in front of all other windows:

```
set myWind=getat(the windowList, 1,1)
moveToFront myWind
```

Note *Note that the first record of the windowList contains the text "Demo Window" so the long version of the moveToFront would read:*

```
moveToFront window “Demo Window”
```

movie

See the `go` and `play` commands.

the movie — function

Syntax `the movie`

Description This string function returns the name of the currently open movie.

Example This statement assigns the name of the current movie to the text field Current Movie:

```
put the movie into field "Movie Name"
```

See also `pathName` function

movieFileFreeSize — function

Syntax `the movieFileFreeSize`

Description This function returns the amount of unused space in the current movie in bytes.

Note *Movies saved with the Save And Compact command or the Save As command do not have any unused space. This function will return 0.*

movieFileSize — function

Syntax `the movieFileSize`

Description This function returns the size of the current file in bytes.

the movieName — function

Syntax `the movieName`

Description This function indicates the simple name of the current movie. The `movieName` function is equivalent to the movie function.

Example This statement displays the name of the current movie in the message window:

```
put the movieName
```

See also `movie`, `moviePath`, `pathName` functions

the moviePath — function

Syntax `the moviePath`

Description This function indicates the pathname of the folder that the current movie is located in. The `moviePath` function is equivalent to the `pathName` function.

Example This statement displays the pathname of the current movie's folder:

```
put the moviePath
```

See also `movie`, `movieName`, and `pathName` functions

the movieRate of sprite — digital video sprite property

Syntax `the movieRate of sprite` *channelNumber*

Description This sprite property controls the rate at which a digital video movie in a specific channel plays. The movie rate is a value specifying the playback of the digital video movie. A value of 1 is normal forward play, -1 is reverse, 0 is stop. Higher and lower values are possible, but frames may be dropped, depending on the performance of the computer the movie is playing on.

Example This statement sets the rate for a digital video movie in sprite channel 9 to normal playback speed:

```
set the movieRate of sprite 9 to 1
```

This statement has the digital video movie in sprite channel 9 play in reverse:

```
set the movieRate of sprite 9 to -.1
```

---

the movieTime of sprite — digital video sprite property

Syntax `the movieTime of sprite` *channelNumber*

Description This sprite property determines the current time of a digital video movie playing in the channel specified by *channelNumber*. The value of `the movieTime` is measured in ticks.

`The movieTime of sprite` property can be tested and set.

Example This statement displays the current time of the digital video movie in channel 9 in the message window:

```
put the movieTime of sprite 9
```

This statement sets the current time of the digital video movie in channel 9 to the value in the symbol #Poster:

```
set the movieTime of sprite 9 to #Poster
```

mPerform — predefined method

Syntax *object*(`mPerform`, *message*{, *argument1*}{, *argument2*}...)

Description This predefined method only works with XObjects and factory objects. Factories were supported in earlier versions of Director. In Director 4, it is recommended that you use list and parent scripts instead of factories. They are a simpler method of achieving the same result.

This predefined method 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.

This is accomplished 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.

Optional additional arguments, which can be any data type, constant, or property used in the method to be invoked, follow this required first argument.

Typically, the object name is specified by use of the `me` keyword, since the typical use of `mPerform` is within a factory method that invokes one of several other methods.

A powerful use for `mPerform` is to eliminate a lot of `if`...`then` conditional tests within methods that call other methods.

Example This statement creates an instance named `modemPort` of the `SerialPort` XObject:

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

These statements invoke the `mWriteChar` method with the argument `charNum`:

```
modemPort(mPerform, "mWriteChar", charNum)
modemPort(mWriteChar, charNum)
```

See also `factory`, `me`, and `method` keywords

mPut — predefined method

Syntax *object*(`mPut`, *whichElement*, *expression*)

Description This predefined method, was used for managing arrays in earlier versions of Director. In Director 4, it is recommended that you use lists to manage arrays. Lists are a simpler means of achieving the same result.

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.

Note *Methods were used for managing arrays in earlier versions of Director. Lists are a simpler alternative to methods for managing arrays.*

Example These first three statements use `mPut` to put data into an internal array. Using 3, 7, and 12 assigns these values to the third, seventh, twelfth elements of the array:

```
myObject(mPut, 3, 2 + 2)
myObject(mPut, 7, sqrt(2.0))
myObject(mPut, 12, "hello" && "there")
```

This statement displays the value associated with the third element of the array:

```
put myObject(mGet, 3)
```

The result is 4.

Example This statement displays the value associated with the seventh element of the array:

```
put myObject(mGet, 7)
```

The result is 1.4142, which is the square root of 2.

This statement displays the value associated with the twelfth element of the array:

```
put myObject(mGet, 12)
```

The result is the string “hello there”.

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 when *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 These statements create an instance of the XObject `SerialPort` and checks whether it responds to the message string `mWrite`.

```
put SerialPort(mNew, 0) into modemPort
put modemPort(mRespondsTo, "mWrite")
```

See also `mInstanceRespondsTo` predefined method

multiSound system property

Syntax

```
the multiSound
```

Description This system property is TRUE when the system supports more than one sound channel.

Example This statement plays the sound file Music in sound channel 2 if the computer supports more than one sound channel:

```
if the multiSound sound playFile 2, "Music"
```

the name of cast — cast property

Syntax `the name of cast` *whichCastmember*

Description This cast property determines the name of the specified cast member.

- When *whichCastmember* evaluates to a string, it is used as the cast name.
- When *whichCastmember* evaluates to an integer, it is used as the cast number.

The name is a descriptive string assigned by the user. Setting this property is equivalent to entering a name in the Cast Member Info dialog box.

The name cast property can be tested and set.

Example This statement changes the name of cast member named On to Off:

```
set the name of cast "On" to "Off"
```

This statement sets the name of cast member 15 to `Background Sound`:

```
set the name of cast 15 to "Background Sound"
```

This statement sets the variable `itsName` to the name of the cast member that follows the cast member whose number is equal to the variable `i`:

```
put the name of cast (i + 1) into itsName
```

See also `number of 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.

The `name of menu` property can be tested but cannot be set directly. Use the `installMenu` command to set up a custom menu bar.

Example This statement assigns the name of menu number 1 to the variable `firstMenu`:

```
put the name of menu 1 into firstMenu
```

The following handler returns a list of 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 of menus` menu property; `name of menuItem` menu item property

the name of menuItem — menu property

Syntax `the name of menuItem` *whichItem* `of menu` *whichMenu*

Description This menu item property determines the text that appears in the menu item specified by *whichItem* in the menu specified by *whichMenu*. The *whichItem* expression can be either a menu item name or a menu item number; *whichMenu* can be either a menu name or a menu number.

The `name of menuItem` property can be tested and set.

Example This statement sets the variable `itemName` to the name of the eighth item in the Edit menu:

```
put the name of menuItem 8 of menu "Edit" ¬
  into itemName
```

This statement has a specific filename follow the term Open in the File menu:

```
set the name of menuItem "Open" of menu fileMenu ¬
  to "Open" & fileName
```

See also `name of menu` property; `number of menuItem` property;

next keyword

Syntax `next`

Description This keyword refers to the next marker in the movie. The `next` keyword is equivalent to the phrase `the marker (+ 1)`.

Example This statement sends the playback head to the next marker in the movie:

```
go next
```

See also `go next` keyword

next repeat keyword

Syntax `next repeat`

Description This keyword causes Lingo to go to the next step in a repeat loop. This is different from the `exit repeat` keyword.

This repeat loop displays only odd numbers in the message window:

```
repeat with i = 1 to 10
  if (i mod 2) = 0 then next repeat
  put i
end repeat
```

not — logical operator

Syntax `not` *logicalExpression*

Description This logical operator performs a logical negation on a logical expression.

- When the expression specified by *logicalExpression* is TRUE, the result is FALSE (0).
- When the expression specified by *logicalExpression* is FALSE, the result is TRUE (1).

This is a logical operator with a precedence level of 5.

Example This statement determines whether 1 is not less than 2:

```
put not (1 < 2)
```

Because 1 is less than 2, the result is 0, which indicates that the expression is FALSE.

This statement determines whether 1 is not greater than 2:

```
put not (1 > 2)
```

Because 1 is not greater than 2, the result is 1, which indicates that the expression is TRUE.

This handler sets the `checkMark` of `menuItem` for the item Bold in the Style menu to the opposite of its current setting:

```
on resetMenuItem
  set the checkMark of menuItem "Bold" ¬
    of menu "Style" to not (the checkMark ¬
    of menuItem "Bold" of menu "Style")
end resetMenuItem
```

See also `and` and `or` logical operators

nothing — command

Syntax `nothing`

Description This command does nothing at all. It is useful for making the logic of an `if...then` statement more obvious. Also, a nested `if…then…else` statement that contains no explicit command for the `else` clause may require `else nothing`. Otherwise, Lingo interprets the `else` clause as part of the preceding `if`. (See the second example below.)

Example The nested `if...then...else` statement in this handler uses the `nothing` command to satisfy the statement's else clause:

```
on mouseDown
  if the clickOn = 1 then
    if the moveable of sprite 1 = TRUE ¬
    then set the text of cast "Notice" = ¬
    "Drag the ball"
    else nothing
  else set the text of cast "Notice" = ¬
    "Click again"
  end if
end mouseDown
```

This handler has the movie do nothing as long as the mouse button is being pressed:

```
on mouseDown
  repeat while the stillDown
    nothing
  end repeat
end mouseDown
```

See also `if…then` and `if...then...else` keywords

the number of cast — cast property

Syntax

`the number of cast` *whichCastmember*

Description

This cast property indicates the cast number of the cast member specified by *whichCastmember*.

- When *whichCastmember* is a string, the string is used as the cast member name.
- When *whichCastmember* is an integer, the integer is used as the cast member number.
- When *whichCastmember* is an octal number (octal numbers were used in earlier versions of Director), *whichCastmember* is replaced with the decimal equivalent of the octal number. The first cast member, A11 becomes cast number 1; A12 becomes cast number 2; and so on.

The `number of cast` property can be tested, but not set.

Example

This statement assigns the cast number of the cast member Power Switch to the variable `whichCastmember`:

```
put the number of cast "Power Switch" into ¬
  whichCastmember
```

This statement assigns the cast member Red Balloon to sprite 1:

```
set the castNum of sprite 1 ¬
  to the number of cast "Red Balloon"
```

See also

`castNum of sprite` property; `the number of cast members` property

the number of castMembers — property

Syntax `the number of castMembers`

Description This property indicates the number of the last cast member in the current movie. Some of the cast member slots may be empty, so the actual number of cast members may be fewer than the `number of cast members` value.

The `number of castMembers` property can be tested, but not set.

Example The following handler returns a string containing a list of all the cast member 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 `number of cast` cast property

the number of chars in — chunk function

Syntax `the number of chars in` *chunkExpression*

Description This chunk function returns a count of the characters in a chunk expression.

Chunk expressions refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Spaces and control characters such as Tab and Return count as characters.

Example This statement displays the number of characters in the string "Macromedia, the multimedia company" in the message window:

```
put the number of chars ¬
  in "Macromedia, the multimedia company"
```

The result is 33.

This statement sets the variable `charCounter` to the number of characters in the ith word in the string Names:

```
put the number of chars in word i of "Names" into
  charCounter
```

See also `length` function; `number of items in`, `number of lines in`, and `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 chunk function returns a count of the items in a chunk expression. An item chunk is any sequence of characters delimited by commas.

Chunk expressions refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Example This statement displays the number of items in the string "Macromedia, the multimedia company" in the message window:

```
put the number of items ¬
  in "Macromedia, the multimedia company"
```

The result is 2.

This statement sets the variable `itemCounter` to the number of items in the field Names:

```
put the number of items in field "Names" into
  itemCounter
```

See also `item...in` chunk expression keyword; `number of chars in`, `number of lines in`, and `number of words in` chunk functions

the number of lines in — chunk function

Syntax `the number of lines in` *chunkExpression*

Description This chunk 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 cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Example This statement displays the number of lines in the string "Macromedia, the multimedia company" in the message window:

```
put the number of lines ¬
  in "Macromedia, the multimedia company"
```

The result is 1.

This statement sets the variable `lineCounter` to the number of lines in the field Names:

```
put the number of lines in field "Names" into
  lineCounter
```

See also `line...in` chunk expression keyword; `number of chars in`, `number of items in`, and `number of words in` chunk functions

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 custom menu specified by *whichMenu*. The *whichMenu* parameter can be a menu name or a menu number.

The `number of menuItems` menu property can be tested but not set directly. Use the `installMenu` command to set up a custom menu bar.

Example This statement sets the variable `fileItems` to the number of menu items in the custom File menu:

```
put the number of menuItems of menu "File" ¬
  into fileItems
```

This statement sets the variable `itemCount` to the number of menu items in the custom menu whose menu number is equal to the variable `i`:

```
put the number of menuItems of menu i into itemCount
```

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 tested, but not set. Use the `installMenu` command to set up a custom menu bar.

Example This statement determines whether there are any custom menus installed in the movie and installs the menu `Menubar` if no menus are already installed:

```
if the number of menus = 0 then ¬
  installMenu (the number of cast "Menubar")
```

This statement has the message window display the number of menus that are in the current movie:

```
put the number of menus
```

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 tells how many words are in the chunk expression specified by *chunkExpression*.

Chunk expressions refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Example This statement has the message window display the number of words in the string "Macromedia, the multimedia company":

```
put the number of words ¬
  in "Macromedia, the multimedia company"
```

The result is 4.

This handler reverses the order of words in the string specified by the argument wordList:

```
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 wordList
```

See also `number of chars in`, `number of items in`, and `number of lines in` chunk functions; `word...of` chunk expression keyword

numToChar function

Syntax `numToChar(`*integerExpression*`)`

Description This function gives 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.

Example This statement has the message window display the character whose ASCII number is 65:

```
put numToChar(65)
```

The result is the letter “A.”

See also `charToNum` function

objectP function

Syntax `objectP(`*expression*`)`

Description This function indicates whether the expression specified by *expression* is an object produced by a parent script, factory, or XObject.

- When `objectP` is TRUE, the expression is such an object.
- When `objectP` is FALSE, the expression is not such an object.

The "P" in `objectP` stands for "predicate."

It is good practice to use `objectP` to determine which items are XObjects when you create XObjects by using `mNew` or disposing of XObjects by using `mDispose`.

Example This statement checks whether `modemPort` is an XObject and displays the result in the message window:

```
put objectP(modemPort)
```

This handler checks whether `externalFile` is an XObject and disposes of it if it is:

```
on stopMovie
  if objectP(externalFile) then ¬
    externalFile(mDispose)
end stopMovie
```

See also `floatP`, `integerP`, `stringP`, and `symbolP` functions; `mDispose` and `mNew` predefined methods

of keyword

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 tells the number of the position in *stringExpression2* where the first character of *stringExpression1* first occurs.

- When *stringExpression1* is found in *stringExpression2*, the result is the number that indicates the position of the first occurrence.
- When *stringExpression1* is not found in *stringExpression2*, the result is 0.

Lingo counts spaces as characters in both strings. The string comparison is not sensitive to case or diacritical marks. For example, Lingo considers “a” and “Å” the same character.

Example This statement has the message window display the beginning position of the string “media” within the string “Macromedia”:

```
put offset("media","Macromedia")
```

The result is 6.

This statement has the message window display the beginning position of the string “Micro” within the string “Macromedia”:

```
put offset("Micro", "Macromedia")
```

The result is 0, because “Macromedia” doesn’t contain the string “Micro”.

See also `chars` and `length` functions; `contains` and `starts` comparison operators

offset rect — function

Syntax

`offset (`*rectangle*`, `*horizontalChange*`, `*verticalChange*`)`

Description

This function yields a rectangle that is offset from the rectangle specified by *rectangle*. The horizontal offset is the value specified by *widthChange*; the vertical offset is the value specified by *heightChange*.

- When *heightChange* is greater than zero, the offset is toward the top of the stage; when *heightChange* is less than zero, the offset is toward the bottom of the stage.
- When *widthChange* is greater than zero, the offset is toward the right of the stage; when *heightChange* is less than zero, the offset is toward the left of the stage.

The values for *heightChange* and *widthChange* are in pixels.

Example

This statement sets the variable `newPlace` to a rectangle that is 100 pixels to the right and 150 pixels above the rectangle named `oldPlace`:

```
set newPlace to offset rect(oldPlace, 100, 150)
```

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 score scripts, movie scripts, and scripts of cast members. A handler in the script of a cast member or in a score script can only be called by other handlers in the same script. A handler in a movie script can be called from anywhere.

You can use the same handler in more than one movie by putting the handler's script in the shared cast.

For more information about handlers, see Chapter 3, "Concepts," in *Using Lingo*.

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keyDown`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on enterFrame — event handler

Syntax

```
on enterFrame
    statement(s)
end enterFrame
```

Description This event handler contains statements that are executed each time the playback head enters the frame that the `on enterFrame` handler is attached to. The `on enterFrame` handler is equivalent to the `on stepMovie` handler used in earlier versions of Director.

The `on enterFrame` event handler is a good place for Lingo that you want executed once at every new frame.

Place `on enterFrame` handlers in frame scripts or movie scripts.

- When you want to assign the handler to an individual frame, put the handler in the frame script.
- When you want to assign the handler to every frame unless you explicitly instruct the movie otherwise, put the `on enterFrame` handler in a movie script. The handler then executes every time the playback head enters a frame unless the frame script has its own `on enterFrame` handler. When the frame script has its own `on enterFrame` handler, the `on enterFrame` handler in the frame script overrides the one in the movie script.

Example This handler turns off the puppet condition for sprites 1 through 5 each time the playback head enters the frame:

```
on enterFrame
  repeat with i = 1 to 5
    puppetSprite i, FALSE
  end repeat
end
```

See also `on exitFrame`, `on idle`, `on keyDown`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on exitFrame — event handler

Syntax

```
on exitFrame
```

  statement(s)

```
end
```

Description This event handler contains statements that are activated each time the playback head exits the frame that the `on exitFrame` handler is attached to. The `on exitFrame` handler is a useful place for Lingo that resets conditions that are no longer appropriate after leaving the frame.

Place `on exitFrame` handlers in frame scripts or movie scripts.

- When you want to assign the handler to an individual frame, put the handler in the frame script.
- When you want to assign the handler to every frame unless explicitly instructed otherwise, put the handler in a movie script. The `on exitFrame` handler then executes every time the playback head exits the frame unless the frame script has its own `on exitFrame` handler. When the frame script has its own `on exitFrame` handler, the `on exitFrame` handler in the frame script overrides the one in the movie script.

Example This handler turns off all `puppetSprite` conditions when the playback head exits the frame:

```
on exitFrame
  repeat with i = 48 down to 1
    set the puppet of sprite i = FALSE
  end repeat
end exitFrame
```

This handler sends the playback head to a specified frame if the value in the variable `vTotal` exceeds 1000 when the playback head exits the frame:

```
on exitFrame
  if vTotal > 1000 then go to frame "Finished"
end
```

See also `on enterFrame`, `on idle`, `on keyDown`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on idle — event handler

Syntax

```
on idle
  statement(s)
end idle
```

Description This event handler contains statements that are executed whenever the movie has no other events to handle.

This is a useful location for a Lingo statement that you want to execute as frequently as possible. Some common cases are updating values in global variables and displays that tell current movie conditions.

Because statements in `on idle` handlers run frequently, it is good practice to avoid placing Lingo that takes a long time to process in an `on idle` handler.

Example This handler updates the time being displayed in the movie whenever there are no other events to handle:

```
on idle
  put the short time into field "Time"
end idle
```

See also `on enterFrame`, `on exitFrame`, `on keyDown`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on keyDown — event handler

Syntax

```
on keyDown
  statement(s)
end
```

Description This event handler contains statements that are activated when a key is pressed.

When a key is pressed, Lingo searches these locations in order for an `on keyDown` handler: primary event handler, editable text sprite script, script of a text cast member, frame script, and movie script. (For sprites and cast members, `on keyDown` handlers work only for editable text. A `keyDown` on a different type of cast member, such as a bitmap, has no effect.)

Lingo stops searching when it reaches the first location that has an `on keyDown` handler, unless the handler includes the `pass` command to explicitly pass the `keyDown` message on to the next location.

The `on keyDown` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to have occur when the user presses keys.

Where you place an `on keyDown` handler can affect when it runs.

- When you want the handler to apply to a specific editable text sprite, put the handler in a sprite script.
- When you want the handler to apply to an editable text cast member in general, put the handler in a script of the cast member.

