Euphoria 4 API Reference

openEuphoria™

This file documents the Euphoria 4 standard library API.

Created 16 April 2012

Caution

There are some formatting problems with this build.

Undocumented Include Files

Directories excluded:

- `/include/euphoria`
- `/include/std/net`
- `/include/std/win32`

Specific files excluded:

- `machine.e`
- `safe.e`
- `scinot.e`

Legacy files from Euphoria 3.1 (found in `/std`) are only documented in the Euphoria 3.1 distribution.

Built-in Methods

These routines are built into Euphoria and require no includes.

?	abort	and_bits	append
arctan	atom	c_func	c_proc
call	call_func	call_proc	clear_screen
close	command_line	compare	cos
date	delete	delete_routine	equal
find	floor	get_key	getc
getenv	gets	hash	head
include_paths	insert	integer	length
log	machine_func	machine_proc	match
mem_copy	mem_set	not_bits	object
open	option_switches	or_bits	peek
peek2s	peek2u	peek4s	peek4u
peek_string	peeks	pixel	platform
poke	poke2	poke4	position
power	prepend	print	printf
puts	rand	remainder	remove
repeat	replace	routine_id	sequence
sin	splice	sprintf	sqrt
system	system_exec	tail	tan
task_clock_start	task_clock_stop	task_create	task_list
task_schedule	task_self	task_status	task_suspend
task_yield	time	trace	xor_bits

base64

Routines

- encode
- decode

base64 API

decode

Signature: Decode to base64 (See also RFC 2045)

```
decode(sequence in)
```

```
public function
include base64.e
namespace base64
```

Arguments: ≡ `in` - must be a simple sequence of length 4 to 76.

encode

Signature: Encode to base64 (See also RFC 2045)

```
encode(sequence in, integer wrap_column = 0)
```

```
public function
include base64.e
namespace base64
```

Arguments: ≡ `in` - must be a simple sequence
≡ `wrap_column` - column to wrap the base64 encoded message to. defaults to 0, which is do not wrap

Returns: a base64 encoded sequence representing `in`.

cmdline

Constants

- NO_PARAMETER
- HAS_PARAMETER
- NO_CASE
- HAS_CASE
- MANDATORY
- OPTIONAL
- ONCE
- MULTIPLE
- HELP
- VERSIONING

---

cmdline API

AT_EXPANSION

Expand arguments that begin with '@' into the command line. (default)

Signature:

```
AT_EXPANSION
```

```
public enum
include cmdline.e
namespace cmdline
```

EXTRAS

The extra parameters on the cmd line, not associated with any

Signature:

```
EXTRAS
```

```
public constant
include cmdline.e
namespace cmdline
```

HAS_CASE

This option switch is case sensitive. See cmd_parse

Signature:

```
HAS_CASE
```

```
public constant
include cmdline.e
namespace cmdline
```

HAS_PARAMETER

This option switch does have a parameter. See cmd_parse

Signature:

```
HAS_PARAMETER
```

```
public constant
include cmdline.e
namespace cmdline
```

HELP

This option switch triggers the 'help' display. See cmd_parse

Signature:

```
HELP
```

```
public constant
include cmdline.e
namespace cmdline
```

HELP_RID

Additional help routine id. See cmd_parse

Signature:

```
HELP_RID
```

```
public enum
include cmdline.e
namespace cmdline
```

MANDATORY

This option switch must be supplied on command line. See cmd_parse

Signature:

```
MANDATORY
```

```
public constant
include cmdline.e
namespace cmdline
```

MULTIPLE

This option switch may occur multiple times on a command line. See cmd_parse

Signature:

```
MULTIPLE
```

```
public constant
include cmdline.e
namespace cmdline
```

NO_AT_EXPANSION

Do not expand arguments that begin with '@' into the command line.

Signature:

```
NO_AT_EXPANSION
```

```
public enum
include cmdline.e
namespace cmdline
```

NO_CASE

This option switch is not case sensitive. See cmd_parse

Signature:

```
NO_CASE
```

```
public constant
include cmdline.e
namespace cmdline
```

NO_HELP

Disable the automatic inclusion of -h, -? and --help as help switches.

Signature:

```
NO_HELP
```

```
public enum
include cmdline.e
namespace cmdline
```

NO_HELP_ON_ERROR

Disable the automatic display of all of the possible options on error.

Signature:

```
NO_HELP_ON_ERROR
```

```
public enum
include cmdline.e
namespace cmdline
```

NO_PARAMETER

This option switch does not have a parameter. See cmd_parse

Signature:

```
NO_PARAMETER
```

```
public constant
include cmdline.e
namespace cmdline
```

NO_VALIDATION

Do not cause an error for an invalid parameter. See cmd_parse

Signature:

```
NO_VALIDATION
```

```
public enum
include cmdline.e
namespace cmdline
```

NO_VALIDATION_AFTER_FIRST_EXTRA

Do not cause an error for an invalid parameter after the

Signature:

```
NO_VALIDATION_AFTER_FIRST_EXTRA
```

```
public enum
include cmdline.e
namespace cmdline
```

ONCE

This option switch must only occur once on the command line. See cmd_parse

Signature:

```
ONCE
```

```
public constant
include cmdline.e
namespace cmdline
```

OPTIONAL

This option switch does not have to be on command line. See cmd_parse

Signature:

```
OPTIONAL
```

```
public constant
include cmdline.e
namespace cmdline
```

OPT_CNT

The number of times that the routine has been called

Signature:

```
OPT_CNT
```

```
public enum
include cmdline.e
namespace cmdline
```

OPT_IDX

An index into the `opts` list. See cmd_parse

Signature:

```
OPT_IDX
```

```
public enum
include cmdline.e
namespace cmdline
```

OPT_REV

The value `1` if the command line indicates that this option is to remove

Signature:

```
OPT_REV
```

```
public enum
include cmdline.e
namespace cmdline
```

OPT_VAL

The option's value as found on the command line. See cmd_parse

Signature:

```
OPT_VAL
```

```
public enum
include cmdline.e
namespace cmdline
```

PAUSE_MSG

Supply a message to display and pause just prior to abort() being called.

