Topic Index

abs

Forms

```
abs(number)
[the] abs of number
```

Examples

```
abs(-4) -- returns 4
the abs of (y1 - y2)
```

Description

Returns the absolute value of a *number*.

add

Forms

```
numberA + numberB
add numberA to numberB
```

Examples

```
5 + 2 -- yeilds 7
add 9 to 10 -- yeilds 19
```

Description

Returns the addition of *numberA* and *numberB*.

Comments

Where performance is important, use the first form. The second form (the `add` message) is significantly slower as the subtract message must traverse CinsImp's message hierarchy. The short form is executed directly.

Related

divide (+), **subtract (-)**, **multiply (*)**, **div**, **mod**

AllowStyledText

Forms

```
[the] allowStyledText of field
```

Examples

```
allowStyledText of field "Description"
```

Description

Yields `true` if the allowStyledText property of a field is on, `false` otherwise. When allowStyledText is on, the user can apply different fonts, sizes and styles to different passages of text, or individual words. When it is off, the default text font, size and style of the field is enforced for all the field's content.

Can be modified with **set**.

Related

textStyle, textFont, textSize

and

Forms

```
*boolean1* and *boolean2*
```

Examples

```
x and y
true and true -- yeilds true
true and false -- yeilds false
(x = 2) and (5 < y)
the result and it
```

Description

Returns true only if both *boolean1* and *boolean2* are true. Otherwise, it returns false.

Related

or, not, if

answer

Forms

```
answer prompt [with option1 [or option2 [or option3]]]
```

Examples

```
answer "Welcome to CinsImp!"
answer "What kind of recipe would you like?" ¬
           with "Snack" or "Dessert"
answer "The timer has been set." with "Continue"
```

Description

Presents an alert box to the user, containing the *prompt* text (figure X). May provide up to three different option buttons for user response.

The name of the button clicked is placed into the *it* variable.

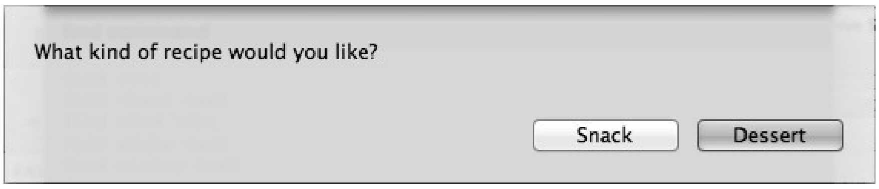
What kind of recipe would you like?

Figure 1: Example of a CinsImp 'answer' alert box

Comments

The default button is always the first. Pressing the Return / Enter key is the same as clicking the default button.

CinsImp will automatically grow the alert box height and button widths to accommodate long *prompt* text and *option* names.

Related

ask, **answer file**, **answer folder**

answer file

Forms

```
answer file prompt [of type type1 [or type2 [or type3]]]
```

Examples

```
answer file "Select a recipe to import:"
answer file "Select a picture for this recipe:" ¬
           of type "picture"
```

Description

Displays the Mac standard file open dialog, which allows the user to select a file on their computer (figure X).

Comments

The full path name of the file is placed in the *it* variable. Or if no file was selected, or the dialog was cancelled, *it* will be empty.

You can specify up to three file *type*s. The default is to allow selection of any type of file. The following special file types can be used:

- "any" – (the default) allows selecting any type of file
- "picture" – allows selecting PNG, JPEG, GIF and TIFF format picture files
- "text" – allows selecting plain text files

Alternatively, you can use the specific file extension for the *type* of file you want to permit, for example, "html", to only allow webpages.

Related

ask file, **answer folder**

answer folder

Forms

```
answer folder prompt
```

Examples

```
answer folder “Select a folder containing recipes to import:”
```

Description

Displays the Mac standard file open dialog, which allows the user to select a folder on their computer (figure X).

Comments

The full path name of the folder is placed in the *it* variable. Or if no file was selected, or the dialog was cancelled, *it* will be empty.

Related

answer file, answer folder

Any

Examples

```
any item
any card of this stack
any word
```

Description

any is a special ordinal that allows you to choose a particular thing (card, word, line, or other object) at random.

ask

Forms

```
ask [password] prompt [with text]
```

Examples

```
ask “How many serves?”
ask “Name the duplicate recipe:” with existingRecipeName
ask password “Please enter the password to access ” ¬
           “this secret recipe:”
ask “What is your favorite mexican dish?” with “Nachos”
```

Description

Presents an alert box to solicit information from the user (figure X). The *prompt* is displayed as the message of the alert.

Places the text typed by the user into the *it* variable. Of if the user Cancels the alert, the *it* variable will be empty.

Comments

You can optionally provide a default *text* for the response.

CinsImp will automatically grow the alert box height to accommodate a longer *prompt* and/or *text*.

If you specify the `password` option, the text will be masked out with bullets (·) and the user will not be able to use the *Edit* > *Copy* command on the field.

Related

answer

ask file

Forms

```
ask file prompt [with defaultName]
```

Examples

```
ask file "Export this recipe as:" with field "RecipeTitle"
ask file "Save shopping list:"
```

Description

Displays the Mac standard save dialog, which allows the user to set a name and location for a file on their computer (figure X).

Comments

The full path name of the file is placed in the *it* variable. Or if the dialog was cancelled, *it* will be empty.

Related

answer file

atan

Forms

```
atan(number)
[the] atan of number
```

Examples

```
atan(tan(1)) -- returns 1
```

Description

Returns the arc tangent of a *number* in radians.

Related

tan, **cos**, **sin**

average

Forms

```
average(item1, item2, …, itemN)
[the] average of items
```

Examples

```
average of “21, 51, 4, -7, 23” -- returns 18.4
average(1,8,9,3,3,2,5) -- returns 4.28571
```

Description

Returns the aggregate average (mean) of a list of *items* – equivalent to adding all the items and dividing by the total number of items.

Related

min, **max**, **sum**

Border

Forms

```
[the] border of field
```

Examples

```
the border of field “Instructions”
```

Description

Yields the border style of the text *field*. Can be modified with **set**.

This property is only available for text fields. It is an error on other element types.

Comments

Supported text field border styles include:

Text Field Border Styles	
`transparent`	The field background is see-thru.
`opaque`	Draws a white background over the card or background picture beneath.
`rectangle`	Draws a white background and a rectangular frame around the outside of the field.
`scrolling`	All the attributes of rectangle with the addition of a vertical scrollbar to allow scrolling multiple lines of text.

Related

Style

CantDelete

Forms

```
[the] cantDelete of element
```

Examples

```
cantDelete of this card
the cantDelete of background "Preferences"
```

Description

Yields `true` if the cantDelete property of the *element* (card or background) is on, otherwise `false`. Can be modified with **set**.

CinsImp will not allow the user or script to delete a card for which this property is on, nor the last card of such a protected background.