- When you want the handler to apply to an entire frame, put the handler in a frame script.
- When you want the handler to apply throughout the entire movie, put the handler in a movie script.

You can override an `on keyDown` handler by placing an alternate `on keyDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyDown` handler assigned to a cast member by placing an `on keyDown` handler in a sprite script.

Example This handler checks whether the Return key was pressed and sends the playback head to another frame if it was:

```
on keyDown
  if the key = RETURN then go to frame "AddSum"
end keyDown
```

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on keyUp — event handler

Syntax

```
on keyUp
  statement(s)
end
```

Description This event handler contains statements that are activated when a key is released. The `on keyUp` handler is similar to the `on keyDown` handler.

When a key is released, Lingo searches these locations in order for an `on keyUp` handler: primary event handler, editable text sprite script, script of a text cast member, frame script, and movie script. (For sprites and cast members, `on keyUp` handlers work only for editable text. A `keyUp` on a different type of cast member, such as a bitmap, has no effect.)

Lingo stops searching when it reaches the first location that has an `on keyUp` handler, unless the handler includes the `pass` command to explicitly pass the `keyUp` message on to the next location.

The `on keyUp` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to have occur when the user releases keys.

Where you place an `on keyUp` handler can affect when it runs.

- When you want the handler to apply to a specific editable text sprite, put the handler in a sprite script.
- When you want the handler to apply to an editable text cast member in general, put the handler in a script of the cast member.
- When you want the handler to apply to an entire frame, put the handler in a frame script.
- When you want the handler to apply throughout the entire movie, put the handler in a movie script.

You can override an `on keyUp` handler by placing an alternate `on keyUp` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyUp` handler assigned to a cast member by placing an `on keyUp` handler in a sprite script.

Example This handler checks whether the Return key was released and sends the playback head to another frame if it was:

```
on keyUp
  if the key = RETURN then go to frame "AddSum"
end keyUp
```

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keyDown`, `on mouseDown`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on mouseDown — event handler

Syntax

```
on mouseDown
  statement(s)
end
```

Description

This event handler contains statements that are activated when the mouse button is pressed.

When the mouse button is pressed, Lingo searches these locations in order for an `on mouseDown` handler: primary event handler, sprite script, script of a cast member, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseDown` handler, unless the handler includes the `pass` command to explicitly pass the `mouseDown` message on to the next location.

The `on mouseDown` event handler is a good place to put Lingo that flashes images, triggers sound effects, or makes sprites move when the user presses the mouse button.

Where you place an `on mouseDown` handler can affect when it runs.

- When you want the handler to apply to a specific sprite, put the handler in a sprite script.
- When you want the handler to apply to a cast member in general, put the handler in a script of the cast member.
- When you want the handler to apply to an entire frame, put the handler in a frame script.
- When you want the handler to apply throughout the entire movie, put the handler in a movie script.

You can override an `on mouseDown` handler by placing an alternate `on mouseDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseDown` handler assigned to a cast member by placing an `on mouseDown` handler in a sprite script.

Example This handler checks whether the user clicks anywhere on the stage and sends the playback head to another frame if he or she does:

```
on mouseDown
  if the clickOn = 0 then go to frame "AddSum"
end mouseDown
```

This handler, assigned to a sprite script, plays a sound when the sprite is clicked:

```
on mouseDown
  play "Crickets"
end mouseDown
```

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keyDown`, `on keyUp`, `on mouseUp`, `on startMovie`, `on stepMovie`, and `on stopMovie` event handlers

on mouseUp — event handler

Syntax

```
on mouseUp
  statement(s)
end mouseUp
```

Description This event handler contains statements that are activated when the mouse button is released.

When the mouse button is released, Lingo searches these locations in order for an `on mouseUp` handler: primary event handler, sprite script, script of a cast member, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseUp` handler, unless the handler includes the `pass` command to explicitly pass the `mouseUp` message on to the next location.

An `on mouseUp` event handler is a good place to put Lingo that changes the appearance of objects—such as buttons—after they are clicked. You can do this by switching the cast member assigned to the sprite after the sprite is clicked and the mouse button is released. The sprite's different appearance indicates that the sprite has already been clicked.

Where you place an `on mouseDown` handler can affect when it runs.

- When you want the handler to apply to a specific sprite, put the handler in a sprite script.
- When you want the handler to apply to a cast member in general, put the handler in a script of the cast member.
- When you want the handler to apply to an entire frame, put the handler in a frame script.
- When you want the handler to apply throughout the entire movie, put the handler in a movie script.

You can override an `on mouseDown` handler by placing an alternate `on mouseDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseDown` handler assigned to a cast member by placing an `on mouseDown` handler in a sprite script.

Example This handler, assigned to sprite 10, switches the cast member assigned to sprite 10 when the user releases the mouse button after clicking the sprite:

```
on mouseUp
  puppetSprite 10, TRUE
  set the castNum of sprite 10 to "Dimmed"
end mouseUp
```

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keyUp`, `on mouseDown`, `on startMovie`, `on stepMovie`, and `on stopMovie` movie handlers

on startMovie — event handler

Syntax

```
on startMovie
  statement(s)
end startMovie
```

Description This event handler contains statements that are activated after the movie preloads cast members but before the movie starts playing, regardless of where the playback head is.

An `on startMovie` handler is a good place to put Lingo that opens resource files, creates global variables, initializes variables, plays a sound while the rest of the movie is loading into memory, and checks and adjusts to computer conditions such as color depth.

Example This handler creates global variables and opens two resource files when the movie starts:

```
on startMovie
  global currentScore
  set currentScore = 0
end startMovie
```

See also `on enterFrame`, `on exitFrame`, `on idle`, `on keydown`, `on keyUp`, `on mouseDown`, `on mouseUp`, `on stepMovie`, and `on stopMovie` event handlers

on stepMovie — event handler

Syntax

```
on stepMovie
  statement(s)
end stepMovie
```

Description This handler contains statements that are executed each time the playback head enters a new frame. This handler, which was used in earlier versions of Director, has the same result as the `on enterFrame` handler.

See also `on enterFrame`, `on exitFrame`, `on idle`, `on startMovie`, and `on stopMovie` event handlers; `perFrameHook` property

on stopMovie — event handler

Syntax

```
on stopMovie
  statement(s)
end stopMovie
```

Description

This event handler contains statements that are activated when the movie stops playing.

An `on stopMovie` handler is a good place to put Lingo that performs "cleanup" tasks—such as closing resource files, clearing global variables, erasing text fields, and disposing of objects—when the movie is finished.

Example

This handler clears global variables and closes two resource files when the movie stops:

```
on stopMovie
  set gCurrentScore = 0
  closeResFile "Special Fonts"
  closeResFile "Special Cursors"
end stopMovie
```

See also

`on enterFrame`, `on exitFrame`, `on idle`, and `on startMovie` event handlers

open — command

Syntax

`open {`*whichDocument* `with}` *whichApplication*

Description

This command launches the application specified by the string *whichApplication*. By specifying *whichDocument*, you can specify a document that the application opens at the same time. When either is in a different folder than the current movie, you must specify the pathname.

If you are running MultiFinder, there must be enough memory to run both Macromedia Director and the other application at the same time.

Example This statement opens the MacWrite application:

```
open "MacWrite"
```

This statement opens the MacWrite application, which is in the folder Applications on the drive myDrive, and the document named storyboards:

```
open storyboards with myDrive & "Applications:" ¬
  & MacWrite
```

See also `openDA`, `openResFile`, and `openXlib` commands

open window — command

Syntax `open window "`*whichWindow*`"`

Description This command opens a window that can play a Director movie and brings it to the front of the stage. The window is specified by *whichWindow* and must a have movie already assigned to it before you can use the `open window` command.

Example This statement opens the window Control Panel and brings it to the front:

```
open window "Control Panel"
```

See also `close window` command

openDA — command

Syntax `openDA` *DAname*

Description This command opens the desk accessory specified by *DAName*, which is the menu item name or an expression that yields the menu item name of any desk accessory installed in the computer.

Example This statement opens the Calculator desk accessory:

```
openDA "Calculator"
```

See also closeDA command

openResFile — command

Syntax `openResFile` *whichFile*

Description This command opens the resource file specified by the string *whichFile*. When the file is in a different folder than the current movie, *whichFile* must specify a pathname.

In earlier versions of Director, this command was necessary to make additional fonts and cursors available in your movies. However, you can now provide custom cursors by importing the cursor as a cast member and using the `cursor` property.

When the file is already open, `openResFile` has no effect. It is good practice to close any open file as soon as you are finished using it.

Do not use `openResFile` to open another application. Its code resources interfere with Director's. Use a resource mover like ResEdit to move the resources you need to a separate resource file.

Example This statement opens the resource file Special Fonts:

```
openResFile "Special Fonts"
```

This statement opens the resource file Special Icons, which is in another folder:

```
openResFile pathName:&"Special Icons"
```

See also `closeResFile` and `showResFile` commands; `cursor` property

openXlib — command

Syntax `openXlib` *whichFile*

Description This command opens the Xlibrary file specified by the string expression *whichFile*. If the file is in a different folder than the current movie, *whichFile* must include the pathname.

It is good practice to close any file you have opened as soon as you are finished using it. Do not use `openXlib` to open another application, because its code resources interfere with Director's. When the file is already open, `openXlib` has no effect.

The `openXlib` command also opens HyperCard XCMDs and XFCNs so that you can use them with Director. When you need to use an XCMD from more than one application in a movie, use this command to open a link to the HyperCard stack, rather than install the XCMD in both places with ResEdit. When you do that, a resource conflict that results in a system beep occurs.

Xlibrary files contain XObjects as XCOD resources. Unlike `openResFile`, `openXlib` makes these XObjects known to Director. Using the `mNew` predefined method, you can then create instances of the XObjects in memory.

Example This statement opens the Xlibrary file `VideoDisc Xlibrary`:

```
openXlib "VideoDisc Xlibrary"
```

This statement opens the Xlibrary file `XObjects`, which is in a different folder than the current movie:

```
openXlib "My Drive:New Stuff:Transporter XObjects"
```

See also `closeXlib` and `showXlib` commands

the optionDown — function

Syntax `the optionDown`

Description This function determines whether the Option key is being pressed.

- When the Option key is being pressed, `the optionDown` is TRUE.
- When the Option key is not being pressed, `the optionDown` is FALSE.

Example This handler checks whether the Option key is being pressed and calls the handler named `doOptionKey` if it is:

```
on keyDown
  if the optionDown then doOptionKey(the key)
end keyDown
```

See also `commandDown`, `controlDown`, `key`, and `shiftDown` functions

or — logical operator

Syntax *logicalExpression1* `or` *logicalExpression2*

Description This operator performs a logical OR operation on two logical expressions.

- When either expression or both expressions 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.

Example This statement has the message window display whether at least one of the expressions 1 < 2 and 1 > 2 is TRUE:

```
put 1 < 2 or 1 > 2
```

Because the first expression is TRUE, the result is 1, which is the numerical equivalent of TRUE.

This statement checks whether the contents of the text cast member named field are either AK or HI, and displays an alert if they are:

```
if field "State" = "AK" or field "State" = "HI" ¬
  then alert "You're off the map!"
```

See also `and` and `not` logical operators

the palette of cast — cast property

Syntax

`the palette of cast` *whichCastmember*

Description

This cast property determines which palette is associated with the cast member specified by *whichCastmember.* This property applies to bitmap cast members only.

- When the palette number is a positive number, it refers to another cast member.
- When the palette number is a negative number, it refers to a built–in palette.

The `palette of cast` property can be tested and set.

Example

This statement displays the palette assigned to the cast member Leaves in the message window:

```
put the palette of cast "Leaves"
```

param — function

Syntax

`param(`*parameter*`)`

Description

This function gives the value of a parameter in a list. The variable *parameter* represents the parameter's position in the list.

Example This handler calculates the average value of a list of parameters:

```
on avg first, second, third
  set n = paramCount()
  set sum = 0.0
    repeat with i = 1 to n
      set sum = param(i) + sum
    end repeat
  return sum/n
end avg
```

This statement passes the handler three values and displays the result in the message window:

```
put avg(1,2,3)
-- 2.0
```

See also `paramCount` function

the paramCount function

Syntax `the paramCount`

Description This function determines the number of parameters sent to the current handler.

Example This handler sets the variable named counter to the number of parameters that were sent to the current handler. In this case, the handler was sent the parameters (1, 2, 3):

```
set counter = paramCount()
```

pass command

Syntax `pass`

Description This command passes an event message to the next location in the message hierarchy. Otherwise, an event message stops at the first location that contains a handler for the event.

Passing an event message to other locations in the message hierarchy lets you execute more than one handler for a given event.

Example Used together, these handlers are both activated by a `mouseUp` event because the first handler contains a `pass` command.

This `on mouse Up` handler attached to sprite 3 executes the handler and then passes the `mouseUp` message on:

```
on mouseUp
  if sprite 3 intersects sprite 4 ¬
    then set the text of cast 10 = ¬
    "You placed it correctly"
  pass
end
```

This `on mouseUp` handler in the frame script executes because the `on mouseUp` handler assigned to the sprite script contains the `pass` command:

```
on mouseUp
  go to "Next test"
end
```

See also `dontPassEvent` command

pasteClipBoardInto — command

Syntax `pasteClipBoardInto cast` *whichCastmember*

Description This command pastes the contents of the Clipboard into the cast member specified by *whichCastmember*. When you paste into an occupied cast window slot, the old cast member is completely erased. For instance, pasting a bitmap into a text cast member makes the bitmap the cast member and erases the text cast member.

You can paste any item that is in a format that Director can use as a cast member. When you copy text from another application, the text's formatting is not retained.

The `pasteClipBoardInto cast` command provides a convenient way to copy objects from other movies and from other applications into the cast window.

Example This statement pastes the contents of the Clipboard into the bitmap cast member `Shrine`:

```
pasteClipBoardInto cast Shrine
```

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 This statement checks whether the pathname contains the term `System` and has the computer beep if it does:

```
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 many cases, using `pause` is recommended over looping on the same frame, or looping between two frames. This is because a pause uses much less processor time than repeatedly moving the playback head to the beginning of the frame. Some exceptions to this general rule are when you are moving sprites or are using `the perFrameHook`, so keeping the playback head going to the same frame is required.

The `pause` command is useful for halting the movie while a menu is displayed or for letting the user look at a screen as long as she or he wants.

Example The following `on mouseUp` handler for a button alternately pauses and continues the animation, like the pause button on a videocassette recorder:

```
on mouseUp
  if the pauseState = TRUE then
    continue
  else
    pause
  end if
end mouseUp
```

See also `continue` command; `pauseState` function

the pausedAtStart of cast — digital video cast property

Syntax `the pausedAtStart of cast` *whichDVMovie trueOrFalse*

Description This digital video cast property specifies whether the Paused at Start checkbox in the Digital Video Cast Member Info dialog box is checked or not.

- When the `pausedAtStart of cast` property is TRUE, the Paused at Start checkbox is checked.
- When the `pausedAtStart of cast` property is FALSE, the Paused at Start checkbox is not checked.

The `pausedAtStart of cast` property can be tested and set.

Example This statement turns on the Paused at Start checkbox in the Digital Video Cast Member Info dialog box for the digital video movie Rotating Chair:

```
set the pausedAtStart of cast "Rotating Chair" = TRUE
```

the pauseState — function

Syntax `the pauseState`

Description This function returns TRUE when the movie is currently paused.

Example This statement checks whether the movie is currently paused and has the movie continue if it is:

```
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` property was required in earlier versions of Director. However, you can now achieve the same results by placing Lingo that you want to execute at every frame in an `on enterFrame` or `on exitFrame` handler, or by adding child objects to the `actorList`.

The `perFrameHook` can be used 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 handler) 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 the subframe argument 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.

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.

Example This factory lets you create objects that 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
```

See also `factory` and `method` keywords

pi **function**

Syntax `pi()`

Description This function gives the value of π, the ratio of a circle's circumference to its diameter. The value of π is given as a floating-point number to the number of decimal places set by the `floatPrecision` property.

Example This statement uses the `pi` function as part of an equation for calculating the area of a circle:

```
set vArea = pi()*power(vRadius,2)
```

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 cast member.

The `picture of cast` property can be tested and set.

Example This statement sets the variable named `pict` to the image in the cast member named `Sunset`:

```
put the picture of cast "Sunset" into pict
```

See also `type of sprite` property

pictureP — function

Syntax `pictureP(`*castMember*`)`

Description This function determines whether the cast member specified by *castMember* is a picture data type.

- When the cast member is a picture data type, `pictureP` is TRUE (1).
- When the cast member is not a picture data type, `pictureP` is FALSE (0).

Example This statement has the message window display whether the cast member `Shrine`, which is a bitmap, is a picture data type:

```
put pictureP("Shrine")
```

The result is 1, which is the numerical equivalent of TRUE.

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 be either a string marker label or an integer frame number. The expression *whichMovie* must be a string that specifies a movie file.When the movie is in another folder, *whichMovie* must specify a pathname.

The `play` command is similar to the `go to` command, but with the `play` command, when the sequence being played is over, the playback head automatically returns to the frame where the `play` command was called. If the `play` command is issued from a frame script, the playback head returns to the next frame; if the `play` command comes from a sprite script or handler, 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 given.

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 that,when called from a handler, play the first frame of each movie. The handler is not suspended while the movie plays but immediately continues executing.

Example

This statement moves the playback head to the marker named blink:

```
play "blink"
```

This statement moves the playback head to the next marker:

```
play marker(1)
```

This statement moves the playback head to a separate movie:

```
play movie "My Drive:More Movies:" & newMovie
```

See also

`go` and `play done` 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 frame 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

playFile

See the `sound playFile` command.

point — function

Syntax `point(`*horizontal*`, `*vertical*`)`

Description This function yields a point that has the horizontal coordinate specified by *horizontal* and the vertical coordinate specified by *vertical*.

Example This statement sets the variable `lastLocation` to the point (250, 400):

```
put point(250, 400) into lastLocation
```

See also `rect` function

power — function

Syntax `power(`*base*`,` *exponent*`)`

Description This function calculates the value of the number specified by base to the exponent specified by exponent.

Example This statement sets the variable `vResult` the value of 4 to the third power:

```
set vResult = power(4,3)
```

preLoad — command

Syntax `preLoad`

`preLoad` *toFrameNum*

`preLoad` *fromFrame*, *toFrameNum*

Description This command preloads cast members in the specified frame or range of frames into memory. Preloading stops when memory is full or when all of the specified cast members have been preloaded.

When used without arguments, the `preLoad` command causes a preload of all cast members used from the current frame to the last frame of a movie.

When used with one argument, *toFrame*, the `preLoad` command causes a preload of all cast members used in the range of frames from the current frame to the frame *toFrame*, as specified by frame number or label name.

When used with two arguments, *fromFrame* and *toFrame*, the `preLoad` command causes a preload of all cast members used in the range of frames from the frame *fromFrame* to the frame *toFrame*, as specified by frame number or label name.

The `preLoad` command also returns the number of the last frame successfully loaded. To access this value, use the `result` function.

Example This statement preloads the cast members used up from the current frame to the frame that has the next marker:

```
preLoad marker (1)
```

This statement preloads the cast members used up from frame to the frame 10 to frame 50:

```
preLoad 10, 50
```

See also `preLoad of Cast` command

---

the preLoad of cast — digital video cast property

Syntax `the preLoad of cast` *castMember*

Description This digital video cast property determines whether the digital video cast member specified by *castMember* has been preloaded into memory.

- When the digital video cast member has been preloaded into memory, `the preLoad of cast` is TRUE.
- When the digital video cast member has not been preloaded into memory, `the preLoad of cast` is FALSE.

Example This statement has the message window display whether the digital video movie Rotating Chair has been preloaded into memory:

```
put the preLoad of cast "Rotating Chair"
```

---

preLoadCast — command

Syntax `preLoadCast`

`preLoadCast` *CastNumber*

`preLoadCast` *fromCastNumber*`,` *toCastNumber*

Description This command preloads cast members. Preloading stops when memory is full or when all of the specified cast members have been preloaded.

When used without arguments, the `preLoadCast` command preloads all cast members in the movie.

When used with the *castNumber* argument, the `preLoadCast` command preloads that cast member.

When used with the arguments *fromCastNumber* and *toCastNumber*, the `preLoadCast` command preloads all cast members in the range specified by the cast member numbers or names.

The `preLoadCast` command returns the cast member number of the last cast member successfully loaded. To obtain this value, use the `result` function.

Example This statement preloads cast member 20:

```
preLoadCast 20
```

This statement preloads cast member Shrine and the ten cast members after it in the cast window:

```
set logo to the number of cast “Shrine”
preLoadCast logo, logo + 10
```

the preLoadEventAbort — property

Syntax

```
the preLoadEventAbort
```

Description This property specifies whether pressing keys or clicking the mouse can stop preloading of cast members.

- When the `preLoadEventAbort` property is TRUE, pressing keys or clicking the mouse can stop preloading of cast members.
- When the `preLoadEventAbort` property is FALSE, pressing keys or clicking the mouse cannot stop preloading of cast members.

The default value is FALSE. The setting of this property affects the current movie.

The `preLoadEventAbort` property can be tested and set.

Example This statement lets the user stop preloading of cast members by pressing keys or clicking the mouse:

```
set the preLoadEventAbort = TRUE
```

See also `preLoad` and `preLoadCast` commands

preLoadRAM property

Syntax

```
the preLoadRAM
```

Description This property specifies the amount of RAM that can be used for preloading a digital video movie. It can be set and tested.

This is useful for managing memory, so that digital video cast members are not given more than a certain limit of memory, and other types of cast members can still be preloaded. When the value is set to FALSE, all available memory can be used for preloading digital video cast memebers.

Example This statement allocates the amount of RAM available for preloading three times the size of the cast member Interview:

```
set the preLoadRAM to 3 * (the size of cast ¬
  "Interview")
```

previous

See the `go previous` command.

printFrom — command

Syntax `printFrom` *fromFrame*`{,` *toFrame*`}{,` *reduction*`}`

Description This command prints whatever is displayed on the stage in each frame starting at the frame specified by *fromFrame*. Optionally you can supply the *toFrame*, and the reduction (100, 50, or 25 percent).

When printing at less than 100 percent, the document prints as a bitmap, so text does not print as sharply as it would at full size.

Example This statement prints what is on the stage in every frame starting at frame 1:

```
printFrom 1
```

This statement prints what is on the stage in every frame from the marker `Intro` to the marker `Tale`. The reduction is 50 percent:

```
printFrom label, ("Intro"), ("Tale"), 50
```

property — keyword

Syntax `property {`*property1*`}{,` *property2*`}{,` *property3*`}{...}`

Description This keyword declares that the properties specified by *property1*, *property2*, and so on are property variables. Property variables, which are used in parent scripts, serve the same purpose as instance variables in factories and XObjects.

You declare property variables at the beginning of the parent script. You can access them from outside the parent script by using the `the` operator.

Example This statement allows each child object created from a single parent script to have its own location and velocity setting:

```
property location, velocity
```

See also `ancestor` property

the puppet of 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.

A puppet sprite is controlled by Lingo instead of the score. For example, Lingo can switch the cast member assigned to a sprite or turn on and off whether the sprite is moveable. For more information on using puppets, see “Using Puppets” in Chapter 4.

The sprite channel must contain a sprite before you can make the channel a puppet.

Making the sprite channel a puppet lets you control any of the sprite properties—such as `castNum of sprite`, `locH of sprite`, and `width of sprite`—from Lingo:

Setting the `puppet of 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 `puppetSprite` property can be tested and set. The default value is FALSE.

Example This statement makes the sprite numbered i + 1 a puppet:

```
set the puppet of sprite (i + 1) to TRUE
```

This statement records whether sprite 5 is a puppet by assigning the value of `the puppet of sprite` to the variable. When sprite 5 is a puppet, `isPuppet` is set to TRUE. When sprite 5 is not a puppet, `isPuppet` is set to FALSE:

```
put the puppet of sprite 5 into isPuppet
```

See also `puppetSprite` command

puppetPalette command

Syntax `puppetPalette` *whichPalette*`{,` *speed*`}{,` *nFrames*`}`

Description This command causes the palette channel to act as a puppet. When the palette channel is a puppet, Lingo can override the palette setting in the palette channel of the score and assign palettes to the movie.

The `puppetPalette` command sets the current palette to the palette cast member 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 fade in the palette by replacing *speed* with an integer expression, with 1 being slowest and 60 being fastest. You can also fade in the palette over several frames by replacing *nFrames* with an integer expression for the number of frames.

A puppet palette remains in effect until you turn it off with the command `puppetPalette 0`. No subsequent palette changes in the score are obeyed when the puppet palette is in effect.

Example This statement makes Rainbow the movie's palette:

```
puppetPalette "Rainbow"
```

This statement makes Grayscale the movie's palette. The transition to the Grayscale palette occurs over a time setting of 15 and between frames labeled Gray and Color:

```
puppetPalette customPalette, 15, ¬
  label("Gray") - label("Color")
```

puppetSound — command

Syntax

`puppetSound` *whichCastmember*

`puppetSound 0`

Description

This command makes the sound channel a puppet and plays the sound cast member specified by *whichCastmember*. When the sound is a puppet, Lingo can override any sounds assigned in the first sound channel of the score.

The `puppetSound` command starts playing the specified sound. To play a sound stored in the cast, replace *whichCastmember* with the name of the sound cast member. The sound will not start playing until the playback head moves or until the `updateStage` command is executed.

The statement `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. Use `puppetSound` to restore control of the sound channel to the score.

Puppet sounds can be useful for playing a sound while a different movie is being loaded into memory.

Example

This statement plays the sound Wind under control of Lingo:

```
puppetSound "Wind"
```

See also

`sound fadeIn`, `sound fadeOut`, and `sound playFile`, and `sound stop` commands

puppetSprite command

Syntax `puppetSprite` *whichSprite*, *state*

Description This command controls whether the sprite specified by *whichSprite* is a puppet. When a sprite is a puppet, any sprite property can be controlled by Lingo instead of the score. For example, Lingo can switch the cast member assigned to a sprite or turn on and off whether the sprite is moveable.

- When *state* is TRUE, Lingo controls the sprite and the score is ignored.
- When *state* is FALSE, the sprite is controlled by the score.

The initial properties of the puppet are taken from whatever sprite is in the channel when the `puppetSprite` command is executed. Subsequent control of the sprite properties through Lingo can change these properties.

The channel must contain a sprite when you use the `puppetSprite` command.

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 that aren't intended to be puppets.

For more information on using puppets, see "Using Puppets" in Chapter 4 of *Using Lingo*.

Example This statement makes the sprite in channel 15 a puppet:

```
puppetSprite 15, TRUE
```

This statement removes the puppet condition from the sprite in the channel numbered i + 1:

```
puppetSprite i + 1, FALSE
```

See also `backColor`, `bottom`, castNum, `constraint`, `cursor`, `foreColor`, `height`, `immediate`, `ink`, `left`, `lineSize`, `locH`, `locV`, `puppet`, `right`, `stretch`, `top`, `type`, and `width` sprite properties; `puppetSprite` property

puppetTempo — command

Syntax `puppetTempo` *framesPerSecond*

Description This command causes the tempo channel to act as a puppet. When the tempo channel is a puppet, Lingo can override the tempo setting in the score and change the tempo assigned to the movie.

The `puppetTempo` command sets the tempo to the number of frames specified by *framesPerSecond*. The maximum frames per second is 60.

You do not need to turn off the puppet tempo condition to have subsequent tempo changes in the score take effect.

Example This statement set the movie's tempo to 30 frames per second:

```
puppetTempo 30
```

This statement increases the movie's old tempo by ten frames per second:

```
puppetTempo oldTempo + 10
```

puppetTransition — command

Syntax `puppetTransition` *whichTransition*`{,` *time*`}¬`
`{,` *chunkSize*`}{,` *changeArea*`}`

This command performs the transition specified by *whichTransition* between the current frame and the next frame.

Replace *whichTransition* with one of the following values:

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

Code	Transition	Code	Transition
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 transitions marked with an asterisk (*) in this table will not work on monitors that are set to 32 bits.

Replace *time* with the number of 1/4 seconds used to complete the transition. The minimum is 0; the maximum is 120 (30 seconds). Replace *chunkSize* with the number of pixels in each chunk of the transition. The minimum is 1; the maximum is 128. Smaller chunk sizes give smoother transitions but are slower.

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 takes 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 chunk size, instead of setting a shorter time.

Replace *changeArea* with a value that determines whether the transition occurs only in the changing area. The `changeArea` is an area within which sprites have changed.

- To have the transition occur only in the areas that change, replace *changeArea* with TRUE, which is the default setting.
- To have the transition occur over the entire stage, replace *changeArea* with FALSE.

Example This statement performs a wipe from right transition. Because no value is specified for *changeArea*, the transition occurs only on the changing area, which is the default:

```
puppetTransition 1
```

This statement performs a wipe from right transition that lasts 1 second, has a chunk size of 20, and occurs over the entire stage:

```
puppetTransition 2, 4, 20, FALSE
```

the purgePriority of cast — cast property

Syntax `the purgePriority of cast` *whichCastmember*

Description This cast property specifies the purge priority of the cast member specified by *whichCastmember.*

Cast members' purge priorities determine the priority that Director follows when choosing which cast members to delete from memory when memory is full. The higher the purge priority, the more likely that the cast member is deleted. The following `purgePriority` settings are available:

0	Never purge
1	Purge last
2	Purge next
3	Purge normal

Setting `purgePriority` for cast members is useful for managing memory when the size of the movie's cast exceeds the available memory. As a general rule, you can minimize pauses while the movie loads cast members by assigning a low purge priority to cast members that are frequently used in the course of the movie. This reduces the number of times that Director reloads the cast member when the movie plays.

The Normal setting allows Director to purge cast members from memory at random. The Next, Last, and Never settings allow you some control over purging.

Note *If you set a lot of cast members to Last or Never, your movie may simply run out of memory.*

Example This statement sets the purge priority of cast member Background to 3, which makes it one of the first cast members to be purged when memory is needed:

```
set the purgePriority of cast "Background" to 3
```

put — command

Syntax `put` *expression*

Description This command evaluates the expression specified by *expression* and displays the result in the message window. This can be used as a debugging tool by tracking the values of variables as the movie plays.

Example This statement displays the time in the message window:

```
put the time
-- "9:10 AM"
```

This statement displays the value assigned to the variable `vBid` in the message window:

```
put vBid
-- "Johnson"
```

See also `put…after`, `put…before`, and `put…into` commands; `return`

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 can refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Example This statement adds the string “fox dog cat” after the contents of the variable `animalList`:

```
put "fox dog cat" after animalList
```

See also `char...of`, `item...of`, `line...of`, `put...before`, `put...into`, and `word...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 can refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

Example These statements set the variable named `animalList` to the string “fox dog cat” and then insert the word elk before the second word of the list:

```
put "fox dog cat" into animalList
put "elk " before word 2 of animalList
```

The result is the string `"fox elk dog cat"`.

Note *In the second statement, there is an intentional space between the word elk and the second quote mark. If it were missing, the resulting string would be* `"fox elkdog cat"`.

These statements set the field named Price to the value of 20.00 plus 7.25, and then insert a dollar sign before the number:

```
put (20.00 + 7.25) into field "Price"
put "$" before field "Price"
```

The result in field Price is “$27.25”.

See also `char…of`, `item…of`, `line…of`, `put…after`, `put…into`, and `word…of` chunk expression keywords

put...into — command

Syntax `put` *expression* `into` *variable*

`put` *expression* `into` *chunkExpression*

Description This command has two different usages.

The first usage evaluates a Lingo expression and stores its value in a local, global, property, 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 can refer to any character, word, item, or line in any container of text. Containers include fields (text cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

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.*

Example This statement sets the variable `x` to the square root of 2:

```
put sqrt(2.0) into x
The result is 1.4142.
```

See also `char…of`, `item…of`, `line…of`, `put…after`, `put…before`, and `word…of` chunk expression keywords; `set` command

Q

the quickTimePresent — function

Syntax `the quickTimePresent`

Description This function determines whether the QuickTime extension is currently loaded into memory.

- When the extension is present, the `quickTimePresent` function is TRUE (1).
- When the extension is not present, the `quickTimePresent` function is FALSE (0).

Example This statement determines whether the QuickTime extension is in memory and plays the QuickTime movie Rotating Chair if it is:

```
if the quickTimePresent = 1 then ¬
  play "Rotating Chair"
```

quit — command

Syntax `quit`

Description This command exits from Director or a projector to the Finder.

Example This statement has the computer exit to the Finder when the user presses Command-q:

```
if the key = "q" and the commandDown then quit
```

See also `restart` and `shutDown` 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

This statement inserts quote characters in the string of text:

```
put "Can you spell" && QUOTE & "Macromedia" ¬
  & QUOTE & "?"
```

The result is quotes around the word Macromedia, as in the following string:

```
Can you spell “Macromedia”?
```

R

ramNeeded — function

Syntax `ramNeeded (`*firstFrame*`, `*lastFrame*`)`

Description This function determines, in bytes, the memory needed to display a range of frames. For example, you can test the size of frames containing 32-bit artwork. If the `ramNeeded` is larger than `the freeBytes`, then go to frames containing 8-bit artwork. Divide by 1024 to convert bytes to kilobytes (K).

Example This statement sets the variable `frameSize` to the number of kilobytes needed to display frames 100 to 125 of the movie:

```
put ramNeeded (100, 125) into frameSize
```

This statement determines whether the memory needed to display frames 100 to 125 is more than the available memory and branches to a movie using cast members that have lower color depth if it is:

```
if ramNeeded (100, 125) > freeBytes then ¬
  play frame "8-bit"
```

See also `freeBytes` function; `the size of cast` cast property

random — function

Syntax `random(`*integerExpression*`)`

Description This function returns a random integer from 1 to the value specified by *integerExpression*.

The `random` function is useful when you want to randomly vary values in a movie. Some possible uses are varying the path through a game, assigning random numbers, or changing the color or position of sprites.

Example This statement assigns random values to the variable `diceRoll`:

```
put random(6) + random(6) into diceRoll
```

This statement randomly changes the foreground color of sprite 10:

```
set the foreColor of sprite 10 = random(255)
```

This handler randomly chooses which of two movie segments to play in the “Noh Tale”:

```
on selectMovie
  if random(2) = 2 then play frame "11a"
  else
    play frame "11-b" of movie "NT.OTher Movie"
  end if
end
```

the randomSeed — property

Syntax `the randomSeed`

Description This property specifies seed for generating random numbers. Using the same seed produces the same sequence of random numbers

The `randomSeed` property can be tested and set.

Example This statement displays the random seed number in the message window:

```
put the randomSeed
```

rect function

Syntax `rect(`*left*`, `*top*`, `*right*`, `*bottom*`)`

Syntax `rect(`*point1*`, `*point2*`)`

Description This function has two uses:

- When you use four arguments, the `rect` function defines a rectangle that has the sides specified by *left*, *top*, *right*, and *bottom*. The *left* and *right* values specify numbers of pixels from the left edge of the stage. The *top* and *bottom* values specify numbers of pixels from the top of the stage.
- When you use two arguments, the `rect` function defines a rectangle that encloses the points specified by *point1* and *point2*.

Example This statement sets the variable `newArea` to a rectangle whose left side is at 100, top is at 150, right side is at 300, and bottom is at 400 pixels:

```
put rect(100, 150, 300, 400) into newArea
```

This statement sets the variable `newArea` to the rectangle defined by the points `firstPoint` and `secondPoint`. The coordinates of `firstPoint` are (100, 150); the coordinates of `secondPoint` are (300, 400). Note that this statement creates the same rect as the rectangle created in the previous example:

```
put rect(firstPoint, secondPoint)
```

See also `point` function

the rect of cast — cast property

Syntax `the rect of cast` *whichCastmember*

Description This cast property indicates the left, top, right, and bottom coordinates of the rectangle of a cast member. The coordinates are returned as a rect.

The `rect of cast` property can be tested but not set.

Example This statement displays the coordinates of bitmap cast member 20:

```
put the rect of cast 20
```

the rect of window — window property

Syntax `the rect of window` *whichWindow*

Description This window property determines the left, top, right, and bottom coordinates of the window specified by *whichWindow*. The coordinates are given as a rect.

The `rect of window` property can be tested and set.

Example This statement displays the coordinates of the window Control Panel:

```
put the rect of window "Control Panel"
```

rect point

See the `rect` function.

the regPoint of cast — cast property

Syntax

`the regPoint of cast` *whichCastmember*

Description

This cast property specifies the registration point of a bitmap cast member. The registration points are listed as horizontal and vertical coordinates in a point that has the form `point (horizontal, vertical)`.

The `regPoint of cast` property can be tested and set.

Example

This statement displays the registration points of the bitmap cast member Desk in the message window:

```
put the regPoint of cast "Desk"
```

This statement changes the registration points of the bitmap cast member Desk to the values in the list:

```
set the regPoint of cast "Desk" = ¬
  point (300, 400)
```

See also

`the rect of cast` property

repeat while — keyword

Syntax

```
repeat while testCondition
  {statements…}
end repeat
```

Description

This keyword structure repeatedly executes the statements as long as the condition specified by *testCondition* is TRUE. Some possible uses for this structure are for Lingo that continues to read text strings until the end of a file is reached, checks items until the end of a list is reached, or repeatedly performs an action until the user clicks or releases the mouse button.

Example This handler starts the timer counting, resets the timer to 0, and then has the timer count up to 60 ticks:

```
on countTime
  startTimer
  repeat while the timer < 60
    -- waiting for timer
  end repeat
end countTime
```

See also `exit`, `exit repeat`, and `repeat with` keywords

repeat with — keyword

Syntax

`repeat with` *counter* `=` *start* `to` *finish*

  {*statements…*}

`end repeat`

Description This keyword structure executes the Lingo specified by *statements* the number of times specified by *counter*. The value of *counter* is the difference between the value specified by *start* and the value specified by *finish*. The counter is incremented by 1 each time Lingo goes through the repeat loop.

The `repeat with` structure is useful for repeatedly applying the same effect to a series of puppets or calculating a series of numbers, such as a number to some exponent.

Example The following handler turns sprites 1 through 30 into puppets:

```
on puppetize
  repeat with channel = 1 to 30
    puppetSprite channel, TRUE
  end repeat
end puppetize
```

See also `exit`, `exit repeat`, and `repeat while` keywords

repeat with...down to — keyword

Syntax `repeat with` *variable* = *startValue* `down to` *endValue*

Description This keyword counts down by increments of 1 from *startValue* to *endValue*.

Example This handler contains a repeat loop that counts down from 20 to 15:

```
on countDown
  repeat with i = 20 down to 15
  set the castNum of sprite 6 to (10 + i)
  updateStage
end repeat
```

repeat with...in list — keyword

Syntax `repeat with` *variable* `in` *someList*

Description This keyword assigns successive values from the specified list to the variable.

Example This statement displays four values in the message window:

```
repeat with x in [1, 2, 3, 4]
  put x
end repeat
```

restart — command

Syntax

```
restart
```

Description This command restarts the computer. It is equivalent to choosing Restart in the Finder's Special menu.

Example This statement restarts the computer when the user presses Command-r:

```
if the key = "r" and the commandDown then restart
```

See also `quit` and `shutDown` commands

the result — function

Syntax

```
the result
```

Description This function gives the value of the return expression in the last handler executed.

The `result` function is useful for obtaining values from movies that are playing in windows and tracking Lingo's progress by displaying results of handlers in the message window as the movie plays.

Example The following handler returns a random roll for two dice:

```
on diceRoll
  return random(6) + random(6)
end diceRoll
```

Example 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 because there are no parentheses following `diceRoll`; `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 be an integer, floating-point number, string, object, or symbol.

When calling a handler or method that 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 under the entry "`the result`".

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
```

If 3 and 7 were used for a and b, the result would be as follows:

```
put max(3, 7)
-- 7
```

See also `result` keyword

RETURN — character constant

Syntax `RETURN`

Description This character constant represents the return key.

Example This statement has a paused movie continue when the user presses the Return key:

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

This statement uses the Return character constant to insert a return between two lines in an alert:

```
alert "Last line in the file." & RETURN & ¬
  "Click OK to exit."
```

the right of sprite — sprite property

Syntax `the right of sprite` *whichSprite*

Description This sprite property indicates the number of pixels that the right edge of the sprite specified by *whichSprite* is from the left edge of the stage.

The `right of sprite` property can be tested, but not set directly. The right horizontal coordinate of a sprite can be set using the `spriteBox` command.