Signature:

```
PAUSE_MSG
```

```
public enum
include cmdline.e
namespace cmdline
```

SHOW_ONLY_OPTIONS

Only display the option list in show_help. Do not display other

Signature:

```
SHOW_ONLY_OPTIONS
```

```
public enum
include cmdline.e
namespace cmdline
```

VALIDATE_ALL

Validate all parameters (default). See cmd_parse

Signature:

```
VALIDATE_ALL
```

```
public enum
include cmdline.e
namespace cmdline
```

VERSIONING

This option switch sets the program version information. If this optionis chosen by the user cmd_parse will display the program version information

Signature:

```
VERSIONING
```

```
public constant
include cmdline.e
namespace cmdline
```

build_commandline

returns a text string based on the set of supplied strings. Typically, this

Signature:

```
build_commandline(sequence cmds)
```

```
public function
include cmdline.e
namespace cmdline
```

Arguments: ≡ `cmds` : A sequence. Contains zero or more strings.

Returns: A **sequence**, which is a text string. Each of the strings in `cmds` is quoted if they contain spaces, and then concatenated to form a single string.

Comments: Though this function does the quoting for you it is not going to protect your programs from globing *, ?. And it is not specied here what happens if you pass redirection or piping characters.

When passing a result from with build_commandline to system_exec, file arguments will benefit from using canonical_path with the TO_SHORT. On Windows, this is required for file arguments to always work. There is a complication with files that contain spaces. On other platforms, this call will also return a useable filename.

Alternatively, you can leave out calls to canonical_path and use system instead.

See Also: parse_commandline, system, system_exec, command_line, canonical_path, TO_SHORT

Example 1:

```
s=build_commandline({"-d",canonical_path("/usr/my docs/",,TO_SHORT)})
-- s now contains a short name equivalent to '-d "/usr/my docs/"'
```

Example 2:

```
s = build_commandline( { "awk", "-e", "'{ print $1"x"$2; }'" } )
system(s,0)
```

cmd_parse

parses command line options and optionally calls procedures based on these options.

Signature:

```
cmd_parse(sequence opts, object parse_options = {},
sequence cmds = command_line())


public function
include cmdline.e
namespace cmdline
```

Arguments: ≡ `opts` : a sequence of records that define the various command line *switches* and *options* that are valid for the application: See Comments section for details
≡ `parse_options` : an optional list of special behavior modifiers: See Parse Options section for details
≡ `cmds` : an optional sequence of command line arguments. If omitted the output from `command_line`() is used.

Returns: A **map**, containing the set of actual options used in `cmds`. The returned map has one special key, `EXTRAS` that are values passed on the command line that are not part of any of the defined options. This is commonly used to get the list of files entered on the command line.

For example: if the command line used was *`myprog -verbose file1.txt file2.txt`* then the `EXTRAS` data value would be `{"file1.txt", "file2.txt"}`.

When any command item begins with an `@` symbol then it is assumed that it prefixes a file name. That file will then be opened and its contents used to add to the command line, as if the file contents had actually been entered as part of the original command line.

Parse Options: `parse_options` is used to provide a set of behavior modifiers that change the default rules for parsing the command line. If used, it is a list of values that will affect the parsing of the command line options.

These modifers can be any combination of:

# `VALIDATE_ALL` -- The default. All options will be validated for all possible errors.

# `NO_VALIDATION` -- Do not validate any parameter.

# `NO_VALIDATION_AFTER_FIRST_EXTRA` -- Do not validate any parameter after the first extra was encountered. This is helpful for programs such as the Interpreter itself: `eui -D TEST greet.ex -name John`. -D TEST should be validated but anything after "greet.ex" should not as it is meant for greet.ex to handle, not eui.

# `HELP_RID` -- The next Parse Option must either a routine id or a set of text strings.

The routine is called or the text is displayed when a parse error (invalid option given, mandatory option not given, no parameter given for an option that requires a parameter, etc...) occurs. This can be used to provide additional help text. By default, just the option switches and their descriptions will be displayed. However you can provide additional text by either supplying a routine_id of a procedure that accepts no parameters, or a sequence containing lines of text (one line per element). The procedure is expected to write text to the stdout device.

# `NO_HELP_ON_ERROR` -- Do not show a list of options on a command line error.

# `NO_HELP` -- Do not automatically add the switches '-h', '-?', and '--help' to display the help text (if any).

# `NO_AT_EXPANSION` -- Do not expand arguments that begin with '@.'

# `AT_EXPANSION` -- Expand arguments that begin with '@'. The name that follows @ will be opened as a file, read, and each trimmed non-empty line that does not begin with a '#' character will be inserted as arguments in the command line. These lines replace the original '@' argument as if they had been entered on the original command line.

…… ♦ If the name following the '@' begins with another '@', the extra '@' is removed and the remainder is the name of the file. However, if that file cannot be read, it is simply ignored. This allows *optional* files to be included on the command line. Normally, with just a single '@', if the file cannot be found the program aborts.

…… ♦ Lines whose first non-whitespace character is '#' are treated as a comment and thus ignored.

…… ♦ Lines enclosed with double quotes will have the quotes stripped off and the result is used as an argument. This can be used for arguments that begin with a '#' character, for example.

…… ♦ Lines enclosed with single quotes will have the quotes stripped off and the line is then further split up use the space character as a delimiter. The resulting 'words' are then all treated as individual arguments on the command line.

An example of parse options:

```
{ HELP_RID, routine_id("my_help"), NO_VALIDATION }
```

Comments: Token types recognized on the command line: # a single '-'. Simply added to the 'extras' list # a single "--". This signals the end of command line options. What remains of the command line is added to the 'extras' list, and the parsing terminates. # -shortName. The option will be looked up in the short name field of `opts`. # /shortName. Same as -shortName. # -!shortName. If the 'shortName' has already been found the option is removed. # /!shortName. Same as -!shortName # --longName. The option will be looked up in the long name field of `opts`. # --!longName. If the 'longName' has already been found the option is removed. # anything else. The word is simply added to the 'extras' list.

For those options that require a parameter to also be supplied, the parameter can be given as either the next command line argument, or by appending '=' or ':' to the command option then appending the parameter data.
For example, **`-path=/usr/local`** or as **`-path /usr/local`**.

On a failed lookup, the program shows the help by calling show_help(`opts`, `add_help_rid`, `cmds`) and terminates with status code 1.

If you do not explicitly define the switches `-h`, `-?`, or `--help`, these will be automatically added to the list of valid switches and will be set to call the show_help routine.

You can remove any of these as default 'help' switches simply by explicitly using them for something else.