Related

Locked

CantModify

Forms

```
[the] cantModify [of this stack]
```

Description

Yields the setting of the stack's Can't Modify option, `true` if it is on and the stack can't be modified, or `false` otherwise. Can usually be changed using **set** (see Comments below).

Comments

If the stack is on a locked volume, disk or network share, or has been locked in the Finder, CantModify will always yield true. Using **set** to try and change the CantModify to false in such a stack will have no effect.

Related

CantPeek, locked

CantPeek

Forms

```
[the] cantPeek [of this stack]
```

Description

Yields the setting of the stack's Can't Peek option, `true` if it is on, `false` otherwise. Can be modified using **set**.

If the Cant Peek option is turned on, users will be unable to 'peek' at button and field outlines with the Command-Option, and Command-Option-Shift key combinations. This also prevents one-click opening of the script editor for buttons and fields.

Related

CantModify

character

Forms

```
char[acter] startOffset [to endOffset] of container
the ordinal char[acter] of container
```

Examples

```
the first character of theText
char 4 to 6 of "Banana Split" -- yields "ana"
any character of field "letters"
```

Description

Yields the specified range of characters from *startOffset* to *endOffset* of the *container*. The character range of the container can also be easily modified using **put**.

The second form allows the use of an *ordinal*, such as first, second or last (see X for a complete list of ordinals).

Related

word, line, item, put

Close

Forms

```
close file path-name
```

Examples

```
close file "RecipeExport.txt"
close file "/Users/josh/Documents/RecipeExport.txt"
```

Description

Closes a file *path-name* that was previously opened with open. You should always close a file when you're finished with it, otherwise other applications may not be able to work with that file. If the file is located on a removable device, such as a USB stick, the device cannot be removed until the file is closed.

Comments

CinsImp automatically closes all files opened by a stack when the stack itself is closed. But you should still ensure that you close files yourself when you're finished with them.

Related

Open

CloseBackground

Description

This system message is sent to the current card immediately prior to leaving the card and background. CinsImp will send the message regardless of what triggers the transition to another background. The message is also sent when a stack is being closed.

Parameters

None.

Related

CloseCard, CloseStack, OpenBackground

CloseCard

Description

This system message is sent to the current card immediately prior to leaving the card. CinsImp will send the message regardless of what triggers the transition to another card. The message is also sent when a stack is being closed.

Parameters

None.

Related

CloseBackground, CloseStack, OpenCard

CloseStack

Description

This system message is sent to the current card immediately prior to closing the stack.

The sequence of events when closing a stack is as follows:

- CloseCard
- CloseBackground
- CloseStack

Parameters

None.

Related

CloseCard, CloseBackground, OpenStack

Command Handlers

Form

```
on message [param1, param2, ...]
  statements
  ...
end message
```

Description

Command handlers allow an object (such as a button or card) to respond to a command *message* or system event *message*. When CinsImp runs a command handler, it runs each of the statements that appear between the `on` and `end` keywords, in sequence.

For example, when you create a button in CinsImp, the button script is automatically filled with a handler for the **mouseUp** system event for you:

```
on mouseUp
  your-statements-go-here
end mouseUp
```

When CinsTalk sends a message to an object, it looks through the object's script for a matching command handler. If one is found, the handler begins to run. (You can find out more about CinsTalk messages and the message hierarchy on page X.)

You can define your own commands by creating a command handler with the name of the command you want to define. In this example, a button invokes the custom `displayAGreeting` command:

```
on mouseUp
  displayAGreeting
end mouseUp

on displayAGreeting
  answer "Hello there."
end displayAGreeting
```

The sequence of CinsTalk command handlers within a script is not important.

Command handlers can take any number of parameters, or even no parameters. Parameters are passed to the command when the command is invoked (see **Command Invocation**). In this example, the displayAGreeting command we defined above has been modified to accept the greeting text as a parameter:

```
on mouseUp
  displayAGreeting "Hello there."
end mouseUp

on displayAGreeting theGreeting
  answer theGreeting
end displayAGreeting
```

It is functionally equivalent to the prior example. Both examples will result in an answer dialog displaying the message "Hello there." (see Figure X).

Related

Function Handlers, **Command Invocation, Exit, Next, Pass**

Command Invocation

Form

```
message [param1, param2, ...]
```

Description

Command Invocation is the technique used to run a command, often one you've defined yourself by creating a custom **Command Handler**. The following example invokes the CinsTalk **beep** command:

```
beep
```

You can also pass a parameter to the command, in this case, the number of times CinsImp should beep:

```
beep 3
```

When you define your own commands by creating a custom **Command Handler**, you can invoke them in a similar way:

```
on mouseUp
  greetTheUser "Hello user."
end mouseUp

on greetTheUser theGreeting
  answer theGreeting
end greetTheUser
```

This example invokes the `greetTheUser` command and passes it a single parameter, `theGreeting`, which is then displayed to the user with the answer command (also another CinsTalk command invocation, see Figure X).

Related

Command Handlers, **Function Invocation**

Concatenate

Forms

```
string1 & string2
string1 && string2
```

Examples

```
"The answer is:" && the result
card field "answer" & " is the answer"
```

```
"Hello" & "world" -- yeilds "Helloworld"
"Hello" && "world" -- yeilds "Hello world"
```

Description

Returns the concatenation of two strings. The second form, &&, also inserts a space between *string1* and *string2*.

Related

newline, **quote**

convert

Forms

```
convert expression to {short date | abbr[ev[iated]] date |
           long date | short time | long time}
```

Examples

```
convert "19 September 2013" to short date
           -- it is now "19/09/2013"
convert "19 September 2013" to dateItems
           -- it is now "2013,9,19,12,0,0,5"
convert theCreationDate to the short time
convert "2013,09,19,16,29,31,5" to long date
           -- it is now "Thursday, 19 September 2013"
convert 946753200 to short date
           -- it is now "19/09/2013"
convert "2013,09,19,16,29,31,5" to short time
           -- it is now "4:29 PM"
convert field "creationDate" to timestamp
```

Description

Provides a flexible means to convert date and time information to and from various formats, including display strings, and CinsImp **dateItem**s and **timeStamp**s.

I f *expression* is a container (a variable or field), the value of that container will be overwritten with the new value. Otherwise, the new value will be placed into the *it* variable.

Comments

Convert uses the Gregorian calendar to parse text representations of the date or time.

When converting **dateItems** to another format, it automatically corrects the output. So for example, to find tomorrow's date on the 31st of August, you could write:

```
convert "2013,08,32,0,0,0,0" to dateItems
```

CinsTalk will automatically realise that 32 is not a valid day of the month, and roll the date forward to the equivalent date – the 1st of September.

Related

date, dateItems, weekDay, month, time, timeStamp

cos

Forms

```
cos(number)
[the] cos of number
```

Examples

Description

Returns the cosine of a *number* in radians.

Related

atan, sin, tan

date

Forms

```
the [short | abbr[[ev]iated] | long] date
date()
```

Examples

```
date()
the long date
```

Description

Returns the current system clock date, in short (the default), abbreviated or long format.