Example This statement calls the handler `offRightEdge` when the right edge of sprite 3 is past the right edge of the stage:

```
if the right of sprite 3 > (the stageRight ¬
  - the stageLeft) then offRightEdge
```

See also `bottom`, `height`, `left`, `locH`, `locV`, `top` and `width` sprite properties; `spriteBox` command

rollOver — function

Syntax `rollOver(`*whichSprite*`)`

Description This function indicates whether the cursor is currently over the bounding rectangle of the sprite specified by *whichSprite*.

- When `rollOver` is TRUE (1), the cursor is currently over the sprite.
- When `rollOver` is FALSE (0), the cursor is not currently over the sprite.

The `rollOver` function is typically used in frame scripts. It is useful for creating handlers that perform an action when the user places the cursor over a specific sprite or simulating additional sprite channels by splitting the stage into regions that send the playback head to a different frame that subdivides the region for the available sprite channels.

Example This statement changes the content of text cast member `Message` to "This is the place." when the cursor is over sprite 6:

```
if rollOver(6) then ¬
put "This is the place." into field "Message"
```

This handler sends the playback head to different frames when the cursor is over certain sprites on the stage. The three sprites in this case could be invisible rectangles in different parts of the stage. Putting additional subdivisions within each of the frames lets you work with more sprites than there are available channels:

```
on enterFrame
  if rollOver(1) then go to frame "Left"
  if rollOver(2) then go to frame "Middle"
  if rollOver(3) then go to frame "Right"
end enterFrame
```

See also `mouseCast` function

the romanLingo — property

Syntax `the romanLingo`

Description This property specifies whether Lingo uses a single-byte or double-byte interpreter.

- When `the romanLingo` is TRUE, Lingo uses a single-byte interpreter.
- When `the romanLingo` is FALSE, Lingo uses a double-byte interpreter.

The Lingo interpreter is faster with single-byte character sets. Some versions of Macintosh system software—Japanese, for example—use a double-byte character set. U.S. system software uses a single-byte character set. Normally, `the romanLingo` is set when starting up Director and is determined by the local version of Macintosh system software.

If you are using a non-roman script system but don't use any double-byte characters in your script, set this property to TRUE to get faster execution of your Lingo scripts.

Example This statement sets the `romanLingo` to TRUE, which has Lingo use a single-byte character set:

```
set the romanLingo to TRUE
```

saveMovie — command

Syntax `saveMovie {`*pathname*`:`*filename*`}`

Description This command saves the current movie. Including the optional parameter saves the movie to the file specified by *pathname*:*filename*.

Example This statement saves the current movie to the file Update:

```
saveMovie "Update"
```

the scoreColor of sprite — sprite property

Syntax `the scoreColor of sprite` *whichSprite*

Description This sprite property indicates the score color assigned to the sprite specified by *whichSprite*. The possible values correspond to color chips 0 to 5 in the current palette.

The `scoreColor of sprite` property can be tested but not set.

Example This statement has the message window display the value for the score color assigned to sprite 7:

```
put the scoreColor of sprite 7
```

the script of menuItem — menu property

Syntax `the script of menuItem` *whichItem* `of menu` *whichMenu*

Description This menu item property determines which Lingo statement is executed when the specified menu item is selected. The *whichItem* expression can be either a menu item name or a menu item number; the *whichMenu* expression can be 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.

Example This statement makes the handler named `goHandler` the handler that is executed when the user chooses the command `Go` from the custom menu Control:

```
set the script of menuItem "Go" of menu ¬
  "Control" to "goHandler"
```

See also `checkMark` and `enabled of menuItem` properties; `installMenu` command; `menu`: keyword

scriptNum of sprite — sprite property

Syntax `scriptNum of sprite` *whichSprite*

Description This sprite property indicates the number of the script assigned to the sprite specified by *whichSprite*.

The `scriptNum of sprite` property can be tested, but not set.

Example This statement displays the number of the script attached to sprite 4:

```
put the scriptNum of sprite 4
```

the scriptText of cast — cast property

Syntax `the scriptText of cast` *whichCastmember*

Description This cast property indicates the text of the script, if any, assigned to the cast member specified by *whichCastmember.*

The `scriptText of cast` property can be tested and set.

Example This statement makes the contents of text cast member 20 the script of cast member 30:

```
set the scriptText of cast 30 = the text of cast 20
```

the searchCurrentFolder — function

Syntax

```
the searchCurrentFolder
```

Description This function determines whether Director searches the current folder when searching filenames.

- When the `searchCurrentFolder` function is TRUE (1), Director searches the current folder when resolving filenames.
- When the `searchCurrentFolder` function is FALSE (0), Director does not search the current folder when resolving filenames.

The `searchCurrentFolder` function can be tested and set.

Example This statement has the message window display whether the `searchCurrentFolder` function is on:

```
put the searchCurrentFolder
```

The result is the number 1, which is the numeric equivalent of TRUE.

This statement sets the `searchCurrentFolder` function to TRUE, which has Director search the current folder when resolving filenames:

```
set the searchCurrentFolder to TRUE
```

the searchPath — function

Syntax `the searchPath`

Description This function provides a list of the pathnames that are searched when Director resolves filenames. When Director cannot find the file in the current folder, it searches for it in the folders listed in `the searchPath`.

The `searchPath` function can be tested but not set.

Example This statement displays the pathnames that Director searches when resolving filenames:

```
put the searchPath
```

See also `the searchCurrentFolder` function

the 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.

The `selection` function only determines which text is selected; you cannot use `the selection` to select text.

Example This statement checks whether any text is selected and displays the alert “Please select a word.” if none is:

```
if the selection = EMPTY then ¬
  alert "Please select a word."
```

See also `selEnd` and `selStart` properties

the selEnd — text property

Syntax `the selEnd`

Description This text property specifies the ending character of a selection. It is used with `the selStart` to determine a selection from the currently editable text, counting from the beginning character.

The `selEnd` text property can be tested and set, and the default value is 0.

Example These statements select “cde” from the text “abcdefg”:

```
set the selStart to 3
set the selEnd to 5
```

This statement calls the handler `noSelection` when `the selEnd` is the same as `the selStart`:

```
if the selEnd = the selStart then noSelection
```

This statement makes a selection 20 characters long:

```
set the selEnd to the selStart + 20
```

See also `editableText` and `hilite` commands; `selection` function; `selEnd` and `text` text properties

the selStart — text property

Syntax `the selStart`

Description This text property specifies the starting character of a selection. It is used with `the selEnd` to determine a selection from the currently editable text, counting from the beginning character.

The `selStart` text property can be tested and set. The default value is 0.

Example These statements select “cde” from the text “abcdefg”:

```
set the selStart to 3
set the selEnd to 5
```

This statement calls the handler `noSelection` when `the selEnd` is the same as `the selStart`:

```
if the selEnd = the selStart then noSelection
```

This statement makes a selection 20 characters long:

```
set the selEnd to the selStart + 20
```

See also `editableText` and `hilite` commands; `selection` function; `selEnd` and `text` text properties

set...to and set...= — command

Syntax `set the` *property* `to` *expression*

set the *property* = *expression*

set *variable* to *expression*

set *variable* = *expression*

Description This command evaluates the expression specified by *expression* and puts the result into the property specified by *property* or the variable specified by *variable.*

Example This statement sets the ink effect for sprite 3 to the ink effect specified by the number 8:

```
set the ink of sprite 3 to 8
```

This statement sets the `soundEnabled` property to the opposite of its current state. When `the soundEnabled` is TRUE (the sound is on), this statement turns it off. When `the soundEnabled` is FALSE (the sound is off), this statement turns it on.

```
set the soundEnabled = not (the soundEnabled)
```

This statement sets the variable named `vowels` to the string “aeiou”:

```
set vowels to "aeiou"
```

See also `property` and `instance` keywords

setaProp — command

Syntax `setaProp` *list*, *property*, *newValue*

Description This command replaces the value assigned to *property* with the value specified by *newValue* in the list specified by *list*. When the property is not already in the list, Lingo adds the new property and value.

Example These statements create a property in the given list, and then change that property:

```
set x = [#a:1]
setaProp x, #a, 2
put x
-- [#a:2]
```

setAt — command

Syntax `setAt` *list*, *orderNumber*, *value*

Description This command replaces the item specified by *orderNumber* with the value specified by *value* in the list specified by *list*.

- When *orderNumber* is greater than the number of items in a linear list, the list is expanded with blank entries to provide the number of places specified by *orderNumber*.
- When *orderNumber* is greater than the number of items in a property list, an error alert occurs.

Example This handler assigns a name to the list [12, 34, 6, 7, 45], replaces the fourth item in the list with the value 10, and then displays the result in the message window:

```
on enterFrame
  set vNumbers = [12, 34, 6, 7, 45]
  setAt vnumbers, 4, 10
  put vNumbers
end enterFrame
```

When the handler runs, the message window displays the following:

```
 [12, 34, 6, 10, 45]
```

setCallBack command

Syntax `setCallBack` *XCMDname*, *value*

Description This command specifies how Lingo handles unsupported callbacks from the HyperTalk XCMD or XFCN specifed by *XCMDname*.

◆ When *value* is TRUE (1), unsupported callbacks from the specified XCMD or XFCN cause a generic alert to be displayed.

◆ When *value* is FALSE (0), unsupported callbacks from the specified XCMD or XFCN are ignored.

◆ When *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.

Example This statement has Lingo ignore unsupported callbacks from the SuperDuperXCMD command:

```
setCallBack superDuperXCMD, 0
```

setProp — command

Syntax `setProp` *list, property, newValue*

Description This command replaces the value assigned to property with the value specified by newValue in the list specified by list. This command is similar to the `setaProp` command, except that this command gives an error when the property is not already in the list.

Example This statement changes the age property of property list x to 11:

```
set x = [#age:10, #sex:0]
setProp x, #age, 11
```

See also `setaProp` command

the shiftDown — function

Syntax `the shiftDown`

Description This function indicates whether the user is pressing the Shift key.

- When the `shiftDown` is TRUE, the user is pressing the Shift key.
- When the `shiftDown` is FALSE, the user is not pressing the Shift key.

Example This statement checks whether the Shift key is being pressed and calls the handler `doShiftKey` if it is:

```
if the shiftDown then doShiftKey (the key)
```

See also `commandDown`, `controlDown`, `key`, and `optionDown` functions

short

See the `date` and `time` functions.

showGlobals — command

Syntax `showGlobals`

Description This command has the message window display all global variables and factories, including XObjects. It is useful for debugging scripts.

See also `clearGlobals` and `showLocals` commands; `global` keyword

showLocals — command

Syntax `showLocals`

Description This command has the message window display all local variables. This command can only be used within handlers, parent script, or factory methods.

Local variables in handlers are abandoned after the handler executes. This command is useful for debugging scripts.

See also `clearGlobals`, `showGlobals` commands; `global` keyword

showResFile — command

Syntax `showResFile {`*whichFile*`}`

Description This command displays a list of resources in the resource file specified by the string *whichFile*. The file must already be open. If the resource file is in a different 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.

Example This statement displays the resource file Special Fonts:

```
showResFile "Special Fonts"
```

See also `closeResFile`, `openResFile`,`openXlib`, and `showXlib` commands

showXlib — command

Syntax `showXlib {`*Xlibfilename*`}`

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`:

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 displays the on-line documentation for that XObject.

Example This statement displays the XObjects in the VideoDisc Library:

```
showXlib "VideoDisc Xlibrary"
```

See also `closeXlib` and `openXlib` commands

shutDown — command

Syntax `shutDown`

Description This command causes the computer to close all open applications and turn itself off. This does the same thing as the Shut Down command in the Finder's Special menu.

Example This statement checks whether the user has pressed Command-q and shuts down the computer if he or she has:

```
if the key = "q" and the commandDown then shutDown
```

See also `quit` and `restart` commands

sin — function

Syntax `sin(`*angle*`)`

Description This function calculates the sine of the specified angle. The angle must be expressed in radians as a floating-point number.

Example The following statement calculates the sine of pi/2:

```
sin (pi()/2.0) = 1
```

Note that the symbol π cannot be used in a Lingo expression.

See also `cos` and `tan` functions

the size of cast — cast property

Syntax `the size of cast` *castName*

Description This cast property permits you to learn the size, in bytes, of a specific cast member number or name. Divide bytes by 1024 to convert to kilobytes.

Example This statement displays the size of the Shrine cast member in the message window:

```
put the size of cast "Shrine" into field "How Big"
```

sort — command

Syntax `sort` *list*

Description This command puts the items in the list specified by *list* into alphanumeric order.

- When the list is a linear list, the list is sorted by values.
- When the list is a property list, the list is sorted alphabetically by properties.

Once a list is sorted, it maintains its sort order even when you add new variables using the `add` command.

Example This statement puts the list `Values`, which consists of [#a: 1, #d: 2, #c: 3], into alphanumeric order. The result appears below the statement:

```
put values
-- [#a: 1, #d: 2, #c: 3]
sort Values
put Values
--[#a: 1, #c: 3, #d: 2]
```

soundBusy — function

Syntax `soundBusy(`*whichChannel*`)`

Description This function determines whether a sound is playing in the sound channel specifed by *whichChannel*.

- When a sound is playing in the specified sound channel, the `soundBusy` function is TRUE (1).
- When no sound is playing in the specified sound channel, the `soundBusy` function is FALSE (0).

Make sure that you allow enough time for the sound to start playing before using `soundBusy` to check the sound channel.

Example This statement checks whether a sound is playing in sound channel 1 and loops in the frame if it is. This would allow the sound to finish before the playback head goes to another frame:

```
if soundBusy(1) then go to the frame
```

See also `sound playFile` and `sound stop` commands

sound close — command

Syntax `sound close` *soundChannel*

Description This command stops the sound playing in and then closes the sound channel specified by *soundChannel*.

Example This statement stops any sound playing in and closes sound channel 1:

```
sound close 1
```

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. When you set this property to FALSE, the volume setting of the sound is not changed but you do not hear the sound.

Example This statement sets `the soundEnabled` property to the opposite of its current setting. It turns the sound on if it is off and turns it off if it is on:

```
set the soundEnabled to not (the soundEnabled)
```

See also `the soundLevel`, `the volume of sound`, `the volume of sprite` properties

sound fadeIn — command

Syntax `sound fadeIn` *whichChannel*

Syntax `sound fadeIn` *whichChannel*, *ticks*

Description This command fades in a sound in the specified sound channel over a period of frames or ticks.

- When ticks is specified, then the fade in occurs evenly over that period of time.
- When ticks is not specified, the default number of ticks is calculated as 15 * (60 / (Tempo setting)) based on the Tempo setting for the first frame of the fade in.

The fade in continues at a predetermined rate until the number of ticks has elapsed, or the sound in the specified channel changes.

Example This statement fades in the sound in channel 1 over 5 seconds:

```
sound fadeIn 1, 5 * 60
```

See also `sound fadeOut` command

sound fadeOut — command

Syntax `sound fadeOut` *whichChannel*

Syntax `sound fadeOut` *whichChannel*, *ticks*

Description This command fades out a sound in the specified sound channel over a period of frames or ticks.

- When ticks is specified, then the fade out occur s evenly over that period of time.
- When ticks is not specified, the default number of ticks is calculated as 15 * (60 / (Tempo setting)) based on the Tempo setting for the first frame of the fade out.

The fadeout continues at a predetermined rate until the number of ticks has elapsed, or the sound in the specified channel changes.

Example This statement fades in the sound in channel 1 over 5 seconds:

```
sound fadeIn 1, 5 * 60
```

See also `sound fadeIn` command

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).

Now that Macintosh computers can produce multichannel sound, this property is becoming obsolete. It is better to use the `volume of sound` property for controlling the sound volume on a channel-by-channel basis.

The `soundLevel` property can be tested and set. The default value is 7.

Example This statement sets the variable `oldSound` equal to the current sound level:

```
put the soundLevel into oldSound
```

This statement sets the sound level to 5:

```
set the soundLevel to 5
```

See also `soundEnabled` property; `volume of sound` sound property

the sound of cast — digital video cast property

Syntax `the sound of cast` *castMember* `to` *onOrOff*

Description This cast property controls the audio output of the digital video cast member specifed by *castMember*.

- When `the sound of cast` is set to 1, sound for the cast member is turned on.
- When `the sound of cast` is set to 0, sound for the cast member is turned off.

Example This statement turns on the sound for the cast member Movie Clip:

```
set the sound of cast "Movie Clip" to 1
```

sound playFile — command

Syntax `sound playFile` *whichChannel*, *whichFile*

Description This command plays the AIFF sound located at *whichFile* in the sound channel specified by *whichChannel*.

When the sound file is in a different folder than the movie, *whichFile* must specify the full pathname to the file.

The `sound playFile` command streams files from disk rather than playing them from RAM the way Director plays sound cast members. As a result, using the `sound playFile` command when playing digital video or when loading cast members into memory can cause conflicts when the computer tries to read the disk in two places at once.

This command requires System 6.0.7 or later to work; otherwise the sound playback will not occur.

Example This statement plays the file named Thunder in channel 1:

```
sound playFile 1, "Thunder"
```

This statement plays the file named Thunder in channel 3:

```
sound playFile 3, the pathName &"Thunder"
```

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 This statement checks whether a sound is playing in sound channel 1 and stops the sound if it is:

```
if soundBusy(1) then sound stop 1
```

See also `soundBusy` function

---

the sourceRect of window — window property

Syntax `the sourceRect of window` *whichWindow*

Description This window property specifies the coordinates of the rectangle that the movie that plays in the window specified by *whichWindow* was originally created for.

Example This statement displays the original coordinates of the movie Control Panel in the message window:

```
put the sourceRect of "Control Panel"
```

sprite — keyword

Syntax `the` *property* `of sprite` *whichSprite*

Description This keyword tells Lingo that the value specified by *whichSprite* is a sprite number. It is used with every sprite property.

A sprite is an occurrence of a cast member in an animation channel of the score.

Example This statement sets the variable named `horizontal` to `the locH of sprite 1`:

```
put the locH of sprite 1 into horizontal
```

This statement turns on the puppet condition for the sprite that has sprite number i + 1:

```
set the puppet of sprite (i + 1) to TRUE
```

See also `cast` keyword; `puppetSprite` command

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 This statement checks whether two sprites intersect and changes the contents of the text cast member Notice to "You placed it correctly." if they do:

```
if sprite i intersects j then ¬
put "You placed it correctly." into field "Notice"
```

See also `sprite...within` comparison operator

sprite...within — comparison operator

Syntax `sprite` *sprite1* `within` *sprite2*

Description This comparison 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 This statement checks whether two sprites intersect and calls the handler `doInside` if they do:

```
if sprite 3 within 2 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 of sprite` property must be TRUE to use this command.

A sprite's coordinates change based on their registration points. For bitmap sprites, it may be necessary to move the registration points in order to obtain proper results.

Example This statement sets the coordinates of sprite 3's bounding rectangle to 50, 50, 200, and 250:

```
spriteBox 3, 50, 50, 200, 250
```

This statement sets the bounding rectangle of the sprite whose number is `mySprite` to the starting values and the current cursor location. This creates a rectangle that stretches from the specifed point to the mouse cursor:

```
spriteBox mySprite, ¬
  startH, startV, the mouseH, the mouseV
```

See also `bottom`, `height`, `left`, `rect of cast`, `right`, `stretch`, `top`, and `width` sprite properties; `puppetSprite` and `updateStage` command

the sqrt — function

Syntax `the sqrt(`*number*`)`

Description This function yields the square root of the number specified by *number*.

- When *number* is a floating-point number, the result is a floating-point number.
- When *number* is an integer, the result is rounded to the nearest integer.

Example This statement displays the square root of 3.0 in the message window:

```
put sqrt(3.0)
-- 1.7321
```

See also `the floatPrecision` property

the stage — system property

Syntax

```
the stage
```

Description

This system property is used to refer to the main movie in commands and functions that relate to windows. This is useful when using the `tell` command to send a message to the main movie from a child movie.

Example

This statement causes the main movie to stop animating:

```
tell the stage to pause
```

This statement displays the current setting of the stage:

```
put the rect of the stage
--rect (0, 0, 640, 480)
```

the stageBottom — function

Syntax

```
the stageBottom
```

Description

This function—along with `the stageLeft`, `the stageRight`, and `the 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`.

The `stageBottom` function can be tested but not set.

Example

These two statements position sprite 3 a distance of 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
```

See also

`stageLeft`, `stageRight`, and `stageTop` functions; `the locH of sprite` and `the locV of sprite` sprite properties

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 color palette 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. The default value is 0 (white).