You can also remove all of these switches as *automatic* help switches by using the `NO_HELP` parsing option. This just means that these switches are not automatically used as 'help' switches, regardless of whether they are used explicitly or not. So if `NO_HELP` is used, and you want to give the user the ability to display the 'help' then you must explicitly set up your own switch to do so. **N.B**, the 'help' is still displayed if an invalid command line switch is used at runtime, regardless of whether `NO_HELP` is used or not.

Option records have the following structure:

# a sequence representing the (short name) text that will follow the "-" option format. Use an atom if not relevant # a sequence representing the (long name) text that will follow the "--" option format. Use an atom if not relevant # a sequence, text that describes the option's purpose. Usually short as it is displayed when "-h"/"

See Also: show_help, command_line

Example 1:

```
sequence option_definition
integer gVerbose = 0
sequence gOutFile = {}
sequence gInFile = {}
function opt_verbose( sequence value)
   if value[OPT_VAL] = -1 then -- (-!v used on command line)
    gVerbose = 0
   else
     if value[OPT_CNT] = 1 then
        gVerbose = 1
     else
        gVerbose += 1
     end if
   end if
 return 1
end function

function opt_output_filename( sequence value)
   gOutFile = value[OPT_VAL]
 return 1
end function

function extras( sequence value)
   if not file_exists(value[OPT_VAL]) then
       show_help(option_definition, sprintf("Cannot find '%s'",
                 {value[OPT_VAL]}) )
       abort(1)
   end if
   gInFile = append(gInFile, value[OPT_VAL])
 return 1
end function

option_definition = {
{ "v", "verbose", "Verbose output",
     { NO_PARAMETER }, routine_id("opt_verbose") },
{ "h", "hash",     "Calc hash values",
     { NO_PARAMETER }, -1 },
{ "o", "output",  "Output filename",
     { MANDATORY, HAS_PARAMETER, ONCE } ,
     routine_id("opt_output_filename") },
{ "i", "import",  "An import path", { HAS_PARAMETER, MULTIPLE}, -1 },
{ "e", "version", "Display version", { VERSIONING, "myprog v1.0" } },
{  0, 0, 0, 0, routine_id("extras")}
}

map:map opts = cmd_parse(option_definition, NO_HELP)

-- When run as:
--     eui myprog.ex -v @output.txt -i /etc/app input1.txt input2.txt
--     and the file "output.txt" contains the two lines ...
--   -- output=john.txt
--   '-i /usr/local'
--
-- map:get(opts, "verbose") --> 1
-- map:get(opts, "hash") --> 0 (not supplied on command line)
-- map:get(opts, "output") --> "john.txt"
-- map:get(opts, "import") --> {"/usr/local", "/etc/app"}
-- map:get(opts, EXTRAS) --> {"input1.txt", "input2.txt"}
```

command_line

returns sequence of strings containing each word entered at the command-line that started your program.

Signature:

```
command_line()
```

```
<built-in> function
```

Returns: # The `path`, to either the Euphoria executable, (eui, eui.exe, euid.exe euiw.exe) or to your bound executable file. # The `next word`, is either the name of your Euphoria main file, or (again) the path to your bound executable file. # Any `extra words`, typed by the user. You can use these words in your program.

There are as many entries as words, plus the two mentioned above.

The Euphoria interpreter itself does not use any command-line options. You are free to use any options for your own program. It does have command line switches though.

The user can put quotes around a series of words to make them into a single argument.

If you convert your program into an executable file, either by binding it, or translating it to C, you will find that all command-line arguments remain the same, except for the first two, even though your user no longer types "eui" on the command-line (see examples below).

See Also: build_commandline, option_switches, getenv, cmd_parse, show_help

Example 1:

```
-- The user types:  eui myprog myfile.dat 12345 "the end"

cmd = command_line()

-- cmd will be:
      {
       "myprog",
       "myfile.dat",
       "12345",
       "the end"}
```

Example 2:

```
-- Your program is bound with the name "myprog.exe"
-- and is stored in the directory c:\myfiles
-- The user types:  myprog myfile.dat 12345 "the end"

cmd = command_line()

-- cmd will be:
       {

        "myfile.dat",
        "12345",
        "the end"
        }
```

```
-- Note that all arguments remain the same as example 1
-- except for the first two. The second argument is always
-- the same as the first and is inserted to keep the numbering
-- of the subsequent arguments the same, whether your program
-- is bound or translated as a .exe, or not.
```

option_switches

retrieves the list of switches passed to the interpreter on the command line.

Signature:

```
option_switches()
```

```
<built-in> function
```

Returns: A **sequence**, of strings, each containing a word related to switches.

Comments: All switches are recorded in upper case.

See Also: Command line switches

Example 1:

```
euiw -d helLo
-- will result in
-- option_switches() being {"-D","helLo"}
```

parse_commandline

parses a command line string breaking it into a sequence of command line

Signature:

```
parse_commandline(sequence cmdline)
```

```
public function
include cmdline.e
namespace cmdline
```

Arguments: ≡ `cmdline` : Command line sequence (string)

Returns: A **sequence**, of command line options

See Also: build_commandline

Example 1:

```
sequence opts = parse_commandline("-v -f '%Y-%m-%d %H:%M'")
-- opts = { "-v", "-f", "%Y-%m-%d %H:%M" }
```

show_help

shows the help message for the given command options.

Signature:

```
show_help(sequence opts, object add_help_rid = - 1,
sequence cmds = command_line(), object parse_options = {})
```

```
public procedure
include cmdline.e
namespace cmdline
```

Arguments: ≡ `opts` : a sequence of options. See the cmd_parse for details.
≡ `add_help_rid` : an object. Either a routine_id or a set of text strings. The default is -1 meaning that no additional help text will be used.

≡ `cmds` : a sequence of strings. By default this is the output from command_line()
≡ `parse_options` : An option set of behavior modifiers. See the cmd_parse for details.

Comments:

• `opts` is identical to the one used by cmd_parse
• `add_help_rid` can be used to provide additional help text. By default, just the option switches and their descriptions will be displayed. However you can provide additional text by either supplying a routine_id of a procedure that accepts no parameters; this procedure is expected to write text to the stdout device. Or you can supply one or more lines of text that will be displayed.