Comments

The precise output of each of the formats of date is configurable in the Mac system preferences.

On Australian English systems, the date formats will usually produce outputs similar to these:

Formats	
`the long date`	`Thursday, 19 September 2013`
The longest format, specifying day, day of month, month name and year.	
`the abbr[ev[iated]] date`	`19 September 2013`
An abbreviated format, specifying the day of month, month name and year.	

`the short date`
`date()`
	`19/09/2013`

The shortest format, specifying the day of month, month and year in numeric form.	

The date provides a description of the date, suitable for display to the user. It's output varies depending on the Mac system software and language used. For this reason, if you want to perform arithmetic on a date, or store a date for later computation, it is far better to use **timeStamp** or **dateItems**. The output of those functions does not depend on the language of the Mac system and can be easily used to perform arithmetic on dates and times.

Related

time, convert, weekDay, month, timeStamp, dateItems

dateItems

Forms

```
the dateItems
dateItems()
```

Description

Returns a list of comma-separated items representative of the current system clock date and time. Specifically, the year, month, day of month, hour, minute, second and day of week (all as numbers).

Comments

Most useful when used with convert to perform date and time arithmetic.

If the time is exactly 4:29pm on Thursday, 19 September 2013, the dateItems would return:

```
2013,9,19,16,29,0,5
```

Related

date, convert, weekDay, month, time, timeStamp

div (integer division)

Forms

```
numberA div numberB
```

Examples

```
9 div 5 -- yeilds 1
```

Description

Returns the whole number result (quotient) of the division of *numberA* by *numberB*. The decimal fraction component of the result is discarded.

Related

mod, divide

divide

Forms

```
numberA / numberB
divide numberA by numberB
```

Examples

```
10 / 5 -- yields 2
9 / 5 -- yields 1.8
```

Description

Returns the result of the division of *numberA* by *numberB*.

Comments

Where performance is important, use the first form. The second form (the `divide` message) is significantly slower as the subtract message must traverse CinsImp's message hierarchy. The short form is executed directly.

Related

add (+), subtract (-), multiply (*), div, mod

DontSearch

Forms

```
[the] dontSearch of element
```

Examples

```
the dontSearch of field id 1
dontSearch of card "Preferences"
```

Description

Yields `true` if the dontSearch property of an element (field, card or background) is on, otherwise `false`. Can be modified with **set**.

Find will never search the content of elements whose dontSearch property is on.

Related

Find

exit

Forms

```
exit handler
exit to user
exit repeat
```

Examples

```
exit to user -- abort the current script
```

```
    exit calculateServingSize -- return from the message handler
```

Description

The first form causes CinsTalk to return from the currently running message handler, to the calling handler, where the script continues to run. If the current handler is a function handler, it will return an empty string by default.

The second form causes CinsImp to immediately abort the currently running script and return control to the user.

The third form causes CinsTalk to escape from the currently running repeat loop. If there is no repeat loop, it causes a syntax error (figure X).

Related

pass, **repeat Loops**, **Handlers**

Exponent

Forms

```
    base ^ index
```

Examples

```
    2 ^ 3 -- yields 8
```

Description

Yields the result of base to the power of index.

find

Forms

```
    find [normal | chars | characters | word | phrase | string]
         searchText [in field]
```

Examples

```
    find “apple”
    find word “water” -- finds water but not watermellon
    find “Josh” in field “FirstName”
    find true in bkgnd field “isMember” of marked cards
```

Description

Searches the fields of the current card for the specified *searchText*. If the text is not found, continues the search for each of the remaining cards of the stack, in sequence. Precisely how a match is determined depends on the mode (see Comments below.)

Issuing the same find command will find the next match, and so on.

A yellow highlight is displayed around the first word of the matching text (it will disappear automatically when you perform any other action.)

Comments

There are several modes of operation, that determine what CinsImp considers a match:

- `normal` – (the default mode) finds words that *begin* with the *searchText* words, for example, "apple" will match "**Apple**" and "**apple**s", but not "pineapple".
- `chars`, `characters` – finds words that *contain* the *searchText* words, for example, "apple" will match "pine**apple**s", "**apple**s" and "apple".
- `words` – finds the *searchText* words exactly as they appear, for example, "apple" will match "**apple**", "**Apple**", and "**APPLE**", but not "apples" or "pineapple".
- `string` – finds the entire *searchText* exactly as it appears, for example, "apple into the pot" will match "put the pine**apple into the pot**s".
- `whole` – finds the *searchText* as a whole word phrase, similar to words mode, but the words must be consecutive and in the same sequence they appear in *searchText*, for example, "apple into the pot" will match "put the **apple into the pot**.", but not "put the pineapple into the pot."

Search terms: The *normal*, *characters* and *words* modes all consider the text as a set of search 'terms' – where each term is a word in the text. The *string* and *whole* modes consider the text as a single search term.

Search terms do not have to appear within the same field. CinsImp attempts to find each of the search terms somewhere on the current card. The terms can be found in different fields and provided all terms are found on a card, that card will still be matched.

Limiting the search: You can limit the search to a specific card or background field. You can also limit the search to only the marked cards. CinsImp will never search fields, cards or backgrounds that have the *Don't Search* property set to true.

CinsImp searches Check Box style fields too. If the field is checked, it is searched as if it contains the text "true". Otherwise as if it contains the word "false".

If a **visual effect** was queued prior to calling find, that effect will be used to transition gracefully from the current card to the first card found that matches the search parameters.

If CinsImp cannot find a match anywhere in the current stack, it displays an error (figure X).

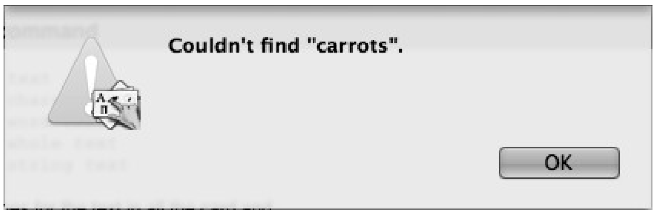
Couldn't find "carrots".

Figure 2: CinsImp can't find the text

Related

sort, **go**, **visual effect**

Function Handlers

Form

```
function message [param1, param2, ...]
  statements
  ...
  return result-value
end message
```

Description

Function handlers allow an object (such as a button or card) to respond to a function *message*. When CinsImp runs a function handler, it runs each of the statements that appear between the `function` and `end` keywords, in sequence.

In this example, a function is defined that adds two numbers and returns the result:

```
function addTwoNumbers a, b
  return a + b
end mouseUp
```

The sequence of CinsTalk command handlers within a script is not important.

From CinsImp's perspective there is very little difference between a **Command Handler** and a **Function Handler**. The only difference is in the way they're invoked. See **Function Invocation** for more information about using functions.