Example This statement sets the variable `oldColor` to the index number of the current stage color:

```
put the stageColor into oldColor
```

This statement sets the stage color to the color assigned to chip 249 on the current palette:

```
set the stageColor to 249
```

See also `backColor of sprite` and `foreColor of sprite` properties

the stageLeft — function

Syntax `the stageLeft`

Description This function—along with `the stageRight`, `the stageTop`, and `the stageBottom`—indicates where the stage is positioned on the desktop. It equals the left horizontal coordinate of the stage, relative to the upper left corner of the main screen. When the stage is flush with the left side of the main screen, this coordinate is zero.

The `stageLeft` function can be tested but not set.

Example This statement checks whether the left edge of the stage is beyond the left edge of the screen and calls the handler `leftMonitorProcedure` if it is:

```
if the stageLeft < 0 then leftMonitorProcedure
```

See also `stageBottom`, `stageRight`, and `stageTop` functions; `the locH of sprite` and `the locV of sprite` sprite properties

the stageRight — function

Syntax

```
the stageRight
```

Description This function—along with `the stageLeft`, `the stageTop`, and `the 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`.

The `stageRight` function can be tested but not set.

Example These two statements position sprite 3 a distance of 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
```

See also `stageBottom`, `stageLeft`, and `stageTop` functions; `the locH of sprite` and `the locV of sprite` sprite properties

the stageTop — function

Syntax

```
the stageTop
```

Description This function—along with `the stageBottom`, `the stageLeft`, and `the 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.

The `stageTop` function can be tested but not set.

Example This statement checks whether the top of the stage is beyond the top of the screen and calls the handler `upperMonitorProcedure` if it is:

```
if the stageTop < 0 then upperMonitorProcedure
```

See also `stageBottom`, `stageLeft`, and `stageRight` functions

startMovie

See the `on startMovie` event handler.

starts — comparison operator

Syntax *string1* `starts` *string2*

Description This comparison 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).

The string comparison is not sensitive to case or diacritical marks; “a” and “Å” are considered the same.

This is a comparison operator with a precedence level of 1.

Example This statement has the message window display whether the word Macromedia starts with the string Macro:

```
put "Macromedia" starts "Macro"
```

The result is 1, which is the numerical equivalent of TRUE.

See also `contains` comparison operator

the startTime of sprite — digital video sprite property

Syntax `the startTime of sprite` *spriteNumber*

Description This sprite property sets the beginning of a digital video movie in the specified sprite channel. The value of `the startTime` is measured in ticks.

When a digital video movie is played, `the startTime` determines where playback begins.

Example This statement has the digital video movie in sprite channel 5 start playing at 100 ticks into the movie:

```
set the startTime of sprite 5 to 100
```

See also `the duration of cast` cast property; `the movieRate of sprite` and `the movieTime of sprite` sprite properties

startTimer — command

Syntax `startTimer`

Description This command sets the `timer` property to zero.

Example This handler set the timer to zero when a key is pressed:

```
on keyDown
  startTimer
end keyDown
```

See also `the timer` property

stepMovie

See the `on stepMovie` event handler.

the stillDown — function

Syntax `the stillDown`

Description This function indicates whether the user is pressing the mouse button.

- When the user is pressing the mouse button, `the stillDown` is TRUE.
- When the user is not pressing the mouse button, `the stillDown` is FALSE.

This function is useful within a `mouseDown` script to trigger certain events only after the `mouseUp`.

Lingo cannot test `the stillDown` when it is used inside a repeat loop. Use `the mouseDown` function inside of repeat loops instead.

Example This statement checks whether the mouse button is being pressed and calls the handler `dragProcedure` if it is:

```
if the stillDown then dragProcedure
```

See also `the mouseDown` function

stop

See the `sound stop` command.

stopMovie

See the `on stopMovie` event handler.

the stopTime of sprite — sprite property

Syntax `the stopTime of sprite` *whichSprite*

Description This property specifies the end of a digital video movie that has the sprite number specified by *spriteNumber*. The value of `the stopTime` is measured in ticks.

When a digital video movie is played, `the stopTime` is where playback halts or loops if the `loop` property is turned on.

Example This statement has the digital video movie in sprite channel 5 stop playing at 1500 ticks into the movie:

```
set the stopTime of sprite 5 to 1500
```

See also `the movieRate of sprite`, `the movieTime of sprite`, and `the startTime of sprite` sprite properties

the stretch of sprite — sprite property

Syntax `the stretch of sprite` *whichSprite*

Description This sprite property determines whether the sprite specified by *whichSprite* can be stretched by using the `spriteBox` command or the `width of sprite` and `height of sprite` properties. If it is True, the bitmap sprite can be stretched.

The `stretch of sprite` property can be tested and set, and the default value is FALSE. When FALSE, the sprite always stays at its default or normal size.

The `stretch of sprite` property applies to bitmap, digital video, and picture cast members, but not to shape, text, or button cast members. Shapes can be stretched at any time by setting their `height of sprite` and `width of sprite` properties, regardless of the setting of their `stretch` property. Text and button cast members cannot be stretched in any case.

Director requires much more processor time to draw stretched sprites than regular sprites, which can affect movie performance.

In order to have its properties set using Lingo, the sprite must be a puppet.

Example This statement checks whether sprite 3 is stretchable and sets the sprite's width to 10 pixels if it is:

```
if the stretch of sprite 3 = TRUE then ¬
  set the width of sprite 3 to 10
```

See also `spriteBox` and `updateStage` commands; `height of sprite` and `width of sprite` properties

string — function

Syntax `string(`*expression*`)`

Description This function converts an integer, floating-point, or symbol expression to a string.

Example This statement adds 2 + 2 and has the message window display the results:

```
put string(2 + 2)
```

This statement converts the number 123 to a string:

```
put string(123)
-- "123"
```

See also `value` function

stringP — function

Syntax `stringP(`*expression*`)`

Description This function determines whether the expression specifed by *expression* is a string.

- When the expression is a string, the result is TRUE.
- When the expression is not a string, the result is FALSE.

The "P" in `stringP` stands for predicate.

Example This statement checks whether "3" is a string:

```
put stringP("3")
```

The result is 1, which is the numeric equivalent of TRUE.

This statement checks whether the floating-point number 3.0 is a string:

```
put stringP(3.0)?-- 0
```

Because 3.0 is a floating-point number and not a string, the result is 0, which is the numeric equivalent of FALSE.

See also `floatP`, `ilk`, `integerP`, `objectP`, and `symbolP` functions

the switchColorDepth — property

Syntax `the switchColorDepth`

Description This property determines whether Director automatically switches the color depth when loading a movie.

- When `the switchColorDepth` is TRUE, Director switches the monitor(s) that the stage occupies to the color depth of the movie that is being loaded.
- When `the switchColorDepth` is FALSE, Director leaves the color depth of the monitor(s) unchanged when a movie is loaded.

When `the switchcolorDepth` is TRUE, nothing happens until a new movie is loaded.

Setting the monitor's color depth to that of the movie is good practice.

- When the monitor's color depth is set below that of the movie, resetting it to the color depth of the movie (assuming that the monitor can provide that color depth) helps maintain the movie's original appearance.
- When the monitor's color depth is higher than that of the movie, reducing the color depth lets you use the minimum amount of memory to play movies. At minimum memory, loading cast members is more efficient and animation can occur faster.

The `switchColorDepth` property can be tested and set. The default value is the setting for the Switch Monitor's Color Depth to Match Movie's checkbox in the Preferences dialog box.

Example This statement sets the variable named `switcher` to the current setting of `switchColorDepth`:

```
put the switchColorDepth into switcher
```

This statement checks whether the current color depth is 8-bit and turns the `switchColorDepth` property on if it is:

```
if the colorDepth = 8 then ¬
  set the switchColorDepth to TRUE
```

See also `colorDepth` property; `colorQD` function

symbolP function

Syntax `symbolP(`*expression*`)`

Description This function determines whether the expression specified by *expression* is a symbol.

- When the expression is a symbol, the result is TRUE.
- When the expression is not a symbol, the result is FALSE.

The "P" in `symbolP` stands for predicate.

Example This statement checks whether #3 is a string:

```
put stringP(#3)
```

TAB — character constant

Syntax TAB

Description This character constant represents the Tab key.

Example This statement checks whether the character typed is the Tab character and calls the handler `doNextField` if it is:

```
if the key = TAB then doNextField
```

This statement advances or retreats the playback head on Tab and Shift-Tab:

```
if the key = TAB then
  if the shiftDown then
    go the frame -1
  else
    go the frame +1
  end if
end if
```

See also `BACKSPACE`, `EMPTY`, and `RETURN` character constants

tan — function

Syntax `tan(`*angle*`)`

Description This function yields the tan of the specified angle. The angle must be expressed in radians as a floating-point number. A radian is an arc in a circle, equal in length to the radius. It is 57.295 degrees. There are 2π or 6.2833 radians in a circle.

Example The following function yields the tangent of pi()/4:

```
tan (pi()/4.0) = 1
```

Note that the π symbol cannot be used in a Lingo expression.

See also `atan`, `cos`, `pi`, `sin`

tell — command

Syntax `tell` *object* `to` *statement*

Syntax

```
tell object
   statement(s)
end tell
```

Description This command communicates the statement or statements specified by *statement(s)* to the object specified by *object*.

The `tell` command is useful for allowing movies to interact. It can be used within a main movie to send a message to a movie playing in a window, or to send a message from a movie playing in a window to the main movie. For example, the `tell` command can let a button in a control panel call a handler in a movie playing in a window. The movie playing in a window could react to the first movie handler by executing the handler. The movie playing in the window could interact with the main movie by sending some value back to the movie.

When you use the `tell` command to send a message to a movie playing in a window, it is important to use the full path name or the window number (in the *windowList*) as the object name. Because opening and closing windows may change the order of the *windowList*, it is a good idea to store the full path name as a global variable. When you do, you can close the window in the `stopMovie` handler of the main movie.

Example This statement has the window Control Panel instruct the movie Simulation to branch to another frame:

```
tell window "Simulation" to go to frame "Save"
```

These statements instruct the movie playing in the childMovie window to continue playing:

```
global childMovie
put the pathName & "Simulation" into childMovie
open window childMovie
tell window childMovie to continue
```

In this set of statements, the `tell` command sends a series of instructions to the movie playing in the childMovie window. Note that a multiple-line `tell` command resembles a handler. It needs an `end` statement:

```
global childMovie
tell window childMovie
  go to frame 5
  set the stageColor to 100
  set the castNum of sprite 4 to 45
  updateStage
end tell
```

In this example, the `tell` command instructs the movie playing in the childMovie window to execute the `calcBalance` handler, to put the result into the `myBalance` global variable, and to display the result in the message window:

```
global childMovie
tell window childMovie to calcBalance
put the result into myBalance
put myBalance
-- $17,300
```

Example When you use the `tell` command to send a message from a movie playing in a window to the main movie, use the system property `the stage` as the object name:

```
tell the stage to go to frame "Main menu"
```

Note *When you use the tell command to call a handler in another movie, make sure that you do not have a handler by the same name in the same script in the local movie. If you do, the local script will be called. This applies only to handlers in the same script in which you are using the tell command.*

---

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 text cast member specifed by *whichCastmember.*

The `text of cast` property is useful for displaying messages and recording what the user types.

The `text of cast` property can be tested and set.

Note that any Lingo change to the text of a cast member removes any special formatting you have applied to individual words or lines. Altering `the text of cast` reapplies global formatting.

Example This statement places the phrase “Thank you.” in the empty text cast member Response:

```
if the text of cast "Response" = EMPTY then ¬
  set the text of cast "Response" to "Thank You."
```

This statement sets the text of cast member Notice to “You have made the right decision.”

```
set the text of cast "Notice" = "You have ¬
  made the right decision!"
```

See also `selEnd` and `selStart` text properties; & and && text operators

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 cast member.

The value of the property is a string consisting of one of the following: "left," "center," or "right." The parameter *whichField* can be either a cast name or a cast number.

The `textAlign of field` property can be tested and set.

The text cast member must contain text, if only a space, to use the `textAlign of field` property. It has no effect on a cast member that contains no text.

Example This statement sets the variable named `alignment` to the current `textAlign of field` setting for the text cast member Rokujo Speaks:

```
put the textAlign of field "Rokujo Speaks" into ¬
  alignment
```

This repeat loop consecutively sets `the textAlign` of the text cast member Rove to left, center, and then right.

```
repeat with i = 1 to 3
  set the textAlign of field "Rove" ¬
  to word i of "left center right"
end repeat
```

See also `text` cast property; `textFont of field`, `textHeight of field`, `textSize of field`, and `textStyle of field` text properties

the textFont of field — text property

Syntax `the textFont of field` *whichField*

Description This text property determines the typeface of the font used to display the specified text cast member. The parameter *whichField* can be either a cast member name or number.

The `textFont of field` text property can be set, affecting every line in the text field. When tested, it returns the height of the first line of text.

The text cast member must contain text, if only a space, to use the `textFont of field` property. It has no effect on a cast member that contains no text.

Example This statement sets the variable named `oldFont` to the current `textFont of field` setting for the text cast member Rokujo Speaks:

```
put the textFont of field "Rokujo Speaks" into oldFont
```

See also `text` cast property; `textAlign of field`, `textHeight of field`, `textSize of field`, and `textStyle of field` 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 cast member. The parameter *whichField* can be either a cast member name or number.

The `textHeight of field` property can be tested and set.

The text cast member must contain text, if only a space, to use the `textHeight of field` property. It has no effect on a cast member that contains no text.

Example This statement sets the variable named `oldHeight` to the current `textHeight of field` setting for the text cast member Rokujo Speaks:

```
put the textHeight of field "Rokujo Speaks" into ¬
  oldHeight
```

See also `text` cast property; `textAlign of field`, `textFont of field`, `textSize of field`, and `textStyle of field` 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 cast member. The parameter *whichField* can be either a cast member name or number.

The `textSize` text property can be tested and set.

The text cast member must contain text, if only a space, to use the `textSize of field` property. It has no effect on a cast member that contains no text.

Example This statement sets the variable named `oldSize` to the current `textSize of field` setting for the text cast member Rokujo Speaks:

```
put the textSize of field "Rokujo Speaks" into ¬
  oldSize
```

See also `text` cast property; `textAlign of field`, `textFont of field`, `textHeight of field`, and `textStyle of field` 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 cast member.

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 that are currently applied. The parameter *whichField* can be either a cast member name or number.

The text cast member must contain text, if only a space, to use the `textStyle of field` property. It has no effect on a cast member that contains no text.

The `textStyle of field` text property can be tested and set.

Example This statement sets the variable named `oldStyle` to the current `textStyle of field` setting for the text cast member Rokujo Speaks:

```
put the textStyle of field "Rokujo" into oldStyle
```

This statement sets the textStyle of field setting for the text cast member Rokujo Speaks to bold italic:

```
set the textStyle of field "Poem" to "bold, italic"
```

See also `text` cast property; `textAlign of field`, `textFont of field`, `textHeight of field`, and `textSize of field` text properties

the — keyword

Syntax `the` *property*

Description All Lingo properties and many sprite properties and 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 they are changed by system events). Sprite properties would change when a new movie is loaded.

then — keyword

See the `if…then` keyword.

the ticks — function

Syntax `the ticks`

Description This function returns the current time in ticks (60ths of a second). Counting ticks begins from the time the computer is started.

Example This statement converts ticks to minutes by dividing the number of ticks by 60 twice and then sets the variable `minutesOn` to the result:

```
put the ticks/60/60 into minutesOn
```

See also `time` function; `timer` property

the time — function

Syntax `the abbr time`

Syntax `the abbrev time`

Syntax `the abbreviated time`

Syntax `the long time`

Syntax `the short time`

Syntax `the 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.

Example These statements have the message window display the time in different formats. Possible results appear below each statement:

```
put the abbreviated time
"1:30 PM"
```

```
put the long time
"1:30:24 PM"
```

```
put the short time
"1:30 PM"
```

Note *The three time formats vary, depending on the country for which your System file was designed. The examples given in this entry are for the United States.*

See also `date` function

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. The default value is TRUE.

Example

This statement sets the variable timing to the value of `the timeoutKeyDown`:

```
put the timeoutKeyDown into timing
```

This statement turns off `the timeoutKeyDown`:

```
set the timeoutKeyDown to FALSE
```

See also

`the 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 and set.

Example

This statement sets the text of field Countdown to the value of the `timeoutLapsed` property. (Dividing `the timeOutLapsed` by 60 converts it to seconds):

```
put the timeoutLapsed / 60 into field "Countdown"
```

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. The default value is 10,800 ticks, which is 3 minutes.

Example

This statement sets `the timeOutLength` to 10 seconds:

```
set the timeoutLength to 10 * 60
```

the timeoutMouse property

Syntax

```
the timeoutMouse
```

Description

This property determines whether `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. The default value is TRUE.

Example

This statement records the current setting of `the timeOutMouse` by setting the variable named `timing` to `the timeOutMouse`.

```
put the timeoutMouse into timing
```

This statement sets `the timeoutMouse` property to FALSE. The result is that the `timeoutLapsed` property keeps its current value when the mouse button is pressed:

```
set the timeoutMouse to FALSE
```

See also

`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, playing 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. The default value is FALSE.

Example

This statement sets `the timeoutPlay` to TRUE, which has Lingo reset the `timeoutLapsed` property to zero when a movie is played:

```
set the timeoutPlay to true
```

See also

`the timeoutLapsed` property

the timeoutScript — property

Syntax

```
the timeoutScript
```

Description

This property determines the string that is executed as a Lingo statement when a timeout occurs.

Setting the `timeOutScript` property is equivalent to executing a `when timeOut then` command that was used in earlier versions of Director.

When the event script you've assigned is no longer appropriate, turn it off with the statement `set the timeOutScript to EMPTY.`

The `timeOutScript` property can be tested and set. The default value is EMPTY.

Example

This statement sets `the timeoutScript` to a calling script for the handler `timeoutProcedure`:

```
set the timeoutScript to "timeoutProcedure"
```

the timer property

Syntax `the timer`

Description This property is a free-running timer that counts time in ticks (60ths of a second). It has nothing to do with `the timeOutScript`. It is only for convenience in timing certain events. The `startTimer` command zeroes the value of the `timer` property.

The `timer` property is useful for setting up delays within handlers. (The `delay` command works only in frame scripts.) For example, you can use `the timer` to synchronize pictures to a sound track by inserting a delay that makes the movie wait until a sound is finished.

Example This handler creates a 1 second delay:

```
on countTime
  startTimer
  repeat while the timer < 60
    nothing
  end repeat
end countTime
```

This statement sets the variable `startTicks` to the current value of `the timer`:

```
set the timer = startTicks
```

See also `lastClick`, `lastEvent`, `lastKey`, and `lastRoll` functions; `startTimer` command

the title of window — window property

Syntax `the title of window` *whichWindow*

Description This window property is the title of the window specified by *whichWindow*.

The `title of window` property can be tested and set.

Example This statement makes Action View the title of window X:

```
set the title of window "X" to "Action View"
```

the titleVisible of window — window property

Syntax `the titleVisible of window` *whichWindow*

Description This window property specifies whether the window specified by *whichWindow* displays the window title in the window's title bar.

The `titleVisible of window` property can be tested and set.

Example This statement display the title of the window Control Panel by setting the window's `titleVisible` property to TRUE:

```
set the titleVisible of "Control Panel" to TRUE
```

to — keyword

The word `to` occurs in a number of Lingo constructs.

See `char…of`, `item…of`, `line…of`, and `word…of` chunk expression keywords; `repeat with, set…to`, and `set…=` commands.

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 *whichSprite*. The coordinate is the number of pixels from the upper left corner of the stage.

The `top of sprite` property can be tested, but not set directly. The top vertical coordinate of a sprite can be set with the `spriteBox` command.

Example This statement checks whether the top of sprite 3 is above the top of the stage and calls the handler `offTopEdge` if it is:

```
if the top of sprite 3 < 0 then offTopEdge
```

See also `bottom`, `height`, `left`, `locH`, `locV`, `right`, and `width` sprite properties; `spriteBox` command

the trace — property

Syntax `the trace` *trueOrFalse*

Description This property specifies whether the movie's trace function is on or off.

- When `the trace` is TRUE (1), the trace function is on.
- When `the trace` is FALSE (0), the trace is off.

Example This statement turns the trace function on:

```
set the trace = TRUE
```

the traceLoad property

Syntax `the traceLoad`

Description This property specifies the amount of information that is displayed about cast members as they are loaded. The possible values for the `traceLoad` property have the following effect:

0	Displays no information
1	Displays cast members' names
2	Display cast members' names, number of the current frame, movie name, and file seek offset

The `traceLoad` property can tested and set.

Example This statement has the movie display the names of cast members as they are loaded:

```
set the traceLoad to 1
```

the traceLogFile property

Syntax `the traceLogFile`

Description This property specifies the name of the file that the message window display is written to. You can close the file by setting the `traceLogFile` property to EMPTY ("").

Example This statement has Lingo write the display of the message window to the file messages:

```
set the traceLogFile = "Messages"
```

This statement closes the file that the message window display is being written to:

```
set the traceLogFile = ""
```

the trails of sprite — sprite property

Syntax `the trails of sprite` *whichSprite*

Description This property turns the trails ink effect on and off for the sprite specified by *whichSprite*. In order to set this property, the sprite must have the `puppetSprite` property set to TRUE before setting the `trails` property. Set `the trails` to 0 to turn trails off; set `the trails` to 1 to turn trails on.

To erase trails:

- Animate another sprite across these pixels.
- Use a transition.

Example This statement sets the trails on for sprite 7:

```
set the trails of sprite 7 to 1
```

See also `the directToStage` cast property

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 This statement turns on the `soundEnabled` property by setting it to TRUE:

```
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 the sprite specified by *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 cast member, or to replace one button sprite with another type, or to make it invisible on the stage.