Example 1:

```
-- in myfile.ex
constant description = {
        "Creates a file containing an analysis of the weather.",
        "The analysis includes temperature and rainfall data",
        "for the past week."
    }

show_help({
   {"q","silent","Suppresses any output to console",NO_PARAMETER,-1},
   {"r", 0, "Sets how many lines the console should display",
   {HAS_PARAMETER,"lines"}, -1}}, description)

--> Outputs:

-- myfile.ex options:
--   -q, --silent      Suppresses any output to console
--   -r lines          Sets how many lines the console should display

-- Creates a file containing an analysis of the weather.
-- The analysis includes temperature and rainfall data
-- for the past week.
```

Example 2:

```
-- in myfile.ex
constant description = {
        "Creates a file containing an analysis of the weather.",
        "The analysis includes temperature and rainfall data",
        "for the past week."
    }
procedure sh()
  for i = 1 to length(description) do
     printf(1, " >> %s <<\n", {description[i]})
  end for
end procedure

show_help({
   {"q","silent","Suppresses any output to console",NO_PARAMETER,-1},
   {"r", 0, "Sets how many lines the console should display",
   {HAS_PARAMETER,"lines"}, -1}}, routine_id("sh"))
```

Outputs:

```
myfile.ex options:
-q, --silent      Suppresses any output to console
-r lines          Sets how many lines the console should display

>> Creates a file containing an analysis of the weather. <<
>> The analysis includes temperature and rainfall data  <<
>> for the past week.  <<
```

console

Information

Key Code names. These are the names of the index values for each of the 256 key code values.

See Also:

key_codes

Cursor Style Constants

In the cursor constants below, the second and fourth hex digits (from the left) determine the top and bottom row of pixels in the cursor. The first digit controls whether the cursor will be visible or not. For example: #0407 turns on the 4th through 7th rows.

See Also:

cursor

---

console API

BLOCK_CURSOR

Signature:

```
BLOCK_CURSOR
```

```
public constant
include console.e
namespace console
```

HALF_BLOCK_CURSOR

Signature:

```
HALF_BLOCK_CURSOR
```

```
public constant
include console.e
namespace console
```

KC_LBUTTON

Signature:

```
KC_LBUTTON
```

```
public constant
include console.e
namespace console
```

NO_CURSOR

Signature:

```
NO_CURSOR
```

```
public constant
include console.e
namespace console
```

THICK_UNDERLINE_CURSOR

Signature:

```
THICK_UNDERLINE_CURSOR
```

```
public constant
include console.e
namespace console
```

UNDERLINE_CURSOR

Signature:

```
UNDERLINE_CURSOR
```

```
public constant
include console.e
namespace console
```

allow_break

sets the response to Control+C and Control+Break key presses.

Signature:

```
allow_break(types :boolean b)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `b` : a boolean, TRUE (!= 0) to enable the trapping of Control-C or Control-Break; FALSE (0) to disable it.

Comments: When `b` is 1 (true), Control+C and Control+Break can terminate your program when it tries to read input from the keyboard. When `b` is 0 (false) your program will not be terminated by Control+C or Control+Break.

Initially your program can be terminated at any point where it tries to read from the keyboard.

You can find out if the user has pressed Control-C or Control-Break by calling `check_break`.

See Also: check_break

Example 1:

```
allow_break(0)  -- don't let the user kill the program!
```

any_key

displays a prompt to the user and waits for any key.

Signature:

```
any_key(sequence prompt = "Press Any Key to continue...", integer con = 1)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `prompt` : Prompt to display, defaults to "Press Any Key to continue..."
≡ `con` : Either 1 (stdout), or 2 (stderr). Defaults to 1.

Comments: This wraps wait_key by giving a clue that the user should press a key, and perhaps do some other things as well.

See Also: wait_key

Example 1:

```
any_key() -- "Press Any Key to continue..."
```

Example 2:

```
any_key("Press Any Key to quit")
```

attr_to_colors

converts an attribute code to its foreground and background color components.

Signature:

```
attr_to_colors(integer attr_code)


public function
include console.e
namespace console
```

Arguments: ≡ `attr_code` : integer, an attribute code.

Returns: A sequence of two elements -- {fgcolor, bgcolor}

See Also: get_screen_char, colors_to_attr

Example 1:

```
 ? attr_to_colors(92) --> {12, 5}
```

check_break

returns the number of Control-C and Control-Break key presses.

Signature:

```
check_break()


public function
include console.e
namespace console
```

Returns: An **integer**, the number of times that Control+C or Control+Break have been pressed since the last call to `check_break` or since the beginning of the program if this is the first call.

Comments: This is useful after you have called allow_break(0) which prevents Control+C or Control+Break from terminating your program. You can use `check_break` to find out if the user has pressed one of these keys. You might then perform some action such as a graceful shutdown of your program.

Neither Control+C nor Control+Break will be returned as input characters when you read the keyboard. You can only detect them by calling `check_break`.

See Also: allow_break

Example 1:

```
 k = get_key()
 if check_break() then  -- ^C or ^Break was hit once or more
     temp = graphics_mode(-1)
     puts(STDOUT, "Shutting down...")
     save_all_user_data()
     abort(1)
 end if
```

clear_screen

clears the screen using the current background color (which may be set by bk_color).

Signature:

```
clear_screen()


<built-in> procedure
```

See Also: bk_color

colors_to_attr

converts a foreground and background color set to its attribute code format.

Signature:

```
colors_to_attr(object fgbg, integer bg = 0)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `fgbg` : Either a sequence of {fgcolor, bgcolor} or just an integer fgcolor.
≡ `bg` : An integer bgcolor. Only used when `fgbg` is an integer.

Returns: An integer attribute code.

See Also: get_screen_char, put_screen_char, attr_to_colors

Example 1:

```
? colors_to_attr({12, 5}) --> 92
? colors_to_attr(12, 5) --> 92
```

cursor

selects a style of cursor.

Signature:

```
cursor(integer style)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `style` : an integer defining the cursor shape.

Platform: *windows*

Comments: In pixel-graphics modes no cursor is displayed.

See Also: graphics_mode, text_rows

Example 1:

```
cursor(BLOCK_CURSOR)
```

display

displays the supplied data on the console screen at the current cursor position.