Related

Command Handlers, **Function Invocation, Exit, Next, Pass**

Function Invocation

Form

```
message([param1, param2, ...])
```

Description

Function Invocation is the technique used to run a function, often one you've defined yourself by creating a custom **Function Handler**. The following example invokes the CinsTalk **date** function:

```
put date() -- displays the date in the message box
```

You can also pass a parameter to the command. In the next example, we invoke the CinsTalk **sum** command and provide a list of numbers as a parameter:

```
put sum(3,7,2) -- displays 12 in the message box
```

When you define your own functions by creating a custom **Function Handler**, you can invoke them in a similar way:

```
on mouseUp
  answer createUserGreeting()
end mouseUp

function createUserGreeting
  return "Hello user."
end createUserGreeting
```

This example invokes the `createUserGreeting` function which returns the text "Hello user.". The button then displays the text using the **answer** command (see Figure X).

Related

Function Handlers, Command Invocation

get

Forms

```
get expression
```

Description

Evaluates expression and places the result into the special variable **it**.

Related

It

global

Forms

```
global name [, name [, … ]]
```

Examples

```
global theUsername
```

Description

Defines a global variable and imports it into the scope of the current message handler. You must import a global variable before you can refer to it within a handler, otherwise CinsImp treats the variable name as local to the handler in which it is used.

Comments

The **it** variable is a special global variable that is always available. You do not need to import it using global.

The **Variable Watcher** palette allows you to inspect and change the values of the global variables in a CinsImp stack at any time (see page X).

Related

It

go

Forms

```
go [to] {next | prev[ious] | ordinal | any}
go [to] card
```

Examples

```
go to last card
go to card “Preferences”
go card id 91
go first
go to any
go to third card of this background
go to card 3 -- goes to the third card in the stack
go card theCardNumber
```

Description

Takes the user to the specified card.

If immediately preceded by **visual effect**, the specified effect will be used to transition gracefully to the next card.

Related

visual effect, **find**

greater or equal

Forms

```
value1 >= value2
value1 ≥ value2
```

Examples

```
field “serves” >= numberOfGuests
x >= 2.5
```

Description

Returns true if *value1* is greater than or equal to *value2*, false otherwise.

Comments

If the two values are strings, CinsImp will perform a lexical comparison, returning true if *value1* would sort at the same level or below *value2* in ascending sort order.

If either of the values is a number, CinsImp will try to convert the other value to a number prior to comparison.

See the comments of **is equal** (page X) for further information about how CinsImp compares two values for equality.

Related

less than, less or equal, greater than, if

greater than

Forms

```
value1 > value2
```

Examples

```
field “serves” > numberOfGuests
x > 2.5
```

Description

Returns true if *value1* is greater than *value2*, false otherwise.

Comments

If the two values are strings, CinsImp will perform a lexical comparison, returning true if *value1* would sort below *value2* in ascending sort order.

If either of the values is a number, CinsImp will try to convert the other value to a number prior to comparison.

Related

less than, less or equal, greater or equal, if

ID

Forms

```
[the] id of element
```

Examples

```
the id of card button “Start”
the id of this card
```

Description

Yields the ID assigned to a stack *element* by CinsImp when it was created. Elements that are automatically assigned an ID include cards, backgrounds, buttons and fields.

Comments

`id` is read-only. It is supplied automatically by CinsImp and cannot be modified by you or the user of your stack.

The ID is the fastest way to refer to an object in CinsImp, and one of the safest. The ID of an element is guaranteed to remain the same for the life of a stack. Whereas the **name** and **number** of an element can be changed.

IDs are always positive whole numbers, greater than zero.

Related

name, number

Idle

Description

This message is sent to the current card at regular intervals when there is no script running.

Parameters

None.

If..Else Decisions

Forms

```
if expression then
  statements
[else if expression then
  statements
[else
  statements
]]
end if

if expression
then statement
[else statement]
```

Description

If..Else Decisions allow you to do different things in your scripts conditional on the value of some CinsTalk *expression*. The *expression* must evaluate to either `true` or `false`.

In this example, a script decides what greeting to display based on the time of day:

```
if item 4 of the dateItems > 12 then -- is the Hour > 12 PM?
  answer "Good afternoon."
else
  answer "Good morning."
end if
```

CinsTalk is reasonably flexible about how you structure an If..Then Decision, however, it is recommended that you stick to the two *Forms* described above for clarity.

This example asks the user for their name, and if it's "Josh", displays a customised greeting:

```
ask "What is your name?"
if it is "Josh" then
  answer "Hi Josh!  Nice to see you again."
else
  answer "I don't know who you are."
```

```
        end if
```

Related

Repeat Loops

is equal

Forms

```
        value1 is value2
        value1 = value2
```

Examples

```
        "Apple" is "apple" -- yeilds true
        "pear" is "apple" -- yeilds false
        field "lastName" = "Smith"
        x = 2.5
        x is true
```

Description

Returns the result of comparing *value1* and *value2* for equality; true if both values are equal, false otherwise.

Comments

When comparing textual string values, such as “apple” and “pear”, CinsImp uses a case-insensitive string comparison. String comparisons in CinsImp are almost always case-insensitive.

If the two values are different types, for example, value1 is “apple” and value2 is 5, CinsImp will try to convert one of the values so they are the same type before comparison. If the comparison fails, in this case because CinsImp is unable to convert “apple” to a number, an error message will result (Figure X).

Technical Note: Because of the way numbers with decimal fractions are represented by the computer, for example, 3.14159, CinsImp effectively takes the arithmetic difference between the two values and compares the result to a threshold – 0.00000000001. If the difference is less than this threshold, the values are considered equal.

Related

is not equal, **if**

is not equal

Forms

```
        value1 is not value2
        value1 <> value2
        value1 ≠ value2
```

Examples

```
        "Apple" is not "apple" -- yeilds false
        "pear" is not "apple" -- yeilds true
```

```
field “lastName” <> “Smith”
x <> 2.5
x is not true
```

Description

Returns the result of comparing *value1* and *value2* for inequality; true if both values are not equal, false otherwise.

Comments

See comments for **is equal** (page X) for more information on how this comparison behaves. The behaviour of this comparison is simply the inverse of **is equal**.

Related

is equal, **if**

it

Forms

```
it
```

Description

A special global variable that is always accessible in CinsTalk. Used by **get**, **answer** and **ask** commands and their variants.

Comments

The following handler asks the user for their name and then displays an alert box to greet them accordingly:

```
on fetchUserName
  ask “What is your name?”
  if it is empty then exit fetchUserName
  answer “Hello” && it
end fetchUserName
```

Related

get, **answer**, **ask**, **answer file**, **answer folder**, **ask file**

item

Forms

```
item startOffset [to endOffset] of container
the ordinal item of container
```

Examples

```
item 3 of “apples,peaches,cream” -- yields “cream”
the first item of the dateItems – yields the year,
                                                  for
example, 2013
any item of field “groceries”
```

Description

Yields the specified range of items from *startOffset* to *endOffset* of the *container*. Items are delimited by itemDelimiter.