The sprite types are as follows:

0	inactive sprite (turns the sprite off)
1	bitmap
2	rectangle
3	rounded rectangle
4	oval
5	line topleft to bottomright
6	line bottomleft to ropright
7	text
8	button
9	checkbox
10	radio button
16	Undetermined. Use `castType` property to examine the type of cast member associated with the sprite.

Before setting a sprite to type 1 (bitmap), set the `stretch of sprite` property of the sprite to FALSE. This prevents it from stretching to the size of the previous sprite.

When 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. When you are changing several sprite properties—or several sprites—you only have to use one `updateStage` command at the end of all the changes.

The `type of sprite` property can be tested and set.

Example This statement sets the type of sprite 4 to text:

```
set the type of sprite 4 to 7
```

This statement turns off sprite 1:

```
set the type of sprite 1 to 0
```

See also `stretch of sprite` property

U

union rect — function

Syntax `union rect` *rect1*, *rect2*

Description This function returns the smallest rectangle that encloses the two rectangles *rect1* and *rect2*.

Example

```
put union (rect (0, 0, 10, 10), ¬
rect (15, 15, 20, 20))
-- rect (0, 0, 20, 20)
```

unLoad — command

Syntax

`unLoad`

`unLoad` *theFrameNum*

`unLoad` *fromFrameNum*, *toFrameNum*

Description This command clears the cast members used in a specified frame from memory. When used without an argument, the `unload` command clears the cast members in all the frames of a movie from memory.

When used with one argument, *theFrameNum*, the `unLoad` command clears from memory the cast members in that frame. director automatically unloads the least recently used cast members to accommodate `preLoad` commands or normal cast loading.

When used with two arguments, *fromFrameNum* and *toFrameNum*, the `unLoad` command unloads all cast members in the range specified. You can specify a range of frames by frame numbers or frame labels.

Example This statement clears the cast members used in frame 10 from memory:

```
unLoad 10
```

This statement clears the cast members used from the frame labeled first to the frame labeled last:

```
unLoad "first","last"
```

See also `preLoad`, `preLoadCast`, and `unLoadCast` commands; the `purgePriority of cast` cast property

unLoadCast — command

Syntax `unLoadCast`

Syntax `unLoadCast` *castName*

Syntax `unLoadCast` *fromCastName, toCastName*

Description This command clears the specified cast members from memory. When used without an argument, `unLoadCast` causes all cast members in a movie to be cleared from memory—except for any being used in the current frame.

When used with one argument, *castName*, the `unLoadCast` command clears from memory the cast member name or number that you specify.

When used with two arguments, *fromCastName* and *toCastName*, the `unLoadCast` command unloads all cast members in the range specified.

Example This statement clears the cast member Screen 1:

```
unLoadCast "Screen1"
```

This statement clears from memory all cast members from cast member 11 to cast member 18:

```
unloadCast 11, 18
```

See also `preLoad` and `preLoadCast` commands; the `purgeLevel of cast` cast property

updateMovieEnabled — property

Syntax

```
the updateMovieEnabled
```

Description

This property specifies whether changes made to the current movie are automatically saved when the movie branches to another movie.

- When the `saveChanges` property is TRUE, changes to the movie are automatically saved when the movie branches to another movie.
- When the `saveChanges` property is FALSE, changes to the movie are not automatically saved when the movie branches to another movie.

The default value is FALSE.

Example

This statement has Director save changes to the current movie whenever the movie branches to another movie.

```
set the updateMovieEnabled = TRUE
```

updateStage — command

Syntax

```
updateStage
```

Description

This command redraws the stage immediately. Normally the stage is updated only between frames, but the `updateStage` command updates the stage any time the command is executed from a handler or factory method.

The `updateStage` command is useful for creating animation within one frame, which is common when animating puppets.

Do not use `updateStage` with the `perFrameHook` property. Otherwise, unexpected results could occur.

Example This handler makes the sprite specifed by whichSprite a puppet sprite, changes the sprite's horizontal and vertical locations, and redraws the stage so that the sprite appears in the new location:

```
on moveRight whichSprite, howFar
  puppetSprite whichSprite, TRUE
  set the locH of sprite whichSprite ¬
    to the locH of sprite whichSprite + howFar
  updateStage
end moveRight
```

value — function

Syntax `value(`*string*`)`

Description This function returns the numerical value of a string. This is useful when making use of a numerical string that the user has typed into a text cast member or data from XObjects that return numerical strings.

Example This statement displays the numerical value of the string “the sqrt of” && “2.0”:

```
put value("the sqrt of" && "2.0")
```

The result is 1.4142.

This statement displays the numerical value of the string “penny”:

```
put value("penny")
```

The resulting display in the message window is <VOID>, because the word penny has no numerical value.

See also `string` function

version — system variable

Syntax `global version`

Description This system variable contains the version string for Macromedia Director. The same string appears the the Finder’s Get Info dialog box.

Example This statement displays the version of Macromedia Director in the message window:

```
put version
-- "4.0"
```

the video of cast — digital video cast property

Syntax `the video of` *castName*

Description This cast property enables or disables playing the video that is associated with the cast member.

Example This statement turns off the vidio associated with the Interview cast member:

```
set the video of cast "Interview" to FALSE
```

the visible of sprite — sprite property

Syntax `the visible of sprite` *whichSprite*

Description This sprite property determines whether the sprite specified by *whichSprite* is visible.

- When the `visible of sprite` property is TRUE, the sprite is visible.
- When the `visible of sprite` property is FALSE, the sprite is not visible.

The `visible of sprite` property can be tested and set.

Example This statement makes sprite 8 visible:

```
set the visible of sprite 8 to TRUE
```

the visible of window — window property

Syntax `the visible of window` *whichWindow*

Description This window property determines whether the window specified by *whichWindow* is visible.

- When the `visible of window` property is TRUE, the window is visible.
- When the `visible of window` property is FALSE, the window is not visible.

The `visible of window` property can be tested and set.

Example This statement makes the window Control Panel visible:

```
set the visible of window "Control Panel" to TRUE
```

voidP — function

Syntax `voidP(`*variableName*`)`

Description This function specifies whether the variable specified by *variableName* has been given an initial value.

- When the result is TRUE, the variable has not been given an initial value.
- When the result is FALSE, the variable has been given an initial value.

Example This statement checks whether the variable answer has been given an initial value:

```
put voidP(answer)
```

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 *whichChannel*. Sound channels are numbered 1, 2, 3, ... 1 and 2 are the channels that appear in the score.

The value of the `volume of sound` property ranges from 0 (silent) to 255 (maximum volume).

Example This statement sets the volume of sound channel number i to 130, which is a medium setting:

```
set the volume of sound i to 130
```

See also `sound FadeIn` and `sound FadeOut commands`; `soundEnabled` and `soundLevel` properties

the volume of sprite — digital video sprite property

Syntax `the volume of sprite` *spriteNum*

Description This property can be used to control the volume of a digital video movie cast member. You can use a cast name or number. The values for volume range from -256 to 256. Values of zero or less are silent.

Example This statement sets the volume of the digital video movie playing in sprite channel 7 to 256, which is the maximum sound volume:

```
set the volume of sprite 7 to 256
```

See also `soundLevel` property

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`.

Example This statement causes the computer to beep when the key is pressed:

```
when keyDown then beep
```

This statement causes the movie to advance to the end when the Return key is pressed:

```
when keyDown then ¬
  if the key = return then go to frame "ending"
```

This statement turns off the `keyDown` action:

```
when keyDown then nothing
```

Note *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`.

This command performs the same function as `the mouseDownScript` property.

Example This statement causes the computer to beep when the mouse is pressed:

```
when mouseDown then beep
```

This statement causes the movie to advance to the end when the mouse and option key are pressed:

```
when mouseDown then ¬
  if the optionDown then go to frame "ending"
```

This statement turns off the `mouseDown` action:

```
when mouseDown then nothing
```

Note *The* `mouseDown` *action is automatically turned off when you load a new movie.*

See also `dontPassEvent` command, `the 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.

Example This statement causes the movie to beep when the mouse is released:

```
when mouseUp then beep
```

This statement causes the movie to advance to the end when the mouse and option key are released:

```
when mouseUp then ¬
  if the optionDown then go to frame "ending"
```

This statement turns off the `mouseUp` action:

```
when mouseUp then nothing
```

Note *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 Macromedia 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
```

Example This example causes the movie to advance to the help frame when the user has not responded within the sepcified time:

```
when timeOut then go to frame "help"
```

This statement cancels a previous `when...then` command:

```
when timeOut then nothing
```

Note *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.*

Related Variables dontPassEvent command, timeoutKeydown, timeoutLapsed, timeoutLength, timeoutMouse, timeoutPlay properties

while

See the `repeat while` keyword.

the width of cast — cast property

Syntax `the width of cast` *whichCastmember*

Description This cast property determines the width in pixels of the cast member specified by *whichCastmember*. The `width of cast` applies only to bitmap and shape cast members. It does not affect text or button cast members.

Example This statement assigns the width of cast member 50 to the variable `height`:

```
put the width of cast 50 into height
```

See also `the height of cast` property

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 *whichSprite*. The width applies only to bitmap and shape cast members. It does not affect text or button cast members.

The `width of sprite` property can be tested and set.

Setting this property has no effect on bitmap sprites unless the sprite's `stretch of sprite` property is set to TRUE.

When you set this property within a script while the playback head is not moving, be sure to use the `updateStage` command to redraw the stage. When you are changing several sprite properties—or several sprites—you have to use only one `updateStage` command at the end of all the changes.

Example This statement sets the width of sprite 10 to 26 pixels:

```
set the width of sprite 10 to 26
```

This statement assigns the width of sprite number i + 1 to the variable `howWide`:

```
put the width of sprite (i + 1) into howWide
```

See also `height of sprite` and `stretch of sprite` properties; `spriteBox` command

window — keyword

Syntax `window` *whichWindow*

Description This keyword refers to the movie window—a window that contains a Director movie—specified by *whichWindow*.

Windows that play movies are useful for creating floating palettes, separate control panels, and windows of different shapes. By using windows that play movies, you can have several movies open at once and allow them to interact.

Example This statement opens the window Control Panel:

```
open window "Control Panel"
```

This statement moves the window Control Panel to the front:

```
moveToFront window "Control Panel"
```

See also `close window`, `moveToBack`, `moveToFront`, and `open` window commands; `the drawRect of window`, `the fileName of window`, `the modal of window` , `the rect of window`, `the sourceRect of window`, `the title of window`, `the titleVisible of window`, `the visible of window`, and `the windowType of window` window properties

the windowList property

Syntax

```
the windowList
```

Description This property is a list of all the known movie windows.

Example This statement displays all the known movie windows in the message window:

```
put the windowList
```

This statement clears the windowList:

```
set the windowList = []
```

the windowType of window — window property

Syntax `the windowType of window` *whichWindow*

Description This window property specifies the display style of the window specified by *whichWindow*. The possible values are 0 to 16, which correspond to the Standard Tool Box numbers.

0	standard document window
1	alert box style window
2	plain box
3	plain box with shadow
4	document window without size box
8	document window with zoom box
12	document window with zoom box, but without size box
16	window with curved border

Example This statement sets the value of the display style of the window Control Panel to 8:

```
set the windowType of window "Control Panel" to 8
```

with

See the `repeat with` keyword.

within

See the `sprite…within` comparison operator.

word...of — chunk expression keyword

Syntax `word` *whichWord* `of` *chunkExpression*

Syntax `word` *firstWord* `to` *lastWord* `of` *chunkExpression*

Description This chunk expression keyword specifies 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—such as a Tab or Return—is considered a space.)

The expressions *whichWord*, *firstWord*, and *lastWord* must evaluate to integers that specify a word in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of text. Sources of text include fields (text cast members) and variables that hold strings.

Example These statements set the variable named `animalList` to the string "fox dog cat" and then insert the word elk before the second word of the list:

```
put "fox dog cat" into animalList
put "elk " before word 2 of animalList
```

The result is the list "fox elk dog cat".

This statement has the message window display the fifth word of the same string:

```
put word 5 of "fox elk dog cat"
```

Because there is no fifth word in this string, the message window displays two quote marks (" "), which indicate an empty string.

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.

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 *whichLibrary*. The XLibrary must have been previously opened with the `openXlib` command. If you specify EMPTY for *whichLibrary*, 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.

Example This statement displays the XObjects available in the Xlibrary named AppleCD XObj:

```
put xfactoryList("AppleCD XObj")
```

This statement displays the first line of the list of all available XObjects in all open Xlibraries:

```
put line 1 of xfactoryList(EMPTY)
```

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*.

The *delayTicks* argument is the delay in ticks between each movement of the zoom rectangles. If *delayTicks* is not specified, the delay is 1.

Example This statement creates a zoom effect between sprites 7 and 3:

```
zoomBox 7, 3
```

Appendix A

Lingo Changes

This appendix lists the new Lingo elements in Director 4, and Lingo that is no longer supported. For a complete description of each element, consult the alphabetical Lingo Dictionary.

New Lingo

Lingo element	Category
[ ]	list operator
abort	command
the actorList	property
add	command
addAt	command
addProp	command
alert	command
ancestor	property
append	command
atan	function
the backColor of cast	cast property
birth	function

New Lingo

Lingo element	Category
the blend of sprite	sprite property
the castType of cast	cast property
the center of cast	digital video cast property
clearGlobals	command
the clickLoc	function
close window	command
the controller of cast	digital video cast property
copyToClipBoard	command
cos	function
count	function
the crop of cast	digital video cast property
deleteAt	command
deleteProp	command
the depth of cast	cast property
the directToStage of cast	digital video cast property
down	keyword
the drawRect of window	window property
duplicate cast	command
the duration of cast	cast property
the editableText of sprite	sprite property
enterFrame	event handler
erase cast	command
exitFrame	event handler
exp	function

New Lingo

Lingo element	Category
the fileName of window	window property
findEmpty	function
findPos	function
findPosNear	function
float	function
the foreColor of cast	cast property
forget window	command
the frameLabel	frame property
the framePalette	frame property
the frameRate of cast	digital video cast property
the frameScript	frame property
the frameTempo	frame property
getaProp	function
getAt	function
getLast	function
getNthFileNameInFolder	function
getOne	function
getPos	function
getProp	function
getPropAt	function
go loop	command
go next	command
go previous	command
halt	command
the height of cast	cast property

New Lingo

Lingo element	Category
ilk point	function
ilk rect	function
importFileInto	command
inflate rect	function
inside	function
intersect	function
the itemDelimiter	property
keyUp	function
the keyUpScript	property
the last	function
the lastFrame	property
list	function
listP	function
the loaded of cast	cast property
log	function
loop	keyword
the loop of cast	digital video cast property
map	function
max	function
maxInteger	function
mci	command
min	function
mInstanceRespondsTo	predefined method
the modal of window	window property
the modified of cast	cast property

New Lingo

Lingo element	Category
moveToBack	command
moveToFront	command
movieFileFreeSize	function
movieFileSize	function
the movieName	function
the moviePath	function
the movieRate of sprite	digital video sprite property
the movieTime of sprite	digital video sprite property
multiSound	property
next	keyword
next repeat	keyword
offset rect	function
on enterFrame	event handler
on exitFrame	event handler
on keyDown	event handler
on keyUp	event handler
open window	command
the palette of cast	cast property
param	function
the paramCount	function
pass	command
pasteClipBoardInto	command
the pausedAtStart of cast	Digital video cast property
pi	function
pictureP	function

New Lingo

Lingo element	Category
power	function
the preLoad of cast	digital video cast property
the preLoadEventAbort	system property
preLoadRAM	property
property	keyword
the purgePriority of cast	cast property
the quickTimePresent	function
ramNeeded	function
the randomSeed	property
rect	function
the rect of cast	cast property
the rect of window	window property
the regPoint of cast	cast property
repeat with...down to	keyword
repeat with...in list	keyword
the romanLingo	property
saveMovie	command
the scoreColor of sprite	sprite property
scriptNum of sprite	sprite property
the scriptText of cast	cast property
the searchCurrentFolder	function
the searchPath	function
setaProp	command
setAt	command
setProp	command

New Lingo

Lingo element	Category
the size of cast	cast property
sort	command
sound close	command
the sound of cast	digital video cast property
the sourceRect of window	window property
stage	system property
the startTime of sprite	digital video sprite property
the stopTime of sprite	digital video sprite property
tan	function
tell	command
the title of window	window property
the titleVisible of window	window property
the trace	property
the traceLoad	property
the traceLogFile	property
the trails of sprite	sprite property
union rect	function
unLoad	command
unLoadCast	command
updateMovieEnabled	property
the video of cast	digital video cast property
the visible of sprite	sprite property
the visible of window	window property
voidP	function
the volume of sprite	digital video sprite property

New Lingo

Lingo element	Category
the width of cast	cast property
window	keyword
the windowList	property
the windowType of window	window property

Director 4 introduces changes in Lingo syntax. Director 4 automatically updates some outdated syntax when you open an old movie. Also, when you check Allow Outdated Lingo in the Movie Info dialog box, Director does accept some outdated Lingo. However, you cannot use outdated Lingo when you create new scripts in Director 4.

Outdated Lingo

Lingo element	Category	Current Use
A11...H88	cast identifier	Octal cast identifiers are accepted only when Allow Outdated Lingo is checked in the Movie Info dialog box. To convert from octal to decimal cast identifiers, use the conversion chart in Appendix D or the Cast ID Style option in the Cast Window Options dialogue box.
the immediate of sprite	sprite property	Use `on mouseDown` in a sprite script or cast member script.
macro	keyword	Macros have been replaced by handlers and scripts. Use the `on` keyword to define a handler. See *Using Lingo* for a complete discussion of handlers. Director 4 automatically updates macros when you open an old movie.
on stepMovie	movie handler	Use `on enterFrame`. Check Allow Outdated Lingo in the Movie Info dialog box to recognize this handler.
playAccel	command	Becuase of performance improvements built into Director 4, the Accelerator application has been eliminated. This command is no longer used.

Appendix B

ASCII Character Chart

This appendix shows the ASCII equivalents for 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	
	12	18	dc2	
	13	19	dc3	
	14	20	dc4	
	15	21	nak	
	16	22	syn	

Character	Hex	Decimal	Name	Keystrokes on US keyboard
	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		/

Character	Hex	Decimal	Name	Keystrokes on US keyboard
0	30	48		0
1	31	49		1
2	32	50		2
3	33	51		3
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

Character	Hex	Decimal	Name	Keystrokes on US keyboard
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
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

Character	Hex	Decimal	Name	Keystrokes on US keyboard
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
x	78	120		x
y	79	121		y
z	7a	122		z

Character	Hex	Decimal	Name	Keystrokes on US keyboard
{	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-~ (tilde), 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-~ (tilde), e
ê	90	144		Option-I, e
ë	91	145		Option-U, e
í	92	146		Option-E, i
ì	93	147		Option-~ (tilde), i

Character	Hex	Decimal	Name	Keystrokes on US keyboard
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-~ (tilde), o
ô	99	153		Option-I, o
ö	9a	154		Option-U, o
õ	9b	155		Option-N, o
ú	9c	156		Option-E, u
ù	9d	157		Option-~ (tilde), 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

Character	Hex	Decimal	Name	Keystrokes on US keyboard
≠	ad	173		Option-= (equal sign)
Æ	ae	174		Option-Shift-" (quote)
Ø	bf	175		Option-Shift-O
∞	b0	176		Option-5
±	b1	177		Option-Shift-=
≤	b2	178		Option-, (comma)
≥	b3	179		Option-. (period)
¥	b4	180		Option-Y
µ	b5	181		Option-M
∂	b6	182		Option-D
∑	b7	183		Option-W
∏	b8	184		Option-Shift-P
π	b9	185		Option-P
∫	ba	186		Option-B
ª	bb	187		Option-9
º	bc	188		Option-0
Ω	bd	189		Option-Z
æ	be	190		Option-" (quote)
ø	bf	191		Option-O
¿	c0	192		Option-Shift-?
¡	c1	193		Option-I
¬	c2	194		Option-L
√	c3	195		Option-V
ƒ	c4	196		Option-F
≈	c5	197		Option-X

Character	Hex	Decimal	Name	Keystrokes on US keyboard
∆	c6	198		Option-O
«	c7	199		Option-\ (backslash)
»	c8	200		Option-Shift-\
…	c9	201		Option-; (semicolon)
	ca	202	(fixed space)	Option-Spacebar
À	cb	203		Option-~ (tilde), 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- - (hyphen)
—	d1	209	(m-dash)	Option-Shift- - (hyphen)
“	d2	210		Option-] (right bracket)
”	d3	211		Option-Shift-] (right bracket)
‘	d4	212		Option-[(left bracket)
’	d5	213		Option-Shift-[(left bracket)
÷	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 in Helvetica.

Character	Hex	Decimal	Name	Keystrokes on US keyboard
Ÿ	d9	217		Option-Shift-~ (tilde)
⁄	da	218		Option-Shift-1
¤	db	219		Option-Shift-2
‹	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

Character	Hex	Decimal	Name	Keystrokes on US keyboard
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-; (semicolon)
Û	f3	243		Option-Shift-Z
Ù	f4	244		Option-Shift-X
ı	f5	245		Option-Shift-B
ˆ	f6	246		Option-Shift-N
˜	f7	247		Option-Shift-M
¯	f8	248		Option-Shift-, (comma)
˘	f9	249		Option-Shift-. (period)
˙	fa	250		Option-H
¸	fb	251		
˝	fc	252		
˛	fd	253		
ˇ	fe	254		

Appendix C

Lingo Quick Reference

This appendix provides a list of commonly used Lingo, grouped by category.

Categories

Cast members

Cast window management

Code structures

Constants

Digital video

Event handlers and messages

Events

External to Director

Factories, methods, and XObjects

Keyboard

Lists

Logical operators and functions

Menus

Mouse and pointer

Movie in a window

Operators and math functions

Output

Parent scripts

Playing movies

Predefined methods

Puppets

Rectangle and point coordinates

Sound

Sprites

System

Text

Time

Variables

Lingo syntax

The following typographic conventions are used in this section:

`word`	actual Lingo word
word	placeholder for a specific name or parameter
{*word*}	optional items

Cast members

Word	Syntax	Category
`cast`	`the` *property* `of cast` *whichCastmember*	keyword
`buttonStyle`	`the buttonStyle`	property
`castNum of sprite`	`the castNum of sprite` *whichSprite*	sprite property
`the castType of cast`	`the castType of cast` *cast member*	cast property
`checkBoxAccess`	`the checkBoxAccess`	property
`checkBoxType`	`the checkBoxType`	property
`duplicate cast`	`duplicate cast` *original*{`,` *new*}	command
`field`	`field` *whichField*	keyword
`the frameScript`	`the frameScript`	frame property
`name`	`the name of cast` *whichCastmember*	cast property
`number of castMembers`	`the number of cast` *whichCastmember*	cast property
`number`	`the number of castmembers`	property
`picture of cast`	`the picture of cast` *whichCastmember*	cast property
`the purgePriority of cast`	`the purgePriority of cast ¬` *whichCastmember*	cast property

Cast members

Word	Syntax	Category
`scriptNum of sprite`	`scriptNum of sprite` *whichSprite*	sprite property
`text of cast`	`the text of cast` *whichCastmember*	text property
`the`	`the` *property*	keyword
`the width of cast`	`the width of cast` *whichCastmember*	cast property

Cast window management

Word	Syntax	Category
`the backColor of cast`	`set the backColor of cast` *castName* ¬ `to` *colorNumber*	cast property
`duplicate cast`	`duplicate cast` *original{, new}*	command
`erase cast`	`erase cast` *whichCastmember*	command
`the fileName of cast`	`the fileName of cast` *cast member*	cast property
`findEmpty`	`findEmpty(cast` *castNum*`)`	function
`the height of cast`	`the height of cast` *whichCastmember*	cast property
`importFileInto`	`importFileInto` *cast member, fileName*	command
`the loaded of cast`	`the loaded of cast` *whichCastmember*	cast property
`the modified of cast`	`the modified of cast` *castMember*	cast property
`move cast`	`move cast` *whichCastmember*`{, cast` ¬ *whichLocation*`}`	command
`the palette of cast`	`the palette of cast` *whichCastmember*	cast property
`pasteClipBoardInto`	`pasteClipBoardInto cast` *whichCastmember*	command
`the regPoint of cast`	`the regPoint of cast` *whichCastmember*	cast property

Cast window management

Word	Syntax	Category
`the scriptText of cast`	`the scriptText of cast` *whichCastmember*	cast property
`the size of cast`	`the size of cast` *castName*	cast property
`unLoad`	`unLoad`	command
`unLoadCast`	`unLoadCast`	command
`the width of cast`	`the width of cast` *whichCastmember*	cast property

Code structures

Word	Syntax	Category
`--`	`-- [comment ]`	comment delimiter
`¬`	`part of this statement continues ¬` `on the next line`	special character
`down`	`down`	keyword
`exit`	`exit`	keyword
`exit repeat`	`exit repeat`	keyword
`global`	`global` *variable1*{, *variable2* }{, *variable3* }	keyword
`halt`	`halt`	command
`if`	`if` *logicalExpression* `then` *then-statement*	keyword
`next repeat`	`next repeat`	keyword
`nothing`	`nothing`	command
`on`	`on` *handlerName* {*argument1* }{, *argument2* }¬ {, *argument3* }… {*statement* } `end` *handlerName*	keyword
`repeat`	`repeat with` *counter* = *start to finish* {*statements*} `end repeat`	keyword

Code structures

Word	Syntax	Category

`repeat while`	`repeat while` *testCondition*
	  {*statements*}
	`end repeat`
		keyword

`repeat with...down to`	`repeat with` *variable* = *startValue* `down` ¬ `to` *endValue*	keyword
`repeat with...in list`	`repeat with` *variable* `in` *someList*	keyword
`result`	`the result`	function
`return`	`return` *expression*	keyword
`the trace`	`the trace` *trueOrFalse*	property
`the traceLoad`	`the traceLoad`	property
`the traceLogFile`	`the traceLogFile`	property

Constants

Word	Syntax	Category
`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

Digital video

Word	Syntax	Category
`the center of cast`	`the center of cast` *castName*	digital video cast property
`the controller of cast`	`the controller of cast` *castName*	digital video cast property
`the crop of cast`	`the crop of cast`	digital video cast property
`the directToStage of cast`	`the directToStage of cast` *castName*	digital video cast property
`the duration of cast`	`the duration of cast` *castName*	digital video cast property
`the frameRate of cast`	`the frameRate of cast` *DVcast member*	digital video cast property
`the loop of cast`	`the loop of cast` *castName*	digital video cast property
`the movieRate of sprite`	`the movieRate of sprite` *channelNumber*	digital video sprite property

Digital video

Word	Syntax	Category
`the movieTime of sprite`	`the movieTime of sprite` *channelNumber*	digital video sprite property
`the pausedAtStart`	`the pausedAtStart of cast ¬` *whichDVMovie trueOrFalse*	digital video cast property
`the preLoad of cast`	`the preLoad of cast` *castMember*	digital video cast property
`preLoadRAM`	`the preLoadRAM`	property
`the quickTimePresent`	`the quickTimePresent`	function
`the sound of cast`	`the sound of cast` *castMember* `to ¬` *onOrOff*	digital video cast property
`the startTime of sprite`	`the startTime of sprite` *spriteNumber*	digital video sprite property
`the stopTime of sprite`	`the stopTime of sprite` *whichSprite*	digital video sprite property
`the video of cast`	`the video of` *castName*	digital video cast property
`the volume of sprite`	`the volume of sprite` *spriteNum*	digital video sprite property

Event handlers and messages

Word	Syntax	Category
`abort`	`abort`	command
`the actorList`	`the actorList`	property
`alert`	`alert` *message*	command
`on`	`on` *handlerName {argument1}*¬ *{, argument2 }{, argument3 }…* *{statements}* `end` *handlerName*	keyword

`on enterFrame`	`on enterFrame`
	statement(s)
	`end enterFrame`
		event handler

`on exitFrame`	`on exitFrame`
	statement(s)
	`end exitFrame`
		event handler

`on idle`	`on idle`
	statement(s)
	`end idle`
		movie handler

`on keyDown`	`on keyDown`
	statement(s)
	`end keyDown`
		event handler

`on keyUp`	`on keyUp`
	statement(s)
	`end keyUp`
		event handler

`on mouseDown`	`on mouseDown`
	statement(s)
	`end mouseDown`
		event handler

`on mouseUp`	`on mouseUp`
	statement(s)
	`end mouseUp`
		event handler

`on startMovie`	`on startMovie`
	statement(s)
	`end startMovie`
		event handler

`on stepMovie`	`on stepMovie`
	statement(s)
	`end stepMovie`
		event handler

Event handlers and messages

Word	Syntax	Category

`on stopMovie`	`on stopMovie`
	  statement(s)
	`end stopMovie`
		event handler

`param`	`param(`*parameter*`)`	function
`the paramCount`	`the paramCount`	function

Events

Word	Syntax	Category
`alert`	`alert` *message*	command
`dontPassEvent`	`dontPassEvent`	command
`when keyDown`	`when keyDown then` *statement*	command
`the keyUpScript`	`the keyUpScript`	property
`lastEvent`	`the lastEvent`	function
`when mouseDown`	`when mouseDown then` *statement*	command
`when mouseUp`	`when mouseUp then` *statement*	command
`nothing`	`nothing`	command
`pass`	`pass`	command
`perFrameHook`	`the perFrameHook`	property
`when timeOut`	`when timeOut then` *statement*	command

External to Director

Word	Syntax	Category
`closeDA`	`closeDA`	command
`closeResFile`	`closeResFile` {*whichFile*}	command
`closeXlib`	`closeXlib` {*whichFile*}	command

External to Director

Word	Syntax	Category
`copyToClipBoard`	`copyToClipBoard cast` *cast member*	command
`getNthFileNameIn¬ Folder`	`getNthFileNameInFolder(`*folderPath*, ¬ *fileNumber*`)`	function
`importFileInto`	`importFileInto` *cast member*, *fileName*	command
`movieFileFreeSize`	`the movieFileFreeSize`	function
`movieFileSize`	`the movieFileSize`	function
`the movieName`	`the movieName`	function
`the moviePath`	`the moviePath`	function
`open`	`open` {*whichDocument* `with`} ¬ *whichApplication*	command
`openDA`	`openDA` *DAname*	command
`openResFile`	`openResFile` *whichFile*	command
`openXlib`	`openXlib` *whichFile*	command
`pathName`	`the pathName`	function
`saveMovie`	`saveMovie` {*pathname:filename*}	command
`the searchCurrentFolder`	`the searchCurrentFolder`	function
`the searchPath`	`the searchPath`	function
`setCallBack`	`setCallBack` *XCMDname*, *value*	command
`showResFile`	`showResFile` {*whichFile*}	command
`showXlib`	`showXlib` {*Xlibfilename*}	command
`sound playFile`	`sound playFile` *whichChannel*, *whichFile*	command
`updateMovieEnabled`	`the updateMovieEnabled`	property
`xFactoryList`	`xFactoryList (`*whichLibrary*`)`	function

Factories, methods, and XObjects

Word	Syntax	Category
`factory`	`factory` *factoryName* `methods`	keyword
`factory`	`factory(`*factoryName*`)`	function
`instance`	`instance` *variable1{, variable2}{, variable3}…*	keyword
`me`	`me`	keyword
`method`	`method` *methodName {argument1}{, argument2}…*	keyword
`mInstanceRespondsTo`	*XObject*`(mInstanceRespondsTo,` *message*`)`	predefined method

Keyboard

Word	Syntax	Category
`commandDown`	`the commandDown`	function
`controlDown`	`the controlDown`	function
`key`	`the key`	function
`keyCode`	`the keyCode`	function
`keyDownScript`	`the keyDownScript`	property
`keyUp`	`keyUp`	function
`the keyUpScript`	`the keyUpScript`	property
`lastKey`	`the lastKey`	function

`on keyDown`	`on keyDown`
	  statement(s)
	`end keyDown`
		event handler

`on keyUp`	`on keyUp`
	  statement(s)
	`end keyUp`
		event handler

`optionDown`	`the optionDown`	function

Keyboard

Word	Syntax	Category
`shiftDown`	`the shiftDown`	function
`stillDown`	`the stillDown`	function

Lists

Word	Syntax	Category
`[ ]`	`[`*entry1, entry2, entry3, ...*`]`	list operator
`the actorList`	`the actorList`	property
`add`	`add` *linearList, value*	command
`addAt`	`addAt` *list*, *position*, *value*	command
`addProp`	`addProp` *list, property, value*	command
`append`	`append` *list*, *value*	command
`count`	`count(`*list*`)`	function
`deleteAt`	`deleteAt` *list*, *number*	command
`deleteProp`	`deleteProp` *list*, *propert*y	command
`findPos`	`findPos(`*list*, *prop*`)`	function
`findPosNear`	`findPosNear(`*list*, *prop*`)`	function
`getaProp`	`getaProp(`*list, positionOrProperty*`)`	function
`getAt`	`getAt(`*list*, *position*`)`	function
`getLast`	`getLast(`*list*`)`	function
`getOne`	`getOne(`*list, value*`)`	function
`getPos`	`getPos(`*list, value*`)`	function
`getProp`	`getProp(`*list, property)*	function
`getPropAt`	`getPropAt`*(list, index)*	function

Lists

Word	Syntax	Category
`ilk list`	`ilk list`	function
`list`	`list(`*value1*`, `*value2*`, `*value3...*`)`	function
`listP`	`listP(`*item*`)`	function
`max`	`max(`*list*`)`	function
`min`	`min(`*list*`)`	function
`setaProp`	`setaProp` *list, property, newValue*	command
`setAt`	`setAt` *list, orderNumber, value*	command
`setProp`	`setProp` *list, property, newValue*	command
`sort`	`sort` *list*	command

Logical operators and functions

Word	Syntax	Category
and	*logicalExpression1* and *logicalExpression2*	logical operator
`contains`	*stringExpression1* `contains` *stringExpression2*	comparison operator
`integerP`	`integerP(`*expression*`)`	function
`intersect`	`intersect {`*rectangle1*`, `*rectangle2*`}`	function
`not`	`not` *logicalExpression*	logical operator
`objectP`	`objectP(`*expression*`)`	function
`or`	*logicalExpression1* `or` *logicalExpression2*	logical operator
`pictureP`	`pictureP(`*castMember*`)`	function
`soundBusy`	`soundBusy(`*whichChannel*`)`	function
`sprite... intersects`	`sprite` *sprite1* `intersects` *sprite2*	comparison operator
`sprite...within`	`sprite` *sprite1* `within` *sprite2*	comparison operator

Logical operators and functions

Word	Syntax	Category
`starts`	*string1* `starts` *string2*	comparison operator
`stringP`	`stringP(`*expression*`)`	function
`symbolP`	`symbolP(`*expression*`)`	function

Menus

Word	Syntax	Category
`checkMark of menuItem`	`the checkMark of menuItem` *whichItem* ¬ `of menu` *whichMenu*	menu property
`enabled of menuItem`	`the enabled of menuItem` *whichItem* `of` ¬ `menu` *whichMenu*	menu property
`installMenu`	`installMenu` *cast member*	command
`menu:`	`menu:` *menuName* *itemName* ≈ *script*	keyword
`name of menu`	`the name of menu` *whichMenu*	menu property
`name of menuItem`	`the name of menuItem` *whichItem* `of` ¬ `menu` *whichMenu*	menu property
`number of menuItems`	`the number of menuItems of` ¬ `menu` *whichMenu*	menu property
`number of menus`	`the number of menus`	menu property
`script of menuItem`	`the script of menuItem` *whichItem* `of` ¬ `menu` *whichMenu*	menu property

Mouse and pointer

Word	Syntax	Category
`clickOn`	`the clickOn`	function
`cursor`	`cursor [castNumber, maskCastNumber]`	command

Mouse and pointer

Word	Syntax	Category
`cursor`	`cursor` *whichCursor*	command
`cursor of sprite`	`the cursor of sprite` *whichSprite*	sprite property
`doubleClick`	`the doubleClick`	function
`lastClick`	`the lastClick`	function
`lastRoll`	`the lastRoll`	function
`mouseCast`	`the mouseCast`	function
`the clickLoc`	`the clickLoc`	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(`*whichSprite*`)`	function
`stillDown`	`the stillDown`	function

Movie in a window

Word	Syntax	Category
`close window`	`close window` *windowIdentifier*	command
`the drawRect of window`	`the drawRect of window` *windowName*	window property

Movie in a window

Word	Syntax	Category
`the fileName of window`	`the fileName of window` *whichWindow*	window property
`forget window`	`forget window` *whichWindow*	command
`inflate rect`	`inflate(rectangle,` *widthChange*`, ¬` *heightChange*`)`	function
`the modal of window`	`the modal of window "`*window*`"`	window property
`moveToBack`	`moveToBack window "`*whichWindow*`"`	command
`moveToFront`	`moveToFront window "`*whichWindow*`"`	command
`open window`	`open window "`*whichWindow*`"`	command
`the rect of window`	`the rect of window` *whichWindow*	window property
`tell`	`tell` *object* `to` *statement; object* `statement(s)` `end tell`	command
`the title of window`	`the title of window` *whichWindow*	window property
`the titleVisible of window`	`the titleVisible of window` *whichWindow*	window property
`the visible of window`	`the visible of window` *whichWindow*	window property
`window`	`window` *whichWindow*	keyword
`the windowList`	`the windowList`	property
`the windowType of window`	`the windowType of window` *whichWindow*	window property

Operators and math functions

Word	Syntax	Category
`&`	*expression1* `&` *expression2*	text operator
`&&`	*expression1* `&&` *expression2*	text operator
`( )`	`(`*expression*`)`	grouping operator
`*`	*expression1* `*` *expression2*	arithmetic operator
`+`	*expression1* `+` *expression2*	arithmetic operator
`-`	*expression1* `-` *expression2*	arithmetic operator
`-`	`-`*expression*	arithmetic operator
`/`	*expression1* `/` *expression2*	arithmetic operator
`<`	*expression1* `<` *expression2*	comparison operator
`<=`	*expression1* `<=` *expression2*	comparison operator
`<>`	*expression1* `<>` *expression2*	comparison operator
`=`	*expression1* `=` *expression2*	comparison operator
`>`	*expression1* `>` *expression2*	comparison operator
`>=`	*expression1* `>=` *expression2*	comparison operator
`#`	`#`*symbolName*	definition operator
`abs`	`abs(`*numericExpression*`)`	function
`atan`	`atan(`*number*`)`	function
`cos`	`cos(`*angle*`)`	function
`exp`	`exp(`*integer*`)`	function
`integer`	`integer(`*numericExpression*`)`	function
`log`	`log(`*number*`)`	function
`maxInteger`	`the maxInteger`	function
`mod`	*integerExpression1* `mod` *integerExpression2*	arithmetic operator
`pi`	`pi()`	function

Operators and math functions

Word	Syntax	Category
`power`	`power(`*base, exponent*`)`	function
`random`	`random(`*integerExpression*`)`	function
`the randomSeed`	`the randomSeed`	property
`sin`	`sin(`*angle*`)`	function
`sqrt`	`sqrt(`*numericExpression*`)`	function
`sqrt`	`the sqrt(`*number*`)`	function
`tan`	`tan(`*angle*`)`	function
`value`	`value(`*string*`)`	function

Output

Word	Syntax	Category
`perFrameHook`	`the perFrameHook`	property
`printFrom`	`printFrom` *fromFrame*{ , *toFrame*}¬ { , *reduction*}	command

Parent scripts

Word	Syntax	Category
`ancestor`	`property ancestor`	property
`birth`	`birth(script` *parentScriptName, value1,* ¬ *value2, ...*`)`	function
`property`	`property` {*property1*}{, *property2*} ¬ {, *property3*} {...}	keyword

Playing movies

Word	Syntax	Category
`continue`	`continue`	command
`frame`	`the frame`	function
`the frameLabel`	`the frameLabel`	frame property
`the frameScript`	`the frameScript`	frame property
`go`	`go {to} {frame}` *whichFrame*	command
`go`	`go {to} movie` *whichMovie*	command
`go`	`go {to} {frame}` *whichFrame* `of ¬` `movie` *whichmovie*	command
`go loop`	`go loop`	command
`go next`	`go next`	command
`go previous`	`go previous`	command
`the purgePriority of cast`	`the purgePriority of ¬` `cast` *whichCastmember*	cast property
`halt`	`halt`	command
`label`	`label(`*expression*`)`	function
`labelList`	`the labelList`	function
`the lastFrame`	`the lastFrame`	property
`loop`	`loop`	keyword
`marker`	`marker(`*integerExpression*`)`	function
`movie`	`the movie`	function
`movieFileFreeSize`	`the movieFileFreeSize`	function
`movieFileSize`	`the movieFileSize`	function
`the movieName`	`the movieName`	function
`the moviePath`	`the moviePath`	function
`next`	`next`	keyword

Playing movies

Word	Syntax	Category
`pathName`	`the pathName`	function
`pause`	`pause`	command
`pauseState`	`the pauseState`	function
`play`	`play {frame}` *whichFrame*	command
`play`	`play movie` *whichMovie*	command
`play`	`play {frame}` *whichFrame* `of movie ¬` *whichMovie*	command
`play done`	`play done`	command
`preLoad`	`preLoad` *fromFrame*`,` *toFrameNum*	command
`preLoadCast`	`preLoadCast` *fromCastNumber*`,` *toCastNumber*	command
`quit`	`quit`	command
`ramNeeded`	`ramNeeded(`*firstFrame, lastFrame*`)`	function
`saveMovie`	`saveMovie` {*pathname:filename*}	command
`switchColorDepth`	`the switchColorDepth`	property
`updateMovieEnabled`	`the updateMovieEnabled`	property

Predefined methods and special messages

Word	Syntax	Category
`mAtFrame`	`method mAtFrame frameNumber, ¬` `subFrameNumber` {*statements*} `end mAtFrame`	special message
`mDescribe`	*XObjectName*`(mDescribe)`	predefined method
`mDispose`	*object* `mDispose`	predefined method
`mGet`	*object*`(mGet,` *whichElement*`)`	predefined method

Predefined methods and special messages

Word	Syntax	Category
`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,` *whichElement*`,` *expression*`)`	predefined method
`mRespondsTo`	*XObjectInstance*`(mRespondsTo,` *message*`)`	predefined method

Puppets

Word	Syntax	Category
`the framePalette`	`the framePalette`	frame property
`puppet of sprite`	`the puppet of sprite` *whichSprite*	sprite property
`puppetPalette`	`puppetPalette` *whichPalette*¬ `{,` *speed*`}{,` *nFrames*`}`	command
`puppetSound`	`puppetSound` *whichCastmember*	command
`puppetSound`	`puppetSound 0`	command
`puppetSprite`	`puppetSprite` *whichSprite*`,` *state*	command
`puppetTempo`	`puppetTempo` *framesPerSecond*	command
`puppetTransition`	`puppetTransition` *whichTransition*¬ `{,` *time*`}{,` *chunkSize*`}`¬ `{,` *changeArea*`}`	command
`updateStage`	`updateStage`	command
`xFactoryList`	`xFactoryList(`*whichLibrary*`)`	function

Rectangle and point coordinates

Word	Syntax	Category
`ilk point`	`ilk point`	function
`ilk rect`	`ilk rect`	function
`inflate rect`	`inflate(rectangle,` *widthChange*, ¬ *heightChange*`)`	function
`inside`	`inside(`*point, rectangle*`)`	function
`intersect`	`intersect(`*rectangle1*, *rectangle2*`)`	function
`map`	`map(`*targetRect, sourceRect, destination Rect*`)`	function
`offset rect`	`offset(`*rectangle*, *horizontalChange*, ¬ *verticalChange*`)`	function
`point`	`point(`*horizontal*, *vertical*`)`	function
`rect`	`rect(`*left, top, right, bottom*`)`	function
`the rect of cast`	`the rect of cast` *whichCastmember*	cast property
`the rect of window`	`the rect of window` *whichWindow*	window property
`the sourceRect of window`	`the sourceRect of window` *whichWindow*	window property
`union rect`	`union rect` *rect1, rect2*	function

Sound

Word	Syntax	Category
`puppetSound`	`puppetSound` *whichCastmember*	command
`puppetSound`	`puppetSound 0`	command
`sound close`	`sound close` *soundChannel*	command
`sound fadeIn`	`sound fadeIn` *whichChannel*	command
`sound fadeIn`	`sound fadeIn` *whichChannel*, *ticks*	command
`sound fadeOut`	`sound fadeOut` *whichChannel*	

Sound

Word	Syntax	Category
sound fadeOut	sound fadeOut *whichChannel*, *ticks*	command
the sound of cast	the sound of cast *castMember* to ¬ *onOrOff*	digital video cast property
sound playFile	sound playFile *whichChannel*, *whichFile*	command
sound stop	sound stop *whichChannel*	command
soundBusy	soundBusy(*whichChannel*)	function
soundEnabled	the soundEnabled	property
soundLevel	the soundLevel	property
the volume of sprite	the volume of sprite *spriteNum*	digital video sprite property

Sprites

Word	Syntax	Category
backColor	the backColor of sprite *whichSprite*	sprite property
bottom	the bottom of sprite *whichSprite*	sprite property
the blend of sprite	the blend of sprite	sprite property
castNum of sprite	the castNum of sprite *whichSprite*	sprite property
constrainH	constrainH(*whichSprite, integerExp*)	function
constraint of sprite	the constraint of sprite *whichSprite*	sprite property
constrainV	constrainV(*whichSprite, integerExp*)	function
the editableText of sprite	the editableText of sprite *whichSprite*	sprite property
foreColor of sprite	the foreColor of sprite *whichSprite*	sprite property
the framePalette	the framePalette	frame property

Sprites

Word	Syntax	Category
`height of sprite`	`the height of sprite` *whichSprite*	sprite property
`hilite`	`the hilite of cast` *whichCastmember*	button property
`ink of sprite`	`the ink of sprite` *whichSprite*	sprite property
`left of sprite`	`the left of sprite` *whichSprite*	sprite property
`lineSize of sprite`	`the lineSize of sprite` *whichSprite*	sprite property
`locH of sprite`	`the locH of sprite` *whichSprite*	sprite property
`locV of sprite`	`the locV of sprite` *whichSprite*	sprite property
`moveableSprite of sprite`	`moveableSprite of sprite` *whichSprite*	command
`the right of sprite`	`the right of sprite` *whichSprite*	sprite property
`the scoreColor of sprite`	`the scoreColor of sprite` *whichSprite*	sprite property
`scriptNum of sprite`	`scriptNum of sprite` *whichSprite*	sprite property
`spriteBox`	`spriteBox` *whichSprite, left, top, right, bottom*	command
`the stretch of sprite`	`the stretch of sprite` *whichSprite*	sprite property
`top of sprite`	`the top of sprite` *whichSprite*	sprite property
`type of sprite`	`the type of sprite` *whichSprite*	sprite property
`updateStage`	`updateStage`	command
`the trails of sprite`	`the trails of sprite` *whichSprite*	sprite property
`the visible of sprite`	`the visible of sprite` *whichSprite*	sprite property
`width of sprite`	`the width of sprite` *whichSprite*	sprite property
`zoomBox`	`zoomBox` *startSprite, endSprite{, delayTicks}*	command

System

Word	Syntax	Category
beep	beep [*numberOfTimes*]	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
the itemDelimiter	the itemDelimiter	property
the loaded of cast	the loaded of cast *whichCastmember*	cast property
machineType	the machineType	function
maxInteger	the maxInteger	function
mci	mci "string"	command
memorySize	the memorySize	function
multiSound	the multiSound	property
the preLoadEventAbort	the preLoadEventAbort	property
the purgePriority of cast	the purgePriority of ¬ cast *whichCastmember*	cast property
the quickTimePresent	the quickTimePresent	function
quit	quit	command
ramNeeded	ramNeeded (*firstFrame, lastFrame*)	function
restart	restart	command
the romanLingo	the romanLingo	property
shutDown	shutDown	command
stage	the stage	system property

System

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
`the`	`the` *property*	keyword
`version`	`version`	system variable

Text

Word	Syntax	Category
`alert`	`alert` *message*	command
`char...of`	`char` *whichCharacter* `of` *chunkExpression*	
`char...to`	`char` *firstCharacter* `to ¬` *lastCharacter* `of` *chunkExpression*	keyword
`chars`	`chars(`*stringExpression*`, ¬` *firstCharacter*`,` *lastCharacter*`)`	function
`charToNum`	`charToNum(`*stringExpression*`)`	function
`contains`	*stringExpression1* `contains` *stringExpression2*	comparison operator
`delete`	`delete` *chunkExpression*	command
`do`	`do` *stringExpression*	command
`editableText of sprite`	`editableText of sprite` *whichSprite*	sprite property
`field`	`field` *whichField*	keyword

Text

Word	Syntax	Category
`the foreColor of cast`	`set the foreColor of cast` *castName* ¬ `to` *colorNumber*	cast property
`hilite`	`hilite` *chunkExpression*	command
`item...of`	`item` *whichItem* `of` *chunkExpression*	keyword
`item...to`	`item` *firstItem* `to` *lastItem* `of` *chunkExpression*	keyword
`the itemDelimiter`	`the itemDelimiter`	property
`the last`	`the last` *chunk* `in (`*chunkExpression*`)`	function
`length`	`length(`*string*`)`	function
`line...of`	`line` *whichLine* `of` *chunkExpression*	keyword
`line...to`	`line` *firstLine* `to` *lastLine* `of` *chunkExpression*	keyword
`number of chars`	`the number of chars in` *chunkExpression*	chunk function
`number of items`	`the number of items in` *chunkExpression*	chunk function
`number of lines`	`the number of lines in` *chunkExpression*	chunk function
`number of words`	`the number of words in` *chunkExpression*	chunk function
`numToChar`	`numToChar(`*integerExpression*`)`	function
`offset`	`offset(`*stringExpression1*`,` *stringExpression2*`)`	function
`put...after`	`put expression after` *chunkExpression*	command
`put...before`	`put` *expression* `before` *chunkExpression*	command
`put...into`	`put` *expression* `into` *chunkExpression*	command
`selection`	`the selection`	function
`selEnd`	`the selEnd`	text property
`selStart`	`the selStart`	text property
`starts`	*string1* `starts` *string2*	comparison operator
`string`	`string(`*expression*`)`	function

Text

Word	Syntax	Category
`text`	`the text of cast` *whichCastmember*	text property
`textAlign`	`the textAlign of field` *whichField*	text property
`textFont`	`the textFont of field` *whichField*	text property
`textHeight`	`the textHeight of field` *whichField*	text property
`textSize`	`the textSize of field` *whichField*	text property
`textStyle`	`the textStyle of field` *whichField*	text property
`value`	`value(`*string*`)`	function
`word...of`	`word` *whichWord* `of` *chunkExpression*	keyword
`word...to`	`word` *firstWord* `to` *lastWord* `of` ¬ *chunkExpression*	keyword

Time

Word	Syntax	Category
`date`	`the abbr date`	function
`date`	`the abbrev date`	function
`date`	`the abbreviated date`	function
`date`	`the date`	function
`date`	`the long date`	function
`date`	`the short date`	function
`delay`	`delay` *numberOfTicks*	command
`framesToHMS`	`framesToHMS(`*frames*`,` *tempo*`,` ¬ *dropFrame*`,` *fractionalSeconds*`)`	function
`HMStoFrames`	`HMStoFrames(`*frames*`,` *tempo*`,` ¬ *dropFrame*`,` *fractionalSeconds*`)`	function
`startTimer`	`startTimer`	command

Time

Word	Syntax	Category
`ticks`	`the ticks`	function
`time`	`the abbr time`	function
`time`	`the abbrev time`	function
`time`	`the abbreviated time`	function
`time`	`the long time`	function
`time`	`the short time`	function
`time`	`the time`	function
`when timeOut`	`when timeOut then` *statement*	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
`clearGlobals`	`clearGlobals`	command
`listP`	`listP(`*item*`)`	function
`param`	`param(`*parameter*`)`	function
`the paramCount`	`the paramCount`	function
`picture of cast`	`the picture of cast` *whichCastmember*	cast property
`property`	`property` {*property1*}{, *property2*}¬ {, *property3*}{...}	keyword
`put`	`put` *expression*	command
`set...=`	`set` *variable* `=` *expression*	command
`set...=`	`set the` *property* `=` *expression*	command
`set...to`	`set` *variable* `to` *expression*	command
`set...to`	`set the` *property* `to` *expression*	command
`showGlobals`	`showGlobals`	command
`showLocals`	`showLocals`	command
`voidP`	`voidP(`*variableName*`)`	function

Octal to Decimal Converter

In Director 4, references to octal cast IDs such as A11 are being phased out. Lingo will only accept octal references in Director 3.1.3 or earlier movies when Allow Outdated Lingo is checked in the Movie Info dialog box.

Octal (A11) is still an option in the Cast Window Options dialog box to allow you to convert your cast references.

For a quick octal to decimal conversion, use the Lingo command:

```
put the number of cast x
```

in the message window, where x is the octal ID.

It's always a good idea to refer to cast members in Lingo by cast name rather than cast number so that if cast positions change, the reference is maintained.

A11	1	A47	31	A85	61	B43	91	B81	121	C37	151
A12	2	A48	32	A86	62	B44	92	B82	122	C38	152
A13	3	A51	33	A87	63	B45	93	B83	123	C41	153
A14	4	A52	34	A88	64	B46	94	B84	124	C42	154
A15	5	A53	35	B11	65	B47	95	B85	125	C43	155
A16	6	A54	36	B12	66	B48	96	B86	126	C44	156
A17	7	A55	37	B13	67	B51	97	B87	127	C45	157
A18	8	A56	38	B14	68	B52	98	B88	128	C46	158
A21	9	A57	39	B15	69	B53	99	C11	129	C47	159
A22	10	A58	40	B16	70	B54	100	C12	130	C48	160
A23	11	A61	41	B17	71	B55	101	C13	131	C51	161
A24	12	A62	42	B18	72	B56	102	C14	132	C52	162
A25	13	A63	43	B21	73	B57	103	C15	133	C53	163
A26	14	A64	44	B22	74	B58	104	C16	134	C54	164
A27	15	A65	45	B23	75	B61	105	C17	135	C55	165
A28	16	A66	46	B24	76	B62	106	C18	136	C56	166
A31	17	A67	47	B25	77	B63	107	C21	137	C57	167
A32	18	A68	48	B26	78	B64	108	C22	138	C58	168
A33	19	A71	49	B27	79	B65	109	C23	139	C61	169
A34	20	A72	50	B28	80	B66	110	C24	140	C62	170
A35	21	A73	51	B31	81	B67	111	C25	141	C63	171
A36	22	A74	52	B32	82	B68	112	C26	142	C64	172
A37	23	A75	53	B33	83	B71	113	C27	143	C65	173
A38	24	A76	54	B34	84	B72	114	C28	144	C66	174
A41	25	A77	55	B35	85	B73	115	C31	145	C67	175
A42	26	A78	56	B36	86	B74	116	C32	146	C68	176
A43	27	A81	57	B37	87	B75	117	C33	147	C71	177
A44	28	A82	58	B38	88	B76	118	C34	148	C72	178
A45	29	A83	59	B41	89	B77	119	C35	149	C73	179
A46	30	A84	60	B42	90	B78	120	C36	150	C74	180

C75	181	D33	211	D71	241	E27	271	E65	301	F23	331
C76	182	D34	212	D72	242	E28	272	E66	302	F24	332
C77	183	D35	213	D73	243	E31	273	E67	303	F25	333
C78	184	D36	214	D74	244	E32	274	E68	304	F26	334
C81	185	D37	215	D75	245	E33	275	E71	305	F27	335
C82	186	D38	216	D76	246	E34	276	E72	306	F28	336
C83	187	D41	217	D77	247	E35	277	E73	307	F31	337
C84	188	D42	218	D78	248	E36	278	E74	308	F32	338
C85	189	D43	219	D81	249	E37	279	E75	309	F33	339
C86	190	D44	220	D82	250	E38	280	E76	310	F34	340
C87	191	D45	221	D83	251	E41	281	E77	311	F35	341
C88	192	D46	222	D84	252	E42	282	E78	312	F36	342
D11	193	D47	223	D85	253	E43	283	E81	313	F37	343
D12	194	D48	224	D86	254	E44	284	E82	314	F38	344
D13	195	D51	225	D87	255	E45	285	E83	315	F41	345
D14	196	D52	226	D88	256	E46	286	E84	316	F42	346
D15	197	D53	227	E11	257	E47	287	E85	317	F43	347
D16	198	D54	228	E12	258	E48	288	E86	318	F44	348
D17	199	D55	229	E13	259	E51	289	E87	319	F45	349
D18	200	D56	230	E14	260	E52	290	E88	320	F46	350
D21	201	D57	231	E15	261	E53	291	F11	321	F47	351
D22	202	D58	232	E16	262	E54	292	F12	322	F48	352
D23	203	D61	233	E17	263	E55	293	F13	323	F51	353
D24	204	D62	234	E18	264	E56	294	F14	324	F52	354
D25	205	D63	235	E21	265	E57	295	F15	325	F53	355
D26	206	D64	236	E22	266	E58	296	F16	326	F54	356
D27	207	D65	237	E23	267	E61	297	F17	327	F55	357
D28	208	D66	238	E24	268	E62	298	F18	328	F56	358
D31	209	D67	239	E25	269	E63	299	F21	329	F57	359
D32	210	D68	240	E26	270	E64	300	F22	330	F58	360

F61	361	G17	391	G55	421	H13	451	H51	481	H87	511
F62	362	G18	392	G56	422	H14	452	H52	482	H88	512
F63	363	G21	393	G57	423	H15	453	H53	483		
F64	364	G22	394	G58	424	H16	454	H54	484		
F65	365	G23	395	G61	425	H17	455	H55	485		
F66	366	G24	396	G62	426	H18	456	H56	486		
F67	367	G25	397	G63	427	H21	457	H57	487		
F68	368	G26	398	G64	428	H22	458	H58	488		
F71	369	G27	399	G65	429	H23	459	H61	489		
F72	370	G28	400	G66	430	H24	460	H62	490		
F73	371	G31	401	G67	431	H25	461	H63	491		
F74	372	G32	402	G68	432	H26	462	H64	492		
F75	373	G33	403	G71	433	H27	463	H65	493		
F76	374	G34	404	G72	434	H28	464	H66	494		
F77	375	G35	405	G73	435	H31	465	H67	495		
F78	376	G36	406	G74	436	H32	466	H68	496		
F81	377	G37	407	G75	437	H33	467	H71	497		
F82	378	G38	408	G76	438	H34	468	H72	498		
F83	379	G41	409	G77	439	H35	469	H73	499		
F84	380	G42	410	G78	440	H36	470	H74	500		
F85	381	G43	411	G81	441	H37	471	H75	501		
F86	382	G44	412	G82	442	H38	472	H76	502		
F87	383	G45	413	G83	443	H41	473	H77	503		
F88	384	G46	414	G84	444	H42	474	H78	504		
G11	385	G47	415	G85	445	H43	475	H81	505		
G12	386	G48	416	G86	446	H44	476	H82	506		
G13	387	G51	417	G87	447	H45	477	H83	507		
G14	388	G52	418	G88	448	H46	478	H84	508		
G15	389	G53	419	H11	449	H47	479	H85	509		
G16	390	G54	420	H12	450	H48	480	H86	510		

Index

Acknowledgements

Programming by John Thompson.

Project management by Bill Edwards.

Product management by David How.

Dictionary compiled by Joe Schmitz and Lee Allis.

Edited by Toni Haskell and Mary Ann Walsh.

Production by Lee Allis.

Special thanks to Mark Castle, John Dowdell, David Shields, and John Thompson.