Signature:

```
display(object data_in, object args = 1, integer finalnl = - 918_273_645)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `data_in` : Any object.
≡ `args` : Optional arguments used to format the output. Default is 1.
≡ `finalnl` : Optional. Determines if a new line is output after the data. Default is to output a new line.

Comments:

- If `data_in` is an atom or integer, it is simply displayed.

• If `data_in` is a simple text string, then `args` can be used to produce a formatted output with `data_in` providing the *text:format* string and `args` being a sequence containing the data to be formatted.
…… ♦ If the last character of `data_in` is an underscore character then it is stripped off and `finalnl` is set to zero. Thus ensuring that a new line is **not** output.
…… ♦ The formatting codes expected in `data_in` are the ones used by *text:format*. It is not mandatory to use formatting codes, and if `data_in` does not contain any then it is simply displayed and anything in `args` is ignored.

• If `data_in` is a sequence containing floating-point numbers, sub-sequences or integers that are not characters, then `data_in` is forwarded on to the `pretty_print` to display.
…… ♦ If `args` is a non-empty sequence, it is assumed to contain the pretty_print formatting options.
…… ♦ if `args` is an atom or an empty sequence, the assumed `pretty_print` formatting options are assumed to be {2}.

After the data is displayed, the routine will normally output a New Line. If you want to avoid this, ensure that the last parameter is a zero. Or to put this another way, if the last parameter is zero then a New Line will **not** be output.

See Also: pretty_print , text:format

Example 1:

```
display("Some plain text")
        -- Displays this string on the console plus a new line.
display("Your answer:",0)
       -- Displays this string on the console without a new line.
display("cat")
display("Your answer:",,0)
        -- Displays this string on the console without a new line.
display("")
display("Your answer:_")
       -- Displays this string,
       -- except the '_', on the console without a new line.
display("dog")
display({"abc", 3.44554})
       -- Displays the contents of 'res' on the console.
display("The answer to [1] was [2]", {"'why'", 42})
       -- formats these with a new line.
display("",2)
display({51,362,71}, {1})

-- Output would be ...
--      Some plain text
--      Your answer:cat
--      Your answer:
--      Your answer:dog
--          {
--            "abc",
--            3.44554
--           }
--       The answer to 'why' was 42
--           ""
--           {51'3',362,71'G'}
```

display_text_image

display a text image in any text mode.

Signature:

```
display_text_image(text_point xy, sequence text)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `xy` : a pair of 1-based coordinates representing the point at which to start writing
≡ `text` : a list of sequences of alternated character and attribute.

Comments: This routine displays to the active text page, and only works in text modes.

You might use `save_text_image` and `display_text_image` in a text-mode graphical user interface, allowing pop-up dialog boxes and drop-down menus to appear and disappear without losing what was previously on the screen.

See Also: save_text_image, put_screen_char

Example 1:

```
clear_screen()
display_text_image({1,1}, {{'A', WHITE, 'B', GREEN},
                           {'C', RED+16*WHITE},
                           {'D', BLUE}})
-- displays:
--     AB
--     C
--     D
-- at the top left corner of the screen.
-- 'A' will be white with black (0) background color,
-- 'B' will be green on black,
-- 'C' will be red on white, and
-- 'D' will be blue on black.
```

free_console

frees (deletes) any console window associated with your program.

Signature:

```
free_console()
```

```
public procedure
include console.e
namespace console
```

Comments: Euphoria will create a console text window for your program the first time that your program prints something to the screen, reads something from the keyboard, or in some way needs a console. On *windows* this window will automatically disappear when your program terminates, but you can call free_console() to make it disappear sooner. On *unix* the text mode console is always there, but an xterm window will disappear after Euphoria issues a "Press Enter" prompt at the end of execution.

On *unix* `free_console` will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a *unix* terminal a call to `free_console` without any further printing to the screen or reading from the keyboard, will eliminate the "Press Enter" prompt that Euphoria normally issues at the end of execution.

After freeing the console window, you can create a new console window by printing something to the screen, or simply calling `clear_screen`, `position`, or any other routine that needs a console.

When you use the trace facility, or when your program has an error, Euphoria will automatically create a console window to display trace information, error messages etc.

There is a WINDOWS API routine, FreeConsole that does something similar to `free_console`. Use `free_console` instead, because it lets the interpreter know that there is no longer a console to write to or read from.

See Also: clear_screen

get_key

returns the key that was pressed by the user, without waiting. Special

Signature:

```
get_key()
```

```
<built-in> function
```

Returns: An **integer**, either -1 if no key waiting, or the code of the next key waiting in keyboard buffer.

Comments: The operating system can hold a small number of key-hits in its keyboard buffer. `get_key` will return the next one from the buffer, or -1 if the buffer is empty.

Run the `key.bat` program to see what key code is generated for each key on your keyboard.

See Also: wait_key

Example 1:

```
 integer n = get_key()
 if n=-1 then
     puts(1, "No key waiting.\n")
 end if
```

get_screen_char

gets the value and attribute of the character at a given screen location.

Signature:

```
get_screen_char(positive_atom line, positive_atom column,
integer fgbg = 0)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `line` : the 1-base line number of the location
≡ `column` : the 1-base column number of the location
≡ `fgbg` : an integer, if 0 (the default) you get an attribute_code returned otherwise you get a foreground and background color number returned.

Returns:

• If fgbg is zero then a **sequence** of *two* elements, `{character, attribute_code}` for the specified location.
• If fgbg is not zero then a **sequence** of *three* elements, `{characterfg_color, bg_color}`

Comments:

• This function inspects a single character on the *active page*.
• The attribute_code is an atom that contains the foreground and background color of the character, and possibly other operating-system dependant information describing the appearance of the character on the screen.
• With `get_screen_char` and `put_screen_char` you can save and restore a character on the screen along with its attribute_code.
• The `fg_color` and `bg_color` are integers in the range 0 to 15, which correspond to values in the following table.

Color Table:

color number	name

0	black
1	dark blue
2	green
3	cyan
4	crimson
5	purple
6	brown
7	light gray
8	dark gray
9	blue
10	bright green
11	light blue
12	red
13	magenta
14	yellow
15	white

See Also: put_screen_char, save_text_image

Example 1:

```
-- read character and attributes at top left corner
s = get_screen_char(1,1)
-- s could be {'A', 92}
-- store character and attributes at line 25, column 10
put_screen_char(25, 10, s)
```

Example 2:

```
-- read character and colors at line 25, column 10.
s = get_screen_char(25,10, 1)
-- s could be {'A', 12, 5}
```

has_console

determines if the process has a console (or terminal) window.

Signature:

```
has_console()
```

```
public function
include console.e
namespace console
```

Returns: 1 if there is more than one process attached to the current console, 0 if a console does not exist or only one process (EUPHORIA) is attached to the current console.

Notes: This function always returns 1 on *unix* systems.

Example 1:

```
include std/console.e

if has_console() then
    printf(1, "Hello Console!")
end if
```

key_codes

gets and sets the keyboard codes used internally by Euphoria.

Signature:

```
key_codes(object codes = 0)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `codes` : Either a sequence of exactly 256 integers or an atom (the default).