The second form allows the use of an *ordinal*, such as first, second or last (see X for a complete list of ordinals).

The item range of the container can also be easily modified using **put**.

Related

itemDelimiter, **character**, **word**, **line**, **put**

itemDelimiter

Forms

```
[the] itemDelimiter
```

Description

Used by item to allow accessing particular character-delimited components of a string. By default, the itemDelimiter is a comma (,).

Comments

The following example changes the itemDelimiter to a colon:

```
set the itemDelimiter to ":"
```

Related

item, set

less or equal

Forms

```
value1 <= value2
value1 ≤ value2
```

Examples

```
field "serves" <= numberOfGuests
x <= 2.5
```

Description

Returns true if *value1* is less than or equal to *value2*, false otherwise.

Comments

If the two values are strings, CinsImp will perform a lexical comparison, returning true if *value1* would sort at the same level or above *value2* in ascending sort order.

If either of the values is a number, CinsImp will try to convert the other value to a number prior to comparison.

See the comments of **is equal** (page X) for further information about how CinsImp compares two values for equality.

Related

less than, greater or equal, greater than, if

less than

Forms

```
value1 < value2
```

Examples

```
field "serves" < numberOfGuests
x < 2.5
```

Description

Returns true if *value1* is less than *value2*, false otherwise.

Comments

If the two values are strings, CinsImp will perform a lexical comparison, returning true if *value1* would sort above *value2* in ascending sort order.

If either of the values is a number, CinsImp will try to convert the other value to a number prior to comparison.

Related

greater than, **less or equal**, **greater or equal**, **if**

line

Forms

```
line startOffset [to endOffset] of container
the ordinal line of container
```

Examples

```
line 1 to 2 of theText
the middle line of theLines
any line of field "ingredients"
```

Description

Yields the specified range of lines from *startOffset* to *endOffset* of the *container*. The line range of the container can also be easily modified using **put**.

The second form allows the use of an *ordinal*, such as first, second or last (see X for a complete list of ordinals).

Related

character, word, item, put

Locked

Forms

```
[the] locked of field
```

Examples

```
the locked of field "RecipeDescription"
```

Description

Yields true if the field content is locked to prevent editing, or false otherwise. Can be modified with set.

Related

CantModify

max

Forms

```
max(item1, item2, …, itemN)
[the] max of items
```

Examples

```
max of "21, 51, 4, -7, 23" -- returns 51
max(1,8,9,3,3,2,5) -- returns 9
```

Description

Returns the largest number of a list of items.

Related

min, average, sum

me

Forms

```
me
```

Description

Yields the object which contains the currently running script **handler**.

Related

Target

min

Forms

```
min(item1, item2, …, itemN)
[the] min of items
```

Examples

```
min of "21, 51, 4, -7, 23" -- returns -7
min(1,8,9,3,3,2,5) -- returns 1
```

Description

Returns the smallest number of a list of *items*.

Related

max, average, sum

mod (integer remainder)

Forms

```
numberA mod numberB
```

Examples

```
9 mod 5 -- yeilds 4
```

Description

Returns the whole number remainder (modulus) of the division of *numberA* by *numberB*.

Related

div, divide

month

Forms

```
the [short | long] month of monthOfYear
month(monthOfYear)
```

Examples

```
the month of 9 -- yields Sep
the long month of 9 -- yields September
month(9) -- yields Sep
```

Description

Returns the name of the *monthOfYear* using the current system language. The default format is the short format.

Comments

On Australian English systems, the short format yields a 3-letter abbreviation. The long format yields the full name.

Related

date, dateItems, convert, weekDay, time, timeStamp

MouseUp

Description

This message is sent to an object (button or card) when the user presses and releases the mouse over it. This is the most appropriate system event to handle if you want to do something when an object is clicked.

Parameters

Command Handlers

multiply *

Forms

```
numberA * numberB
multiply numberA by numberB
```

Examples

```
2 * 15 - yeilds 30
x * y
```

Description

Returns the multiplication of numberA and numberB.

Comments

Where performance is important, use the first form. The second form (the `multiply` message) is significantly slower as the subtract message must traverse CinsImp's message hierarchy. The short form is executed directly.

Name

Forms

```
[the] [short | abbr[ev[iated]] | long] name of element
[the] [short | abbr[ev[iated]] | long] name of the system
```

Examples

```
the name of this card
short name of field 3
the long name of this stack
the long name of the system -- yields "Mac OS X"
```

Description

Yields the name assigned to an *element* by the user or a script. The first form is supported by the button, field, card, background, stack elements.

Can be modified using **set**.

Comments

There are three different formats of the name:

Formats	
`the long name`	`card button “Start” of card “timer” of stack “Kitchen Hand”`
The longest format, specifying the identity of the element, it's parent element and the stack.	
`the abbr[ev[iated]] name`	`card button “Start”`
An abbreviated format, specifying the identity and element type.	
`the short name`	`Start`
The shortest format, specifying only the name itself.	

The default is the abbreviated format. If an element has no name, the **ID** is used instead.

Related

id, number, version

Negate

Forms

```
-number
```

Examples

```
-x
-2
```

Description

Yields the negative of a number.

Newline

Forms

```
newline
```

Examples

```
“The answer is:” & newline & theAnswer
```

Description

The newline constant evaluates to a line break character and is useful if you want to produce a text string with more than one line.

The example above, with a variable value of “Chicken Soup”, would produce the following output:

```
The answer is:
Chicken Soup
```

Related

quote

next repeat

Forms

```
next repeat
```

Description

Causes CinsTalk to continue running the current repeat loop from the top – jumping straight to the next iteration (if the loop conditions are still satisfied) or exiting the loop otherwise.

Related

exit, repeat Loops

not

Forms

```
not boolean
```

Examples

```
not true
not false
not the dontSearch of field 1
not x
not the result
```

Description

Returns the logical inverse of *boolean*. Thus if *boolean* is true, it returns false. If *boolean* is false, it returns true.

Related

and, or, if

Number

Forms

```
[the] number of element
```

```
[the] number of {cards | backgrounds | buttons | fields}
           [of element]
[the] number of {char[acter]s | words | lines | items}
           of string
```

Examples

```
the number of field "Description"
number of buttons of this card
number of items of theIngredients
the number of words of "Chicken Soup" -- yields 2
```

Description

Depending on the form used, yields either the integer number of the *element* in the sequence of all such elements, or the count of how many elements of a specific type exist within another *element* or *string*.

Comments

The number assigned a button or field represents where it appears in the tab-sequence. You can move an element backwards or forwards in the sequence using the **Send...** commands in the **Object menu** (see X).

The `number` of an element can't be modified by a script.

When asking for the number of items of a *string*, CinsImp uses the **itemDelimiter** to determine item boundaries. By default, the **itemDelimiter** is a comma (,).