Returns: A **sequence** of the current 256 keyboard codes, prior to any changes that this function might make.

Comments: When `codes` is a atom then no change to the existing codes is made, otherwise the set of 256 integers in `codes` completely replaces the existing codes.

See Also: key_codes

Example 1:

```
include std/console.e
sequence kc
kc = key_codes()  -- Get existing set.
kc[KC_LEFT] = 263 -- Change the code for the left-arrow press.
key_codes(kc)     -- Set the new codes.
```

These are the names of the index values for each of the 256 key code values.

maybe_any_key

display a prompt to the user and waits for any key, but only if the user is

Signature:

```
maybe_any_key(sequence prompt = "Press Any Key to continue...",
integer con = 1)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `prompt` : Prompt to display, defaults to "Press Any Key to continue..."
≡ `con` : Either 1 (stdout), or 2 (stderr). Defaults to 1.

Comments: This wraps wait_key by giving a clue that the user should press a key, and perhaps do some other things as well.

See Also: wait_key

Example 1:

```
any_key() -- "Press Any Key to continue..."
```

Example 2:

```
any_key("Press Any Key to quit")
```

positive_int

Signature:

```
positive_int(object x)
```

```
public type
include console.e
namespace console
```

prompt_number

prompts the user to enter a number, and returns only a validated input.

Signature:

```
prompt_number(sequence prompt, sequence range)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `st` : is a string of text that will be displayed on the screen.
≡ `s` : is a sequence of two values {lower, upper} which determine the range of values that the user may enter; s can be empty, {}, if there are no restrictions.

Returns: An **atom**, in the assigned range which the user typed in.

Comments: As long as the user enters a number that is less than lower or greater than upper, the user will be prompted again.

If this routine is too simple for your needs, feel free to copy it and make your own more specialized version.

See Also: puts, prompt_string

Example 1:

```
 age = prompt_number("What is your age? ", {0, 150})
```

Example 2:

```
 t = prompt_number("Enter a temperature in Celcius:\n", {})
```

prompt_string

prompts the user to enter a string of text.

Signature:

```
prompt_string(sequence prompt)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `st` : is a string that will be displayed on the screen.

Returns: A **sequence**, the string that the user typed in, stripped of any new-line character.

Comments: If the user happens to type Control-Z (indicates end-of-file), "" will be returned.

See Also: prompt_number

Example 1:

```
 name = prompt_string("What is your name? ")
```

put_screen_char

stores and displays a sequence of characters with attributes at a given location.

Signature:

```
put_screen_char(positive_atom line, positive_atom column,
sequence char_attr)
```

```
public procedure
include console.e
namespace console
```

Arguments: ≡ `line` : the 1-based line at which to start writing
≡ `column` : the 1-based column at which to start writing
≡ `char_attr` : a sequence of alternated characters and attribute codes.

Comments: `char_attr` must be in the form `{character, attribute code, character, attribute code, ...}`.

See Also: get_screen_char, display_text_image

Example 1:

```
-- write AZ to the top left of the screen
-- (attributes are platform-dependent)
put_screen_char(1, 1, {'A', 152, 'Z', 131})
```

save_text_image

copies a rectangular block of text out of screen memory.

Signature:

```
save_text_image(text_point top_left, text_point bottom_right)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `top_left` : the coordinates, given as a pair, of the upper left corner of the area to save.
≡ `bottom_right` : the coordinates, given as a pair, of the lower right corner of the area to save.

Returns: A **sequence**, of {character, attribute, character, ...} lists.

Comments: The returned value is appropriately handled by `display_text_image`.

This routine reads from the active text page, and only works in text modes.

You might use this function in a text-mode graphical user interface to save a portion of the screen before displaying a drop-down menu, dialog box, alert box, and so on.

See Also: display_text_image, get_screen_char

Example 1:

```
-- Top 2 lines are: Hello and World
s = save_text_image({1,1}, {2,5})

-- s is something like: {"H-e-l-l-o-", "W-o-r-l-d-"}
```

set_keycodes

changes the default codes returned by the keyboard.

Signature:

```
set_keycodes(object kcfile)
```

```
public function
include console.e
namespace console
```

Arguments: ≡ `kcfile` : Either the name of a text file or the handle of an open (for reading) text file.

Returns: When this function completes without error, zero (0) is returned, otherwise an error code is returned:
- 0 means without error
- -1 means that the supplied file could not me loaded in to a map.

• -2 means that a new key value was not an integer
• -3 means that an unknown key name was found in the file.

Comments: The text file is expected to contain bindings for one or more keyboard codes.

The format of the files is a set of lines, one line per key binding, in the form `KEYNAME = NEWVALUE`. The *keyname* is the same as the constants but without the "`KC_`" prefix. The key bindings can be in any order.

See Also: key_codes

In the cursor constants below, the second and fourth hex digits (from the left) determine the top and bottom row of pixels in the cursor. The first digit controls whether the cursor will be visible or not. For example: #0407 turns on the 4th through 7th rows.

Example 1:

```
-- doskeys.txt file containing some key bindings
F1 = 260
F2 = 261
INSERT = 456



set_keycodes( "doskeys.txt" )
```

text_rows

sets the number of lines on a text-mode screen.

Signature:

```
text_rows(positive_int rows)


public function
include console.e
namespace console
```

Arguments: ≡ `rows` : an integer, the desired number of rows.

Returns: An **integer**, the actual number of text lines.

Platform: *windows*

Comments: Values of 25, 28, 43 and 50 lines are supported by most video cards.

See Also: graphics_mode, video_config

wait_key

waits for user to press a key, unless any is pending, and returns its key code.

Signature:

```
wait_key()


public function
include console.e
namespace console
```

Returns: An **integer**, which is a key code. If one is waiting in keyboard buffer, then return it. Otherwise, wait for one to come up.

See Also: get_key, getc

convert

Routines

convert API

atom_to_float32

converts an atom to a sequence of 4 bytes in IEEE 32-bit format.

Signature:

```
atom_to_float32(atom a)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `a` : the atom to convert:

Returns: A **sequence**, of 4 bytes, which can be poked in memory to represent `a`.

Comments: Euphoria atoms can have values which are 64-bit IEEE floating-point numbers, so you may lose precision when you convert to 32-bits (16 significant digits versus 7). The range of exponents is much larger in 64-bit format (10 to the 308, versus 10 to the 38), so some atoms may be too large or too small to represent in 32-bit format. In this case you will get one of the special 32-bit values: inf or -inf (infinity or -infinity). To avoid this, you can use atom_to_float64().

Integer values will also be converted to 32-bit floating-point format.

On modern computers, computations on 64 bit floats are no slower than on 32 bit floats. Internally, the PC stores them in 80 bit registers anyway. Euphoria does not support these so called long doubles. Not all C compilers do.

See Also: float32_to_atom, int_to_bytes, atom_to_float64

Example 1:

```
fn = open("numbers.dat", "wb")
puts(fn, atom_to_float32(157.82)) -- write 4 bytes to a file
```

atom_to_float64

converts an atom to a sequence of 8 bytes in IEEE 64-bit format.

Signature:

```
atom_to_float64(atom a)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `a` : the atom to convert.

Returns: A **sequence**, of 8 bytes, which can be poked in memory to represent `a`.

Comments: All Euphoria atoms have values which can be represented as 64-bit IEEE floating-point numbers, so you can convert any atom to 64-bit format without losing any precision.

Integer values will also be converted to 64-bit floating-point format.

See Also: float64_to_atom, int_to_bytes, atom_to_float32

Example 1:

```
fn = open("numbers.dat", "wb")
puts(fn, atom_to_float64(157.82)) -- write 8 bytes to a file
```

atom_to_float80

Signature:

```
atom_to_float80(atom a)
```

```
public function
include convert.e
namespace convert
```

bits_to_int

converts a sequence of bits to an atom that has no fractional part.

Signature:

```
bits_to_int(sequence bits)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `bits` : the sequence to convert.

Returns: A positive **atom**, whose machine representation was given by `bits`.

Comments: An element in `bits` can be any atom. If nonzero, it counts for 1, else for 0.

The first elements in `bits` represent the bits with the least weight in the returned value. Only the 52 last bits will matter, as the PC hardware cannot hold an integer with more digits than this.

If you print s the bits will appear in "reverse" order, but it is convenient to have increasing subscripts access bits of increasing significance.

See Also: bytes_to_int, int_to_bits, operations on sequences

Example 1:

```
a = bits_to_int({1,1,1,0,1})
-- a is 23 (binary 10111)
```

bytes_to_int

converts a sequence of at most 4 bytes into an atom.

Signature:

```
bytes_to_int(sequence s)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `s` : the sequence to convert

Returns: An **atom**, the value of the concatenated bytes of `s`.

Comments: This performs the reverse operation from int_to_bytes

An atom is being returned, because the converted value may be bigger than what can fit in an Euphoria integer.

See Also: bits_to_int, float64_to_atom, int_to_bytes, peek, peek4s, peek4u, poke4

Example 1:

```
atom int32

int32 = bytes_to_int({37,1,0,0})
-- int32 is 37 + 256*1 = 293
```

float32_to_atom

converts a sequence of 4 bytes in IEEE 32-bit format to an atom.

Signature:

```
float32_to_atom(sequence_4 ieee32)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `ieee32` : the sequence to convert:

Returns: An **atom**, the same value as the FPU would see by peeking `ieee64` from RAM.

Comments: Any 32-bit IEEE floating-point number can be converted to an atom.

See Also: float64_to_atom, bytes_to_int, atom_to_float32

Example 1:

```
f = repeat(0, 4)
fn = open("numbers.dat", "rb") -- read binary
f[1] = getc(fn)
f[2] = getc(fn)
f[3] = getc(fn)
f[4] = getc(fn)
a = float32_to_atom(f)
```

float64_to_atom

converts a sequence of 8 bytes in IEEE 64-bit format to an atom.

Signature:

```
float64_to_atom(sequence_8 ieee64)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `ieee64` : the sequence to convert:

Returns: An **atom**, the same value as the FPU would see by peeking `ieee64` from RAM.

Comments: Any 64-bit IEEE floating-point number can be converted to an atom.

See Also: float32_to_atom, bytes_to_int, atom_to_float64

Example 1:

```
 f = repeat(0, 8)
 fn = open("numbers.dat", "rb")  -- read binary
 for i = 1 to 8 do
     f[i] = getc(fn)
 end for
 a = float64_to_atom(f)
```

float80_to_atom

Signature:

```
float80_to_atom(sequence bytes)
```

```
public function
include convert.e
namespace convert
```

hex_text

converts a text representation of a hexadecimal number to an atom.

Signature:

```
hex_text(sequence text)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `text` : the text to convert.

Returns: An **atom**, the numeric equivalent to `text`

Comments:

- The text can optionally begin with '#' which is ignored.
- The text can have any number of underscores, all of which are ignored.
- The text can have one leading '-', indicating a negative number.
- The text can have any number of underscores, all of which are ignored.
- Any other characters in the text stops the parsing and returns the value thus far.

See Also: value

Example 1:

```
 atom h = hex_text("-#3_4FA.00E_1BD")
  -- h is now -13562.003444492816925
  atom h = hex_text("DEADBEEF")
  -- h is now 3735928559
```

int_to_bits

extracts the lower bits from an integer.

Signature:

```
int_to_bits(atom x, integer nbits = 32)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `x` : the atom to convert
≡ `nbits` : the number of bits requested. The default is 32.

Returns: A **sequence**, of length `nbits`, made of 1's and 0's.

Comments: `x` should have no fractional part. If it does, then the first "bit" will be an atom between 0 and 2.

The bits are returned lowest first.

For negative numbers the two's complement bit pattern is returned.

You can use subscripting, slicing, and/or/xor/not of entire sequences etc. to manipulate sequences of bits. Shifting of bits and rotating of bits are easy to perform.

See Also: bits_to_int, int_to_bytes, Relational operators, operations on sequences

Example 1:

```
s = int_to_bits(177, 8)
-- s is {1,0,0,0,1,1,0,1} -- "reverse" order
```

int_to_bytes

converts an atom that represents an integer to a sequence of 4 bytes.