Related

id, name

numberFormat

Forms

```
[the] numberFormat
set [the] numberFormat to format
```

Examples

```
set numberFormat to "0.00" - 0.14159 displayed as 0.14
set numberFormat to "#.000" - 0.14159 displayed as .142
set numberFormat to "0.000" - 0.14159 displayed as 0.142
set numberFormat to "0" - 0.14159 displayed as 0
set numberFormat to "00.000" - 0.14159 displayed as 00.142
set numberFormat to "#.000000" - 0.14159 displayed as .141590
set numberFormat to "#.000###" - 0.14159 displayed as .14159
```

Description

Allows control of the formatting and precision of numbers when they're displayed. Can be modified with **set**.

The default *format* is "0.######".

Comments

The following lines demonstrate how numberFormat can be used to format a currency value with two decimal places:

```
put 3.5 into theAmount
set the numberFormat to "0.00"
put "$" & theAmount -- message box displays "$3.50"
```

Number format consists of one or more zeros (0), hashes (#) and a decimal point. A zero indicates a required digit. A hash indicates an optional digit.

Rounding: If a number has more digits of precision in the decimal fraction than will fit in a given format, the last digit is rounded accordingly:

```
set numberFormat to "0.000" -- 1.2345 displays as 1.235
                               -- 1.2344 displays as 1.234
```

If the digit immediately following the last is 5 or greater, the last digit is rounded up. Otherwise it stays the same.

Logically, valid formats can only ever consist of the following elements, in sequence:

- zero or more hashes (#)
- zero or more zeros (0)
- an optional decimal point (.)
- zero or more zeros(0)
- zero or more hashes (#)

If a number format is invalid, CinsImp ignores the number format all together and the results of displaying a number are undefined.

The numberFormat doesn't affect how CinsImp performs calculations internally. It is only used when a number is displayed in a field, the message box or concatenated with a text string.

Open

Forms

```
open file path-name
```

Examples

```
open file "RecipeExport.txt"
open file "/Users/josh/Documents/RecipeExport.txt"
```

Description

Tries to open the specified file *path-name* as a text file, for reading and writing. If the file is on a locked volume or disk, the file will be opened read-only.

If the open is unsuccessful, CinsTalk sets the result to “Can't open that file.”. You should always check the result after opening a file.

Comments

path-name can be a simple name, or a complete path to the file. If a simple name is specified, CinsImp assumes the file is in the same folder as the current stack.

You can get the path-name of a file or folder in the Finder by examining the *Where:* field in the Finder's *Get Info* command.

Geek Alert: You should not try to open a binary file with CinsImp. CinsImp will not read or write binary files properly and has no mechanism to work with these files at this time.

IMPORTANT: Under no circumstances should you attempt to open a CinsImp stack file with the open command. It is highly probable you will corrupt the stack beyond repair if you do!

Related

ask file, **answer file**, **close**, **read**, **read line**, **write**

OpenBackground

Description

This system message is sent to the current card immediately after arriving at the card and from a different background. CinsImp will send the message regardless of what triggers the transition to another background. The message is also sent when a stack is opened.

Parameters

None.

Related

OpenCard, OpenStack, CloseBackground

OpenCard

Description

This system message is sent to the current card immediately after arriving at the card. CinsImp will send the message regardless of what triggers the transition to another card. The message is also sent when a stack is opened.

Parameters

None.

Related

OpenBackground, OpenStack, CloseCard

OpenStack

Description

This system message is sent to the current card immediately after opening a stack.

The sequence of events when opening a stack is as follows:

- OpenStack
- OpenBackground
- OpenCard

Parameters

None.

or

Forms

```
boolean1 or boolean2
```

Examples

```
x or y
true or true
true or false
(x = 2) or (5 < y)
the result or it
```

Description

Returns true if *boolean1* or *boolean2* is true. Otherwise, it returns false.

Related

and, not, if

pass

Forms

```
pass handler
```

Examples

```
pass mouseUp
```

Description

Passes the message currently being handled to the next object in the message hierarchy. For more information on the message hierarchy, see X.

Related

Exit

Pi

Forms

```
pi
```

Description

A mathematical constant that always evaluates to an approximation of PI, specifically 3.14159265358979323846.

put

Forms

```
put expression
put expression {into | after | before} container
```

Examples

```
put 5 + 2 -- puts 7 into the message box
put empty into it -- clears the contents of it
put 5 into x -- the variable x now holds the value 5
put "apricots" & newline after field "ingredients"
put field "RecipeTitle" into theRecipeTitle
```

Description

The first form simply evaluates *expression* and places the result into the message box.

The second form evaluates the *expression* and places the result into the *container*. The after and before modifiers allow you to append or prepend the result to the *container* respectively.

Comments

put can also be used with character, word, line or item, to modify specific components of a text string:

```
put "apples,oranges,carrots,peaches" into theFoods
put "plums" into item 2 of theFoods
-- theFoods now contains: apples,oranges,plums,peaches
```

This is a very easy to understand, very powerful mechanism for text manipulation that is unique to languages like CinsTalk.

Quote

Forms

```
quote
```

Examples

```
"The recipe" && quote & recipeTitle & quote && ¬
           "is not sufficient to feed your family." && ¬
           "Are you sure you want to cook that for dinner?"
```

Description

The quote constant evaluates to a single quote character (") and is useful when building messages to the user.

The example above, with a variable value of "Sauté Snail", would produce the following output (without line breaks):

```
The recipe “Sauté Snail” is not sufficient to feed your family. Are
you sure you want to cook that for dinner?
```

Related

Newline

random

Forms

```
random(maximum)
the random of maximum
```

Examples

```
random(2)
the random of biggestNumber
```

Description

Generates a random number between 1 and maximum.

Comments

This handler picks a random vegetable from broccoli, carrot or onion. It asks the user to guess which one it picked. If the user is correct, it rewards them with dessert. Otherwise, it apologies sincerely:

```
on guessTheVegetable
  put random(3) into vegetableNumber
  put "broccoli,carrot,onion" into vegetableList
  answer "Which vegetable am I thinking of?" ¬
  with "Broccoli" or "Carrot" or "Onion"
  if it is item vegetableNumber of vegetableList then
    go to card "Dessert"
  else
    answer "Sorry, that wasn't it!"
  end if
end guessTheVegetable
```

Related

any

Read

Forms

```
read from file path-name [at character-offset]
      for character-count
read from file path-name [at character-offset]
      until {end | eof | character}
```

Examples

```
read from file “RecipeImport.txt” until end
read from file “Ingredients.txt” at 10 for 50
read from file "Sentences.txt" until "."
```

Description

Reads text from the file *path-name*, previously opened with **open**. The text is placed into the **it** variable.

If there is a problem reading from the file, **the result** contains an error message. It will be empty if the read was successful.

File pointer: CinsTalk maintains a pointer to the location of the last read or write for the file. Whenever you read from or write to a file, it updates this pointer. So each successive read will read from the end of the last read, and so on. When you first open a file, the file pointer is at the start of the file.

If you specify a *character-offset*, the read will begin at that offset from the start of the file. CinsImp's internal file pointer is moved accordingly.

The second form of read allows you to specify an ending *character*. If a character is specified, CinsImp will read until it encounters the specified character and then stop. The file pointer will be placed immediately following that character, ready to read or write again. The text will be returned to your script as normal, in the **it** variable.

Related

open, **read line**, **write**, **close**

Read Line

Forms

```
read line from file path-name
```

Examples

```
read line from file "Ingredients.txt"
```

Description

Effectively a shorter form of **read**, reads a single line of text from the file *path-name*, previously opened with **open**. The text is placed into the **it** variable.

If there is a problem reading from the file, **the result** contains an error message. If the read was successful, but the end of the file was reached, **the result** will contain "End of file.". Otherwise **the result** will be empty.

File pointer: CinsTalk maintains a pointer to the location of the last read or write for the file. Whenever you read from or write to a file, it updates this pointer. So each successive read will read from the end of the last read, and so on. When you first open a file, the file pointer is at the start of the file.

Related

open, **read**, **write**, **close**

Repeat Loops

Description

Repeat Loops allow you to do something repetitively. The *statements* within a Repeat Loop are run over and over until some condition is met, or CinsTalk encounters an **exit**.

There are many different forms of Repeat Loop, but all have the same basic structure:

```
repeat
  statements-to-be-repeated
  ...
end repeat
```

Repeat Forever

```
repeat [forever]
  statements
  ...
end repeat
```

Description

Doesn't actually run forever! This type of loop will run until CinsImp encounters an **exit**, you manually abort the script or the stack is forcibly closed.

This is the simplest form of loop. All the other forms can be stopped in the same way, however, they all introduce additional conditions under which the loop will finish.

Repeat Number

```
repeat [for] number [times]
  statements
  ...
end repeat
```

Description

This type of loop will finish after a set *number* of iterations. This example will beep 3 times:

```
repeat 3 times
  beep
end repeat
```

Repeat While

```
repeat while expression
  statements
  ...
end repeat
```

Description

This type of loop will continue iterating while *expression* evaluates to `true`. The *condition* is always evaluated at the beginning of the loop, and at the end of each iteration after that.

Repeat Until

```
repeat until expression
```

```
    statements
    ...
end repeat
```

Description

This type of loop will continue iterating until *expression* evaluates to `true`. The *condition* is always evaluated at the beginning of the loop, and at the end of each iteration after that.

Repeat Counted

```
repeat with variable = start [down] to end
  statements
  ...
end repeat
```

Description

This type of loop will continue iterating while *variable* evaluates to something other than *end*. The condition is always evaluated at the beginning of the loop, and at the end of each iteration after that.

Prior to beginning the first iteration, the *variable* is initialised with the *start* value.

At the end of each iteration, CinsTalk automatically increments or decrements (if the `down` keyword is present) the counting *variable*.

Related

If..Then Decision

result

Forms

```
the result
result()
```

Description

Yields the result of the last command performed by CinsImp, for selected commands. Can also be set by using **return** in a **custom handler**.

It is also the result of the last expression evaluated in the message box.

round

Forms

```
round(number)
[the] round of number
```

Examples

```
round (2.4) -- returns 2
round (2.75) -- returns 3
the round of PI -- returns 3
```

Description

Returns the number rounded to the nearest whole integer. If the decimal fraction component of the number is 0.5 or greater, it rounds up. Otherwise it rounds down.

Related

trunc

set

Forms

```
set [the] property [of object] to value
```

Examples

```
set the itemDelimiter to ","
set the name of card button 1 to "Stop"
set the textStyle of field "alert" to "bold,italic"
set numberFormat to "0.00"
set the locked of field "RecipeTitle" to true
set the cantDelete of this card to true
```

Description

Allows the changing of a *property* value to a new *value*.

Shared

Forms

```
[the] shared of background-field
```

Examples

```
shared of background field "StackTitle"
```

Description

Yields `true` if the shared property of a *background-field* is on, otherwise `false`. Can be modified with **set**.

If a background field is shared, the content within it appears the same on all cards shared by that background. Otherwise each card can have it's own content within a that field.

sin

Forms

```
sin(number)
[the] sin of number
```

Examples

Description

Returns the sine of a *number* in radians.

Related

atan, cos, tan

sort

Forms

```
sort [[the] cards [of {this stack | background}]
      [ascending | descending] by sortKey
```

Examples

```
sort by field "LastName"
sort the cards by the weekday of field "dayOfWeek"
sort descending by name of this card
sort cards of first background by field "RecipeTitle"
```

Description

Rearranges the cards of the entire stack.

Comments

By default, CinsImp sorts the cards in *ascending* order, from lowest to highest.

Only the *sortKey* is required. It can be any valid CinsTalk expression and is used by CinsImp to compare each card. The *sortKey* doesn't have to be a field. You could use a property or function. For example, to reverse the order of the cards:

```
sort descending by the number of this card
```

If *background* is specified, only the cards of the specified background will be sorted.

To sort by multiple criterion, for example, by Last Name and then First Name, you could use the following:

```
sort by field "FirstName"
sort by field "LastName"
```

Note that the sort commands are issued in the reverse order to that which you want.

CinsImp implements a stable sort. This means that second and subsequent sorts move the cards as little as necessary. If two cards have the same *sortKey*, they are not moved.

Sorting the cards does not have any effect on the current card.

Related

find

sqrt

Forms

```
sqrt(number)
[the] sqrt of number
```

Examples

```
sqrt(9) -- returns 3
sqrt of 64 -- returns 8
```

Description

Returns the square root of the *number*.

Related

^

Style

Forms

```
[the] style of element
```

Examples

```
the style of card button 1
the style of field "Instructions"
```

Description

Yields the style of the *element* (button or field). Can be modified with **set**.

Comments

Element styles affect the general behavior and appearance of an *element*. Buttons can have the following styles:

Button Styles	
`transparent`	The button background is see-thru.
`push`	Draws a border and 3D push button appearance. The button appears completely opaque atop the card or background picture.

Fields can have the following styles:

Field Styles	
`text`	The field behaves as a simple text editing field.
`check box`	The field appears and behaves as a check box.

Related

Border

Subtract

Forms

```
numberB - numberA
subtract numberA from numberB
```

Examples

```
10 - 7
subtract 7 from 10 -- yeilds 3
```

Description

Returns the result of subtracting numberA from numberB.

Comments

Where performance is important, use the first form. The second form (the `subtract` message) is significantly slower as the subtract message must traverse CinsImp's message hierarchy. The short form is executed directly.

sum

Forms

```
sum(item1, item2, …, itemN)
[the] sum of items
```

Examples

```
sum of "21, 51, 4, -7, 23" -- returns 92
sum(1,8,9,3,3,2,5) -- returns 31
```

Description

Returns the arithmetic sum (addition) of a list of *items*.

Related

average, **min**, **max**

tan

Forms

```
tan(number)
[the] tan of number
```

Examples

Description

Returns the tangent of a *number* in radians.

Related

atan, cos, sin

target

Forms

```
target
```

Description

Yields the object that was the initial target of the message currently being handled, which may not necessarily be the object that is currently handling the message.

Related

Me

Text

Forms

```
[the] text of field
```

Examples

```
text of field "RecipeTitle"
```

Description

Yields the text of the *field*. Can be modified with **set**.

Comments

`text` is superfluous, as the content of a field can be accessed just by referencing the field:

```
put "Apricot Chicken" into field "RecipeTitle"
put field "RecipeTitle" into theTitle
           -- theTitle now contains "Apricot Chicken"
```

It is included purely for language completeness.

TextFont

Forms

```
[the] textFont of element
```

Examples

```
the textFont of field "RecipeTitle"
```

Description

Yields the default setting of font for an *element* (field or button). Can be modified with **set**.

Related

textStyle, **textSize**

TextSize

Forms

```
[the] textSize of element
```

Examples

```
the textSize of field "RecipeTitle"
```

Description

Yields the default setting of text point size for an *element* (field or button). Can be modified with **set**.

Related

textFont, textStyle

TextStyle

Forms

```
[the] textStyle of element
```

Examples

```
the textStyle of field "RecipeTitle"
```

Description

Yields the default setting of bold and italic attributes for an *element* (field or button). Can be modified with **set**.

Comments

If both attributes are set, yields a comma-delimited list: "bold, italic".

This example applies the bold attribute to a background field:

```
set the textStyle of field "RecipeTitle" to "bold"
```

Related

textFont, **textSize**

time

Forms

```
the [short | long] time
time()
```

Examples

```
time()
the long time
```

Description

Returns the current system clock time, in short (the default) or long format.

Comments

The precise output of both formats of time is configurable in the Mac system preferences.

On Australian English systems, the time formats will usually produce outputs similar to these:

Formats	
`the long time`	`4:29:48 PM`
The long format, specifying hour, minute and seconds, and meridian designation if the system is configured for 12-hour (civilian) time.	

`the short time`
`date()`
	`4:29 PM`

The short format, specifying the hour and minute, and meridian designation if the system is configured for 12-hour (civilian) time.	

The time provides a description of the time, suitable for display to the user. It's output varies depending on the Mac system software and language used. It is not intended for use in calculations. For this reason, if you want to perform arithmetic on a time, or store a time for later computation, it is far better to use **timeStamp** or **dateItems**. The output of those functions does not depend on the configuration of the Mac system and can be easily used to perform arithmetic on times and dates.

Related

date, **convert**, **weekDay**, **month, timeStamp, dateItems**

timeStamp

Forms

```
the timeStamp
timeStamp()
```

Examples

```
the timeStamp
```

Description

Returns the current system clock time expressed as a number of seconds since the CinsImp epoch – 19 September 1983, at 4.30pm Australian Central Standard Time. Useful for recording the time and date something occurred for later comparison or calculation, and to avoid issues converting between differing international date and time formats.

Comments

The following fragment calculates the date at this time tomorrow:

```
the timeStamp + 86400 -- there are 86400 seconds in a day
```

Note: If you want to add or subtract more than a day, it's usually more precise and safer to use **dateItems** to perform calculations as CinsImp can then take account of the calendar.

Related

date, convert, weekDay, month, time, dateItems

trunc

Forms

```
trunc(number)
[the] trunc of number
```

Examples

```
trunc of 2.81 -- returns 2
trunc of -7 -- returns -7
```

Description

Returns the whole integer component of a real number.

Related

round

Version

Forms

```
the {short | abb[re[viated]] | long] version of the system
```

Examples

```
the long version of the system -- for example, 10.8.4
```

Description

Yields the version of the host operating system.

Comments

Three formats are available:

Formats	
`the long version`	`10.8.4`
The longest format, specifying the major, minor and bug fix components.	
`the abbr[ev[iated]] version`	`10.8`

An abbreviated format, specifying only the major and minor version components.	
`the short version`	`10.0804`
Provides the version as an easily comparable real number.	

For example, to easily check what version of the system is running, you should use the short format of version:

```
if the short version of the system < 10.0804 then
  answer "Your system software is not up to date." & ¬
            Please run software update!"
end if
```

Related

Name

visual effect

Forms

```
visual [effect] effect [speed]
      [to {card | white | black | grey}]
```

effect:

```
cut
dissolve
wipe left
wipe right
wipe up
wipe down
```

speed:

```
very slow[ly]
slowly
normal
fast
very fast
```

Examples

```
visual effect dissolve slowly
visual effect wipe left to black
visual wipe right to card
```

Description

Queues a visual effect for playback when the card changes.

Related

go, **find**

Wait

Forms

```
wait [for] interval [tick[s] | sec[s] | second[s]]
wait {while | until} expression
```

Examples

```
wait 3 seconds
wait for 20 ticks
wait until theGuestCount > 2
```

Description

Causes the running script to pause. The script resumes after the time *interval* has elapsed, or when the conditional *expression* is satisfied.

If you do not specify the units for the first form, seconds are assumed.

Comments

While CinsTalk is paused, your stack remains unusable as it would be with any CinsTalk script active. If you want to check a condition periodically and it's not time sensitive, you should consider handling the **idle** system event instead.

Related

idle

weekday

Forms

```
the [short | long] weekday of dayOfWeek
weekday(dayOfWeek)
```

Examples

```
the weekday of 5 - yields "Thu"
the long weekday of 5 - yields "Thursday"
weekday(2) - yields "Mon"
```

Description

Returns the name of the *dayOfWeek* using the current system language. The default format is the short format.

Comments

On Australian English systems, the short format yields a 3-letter abbreviation. The long format yields the full name.

Related

date, dateItems, convert, month, time, timeStamp

word

Forms

```
word startOffset [to endOffset] of container
the ordinal word of container
```

Examples

```
word 1 to x of theText
the last word of "Banana Split" -- yields "Split"
any word of field "words"
```

Description

Yields the specified range of words from *startOffset* to *endOffset* of the *container*. The word range of the container can also be easily modified using **put**.

The second form allows the use of an *ordinal*, such as first, second or last (see X for a complete list of ordinals).

Related

character, line, item, put

Write

Forms

```
write text to file path-name
```

Examples

```
write "Apples" & newline to file path-name
write theIngredient & newline to file path-name
```

Description

Writes the text to the file *path-name*, previously opened with **open**.

If there is a problem writing to the file, **the result** contains an error message. Otherwise it will be empty.

Truncation: If you open a file and immediately start writing, CinsImp will clear the entire contents of the file prior to the first write. You will loose any data that was previously saved within.

File pointer: CinsTalk maintains a pointer to the location of the last read or write for the file. Whenever you read from or write to a file, it updates this pointer. So each successive write will write beginning at the end of the last write, and so on.

Related

open, **read**, **read line**, **close**