Signature:

```
int_to_bytes(atom x, integer size = 4)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `x` : an atom, the value to convert.

Returns: A **sequence**, of 4 bytes, lowest significant byte first.

Comments: If the atom does not fit into a 32-bit integer, things may still work right:
• If there is a fractional part, the first element in the returned value will carry it. If you poke the sequence to RAM, that fraction will be discarded anyway.
• If `x` is simply too big, the first three bytes will still be correct, and the 4th element will be `floor(x/power(2,24))`. If this is not a byte sized integer, some truncation may occur, but usually no error.

The integer can be negative. Negative byte-values will be returned, but after poking them into memory you will have the correct (two's complement) representation for the 386+.

See Also: bytes_to_int, int_to_bits, atom_to_float64, poke4

Example 1:

```
s = int_to_bytes(999)
-- s is {231, 3, 0, 0}
```

Example 2:

```
s = int_to_bytes(-999)
-- s is {-231, -4, -1, -1}
```

set_decimal_mark

gets, and possibly sets, the decimal mark that to_number() uses.

Signature:

```
set_decimal_mark(integer new_mark)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `new_mark` : An integer: Either a comma (,), a period (.) or any other integer.

Returns: An **integer**, The current value, before `new_mark` changes it.

Comments:

• When `new_mark` is a *period* it will cause `to_number()` to interpret a dot `(.)` as the decimal point symbol. The pre-changed value is returned.
• When `new_mark` is a *comma* it will cause `to_number()` to interpret a comma `(,)` as the decimal point symbol. The pre-changed value is returned.
• Any other value does not change the current setting. Instead it just returns the current value.
• The initial value of the decimal marker is a period.

to_integer

converts an object into an integer.

Signature:

```
to_integer(object data_in, integer def_value = 0)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `data_in` : Any Euphoria object.
≡ `def_value` : An integer. This is returned if `data_in` cannot be converted into an integer. If omitted, zero is returned.

Returns: An **integer**, either the integer rendition of `data_in` or `def_value` if it has no integer value.

Comments: The returned value is guaranteed to be a valid Euphoria integer.

Example 1:

```
? to_integer(12)             --> 12
? to_integer(12.4)           --> 12
? to_integer("12")           --> 12
? to_integer("12.9")         --> 12

? to_integer("a12")          --> 0 (not a valid number)
? to_integer("a12",-1)       --> -1 (not a valid number)
? to_integer({"12"})         --> 0 (sub-sequence found)
? to_integer(#3FFFFFFF)      --> 1073741823
? to_integer(#3FFFFFFF + 1) --> 0 (too big for a Euphoria integer)
```

to_number

converts the text into a number.

Signature:

```
to_number(sequence text_in, integer return_bad_pos = 0)
```

```
public function
include convert.e
namespace convert
```

Arguments: ≡ `text_in` : A string containing the text representation of a number.
≡ `return_bad_pos` : An integer.
…… ♦ If 0 (the default) then this will return a number based on the supplied text and it will **not** return any position in `text_in` that caused an incomplete conversion.
…… ♦ If `return_bad_pos` is -1 then if the conversion of `text_in` was complete the resulting number is returned otherwise a single-element sequence containing the position within `text_in` where the conversion stopped.
…… ♦ If not 0 then this returns both the converted value up to the point of failure (if any) and the position in `text_in` that caused the failure. If that position is 0 then there was no failure.

Returns:

• an **atom**, If `return_bad_pos` is zero, the number represented by `text_in`. If `text_in` contains invalid characters, zero is returned.

• a **sequence**, If `return_bad_pos` is non-zero. If `return_bad_pos` is -1 it returns a 1-element sequence containing the spot inside `text_in` where conversion stopped. Otherwise it returns a 2-element sequence containing the number represented by `text_in` and either 0 or the position in `text_in` where conversion stopped.

Comments: # You can supply **Hexadecimal** values if the value is preceded by a '#' character, **Octal** values if the value is preceded by a '@' character, and **Binary** values if the value is preceded by a '!' character. With hexadecimal values, the case of the digits 'A' - 'F' is not important. Also, any decimal marker embedded in the number is used with the correct base.

# Any underscore characters or thousands separators, that are embedded in the text number are ignored. These can be used to help visual clarity for long numbers. The thousands separator is a ',' when the decimal mark is '.' (the default), or '.' if the decimal mark is ','. You inspect and set it using set_decimal_mark().

# You can supply a single leading or trailing sign. Either a minus (-) or plus (+).

# You can supply one or more trailing adjacent percentage signs. The first one causes the resulting value to be divided by 100, and each subsequent one divides the result by a further 10. Thus 3845% gives a value of (3845 / 100) ==> 38.45, and 3845%% gives a value of (3845 / 1000) ==> 3.845.

# You can have single currency symbol before the first digit or after the last digit. A currency symbol is any character of the string: "$����".

# You can have any number of whitespace characters before the first digit and after the last digit.

# The currency, sign and base symbols can appear in any order. Thus "$ -21.10" is the same as " -$21.10 ", which is also the same as "21.10$-", etc.

# This function can optionally return information about invalid numbers. If `return_bad_pos` is not zero, a two-element sequence is returned. The first element is the converted number value , and the second is the position in the text where conversion stopped. If no errors were found then the second element is zero.

# When converting floating point text numbers to atoms, you need to be aware that many numbers cannot be accurately converted to the exact value expected due to the limitations of the 64-bit IEEEE Floating point format.

Example 1:

```
object val
val = to_number("12.34")      ---> 12.34 -- No errors
val = to_number("12.34", 1)   ---> {12.34, 0} -- No errors
val = to_number("12.34", -1)  ---> 12.34 -- No errors
val = to_number("12.34a", 1)  ---> {12.34, 6} -- Error at position 6
val = to_number("12.34a", -1) ---> {6} -- Error at position 6
val = to_number("12.34a")     ---> 0 because its not a valid number

val = to_number("#f80c")        --> 63500
val = to_number("#f80c.7aa")    --> 63500.47900390625
val = to_number("@1703")        --> 963
val = to_number("!101101")      --> 45
val = to_number("12_583_891")   --> 12583891
val = to_number("12_583_891%")  --> 125838.91
val = to_number("12,583,891%%") --> 12583.891
```

to_string

converts an object into a text string.

Signature:

```
to_string(object data_in, integer string_quote = 0,
integer embed_string_quote = '"')
```

```
public function
include convert.e
namespace convert
```

Arguments:
≡ `data_in` : Any Euphoria object.
≡ `string_quote` : An integer. If not zero (the default) this will be used to enclose `data_in`, if it is already a string.
≡ `embed_string_quote` : An integer. This will be used to enclose any strings embedded inside `data_in`. The default is '"'

Returns: A **sequence**. This is the string repesentation of data_in.

Comments:

- The returned value is guaranteed to be a displayable text string.
- `string_quote` is only used if `data_in` is already a string. In this case, all occurances of `string_quote` already in `data_in` are prefixed with the '\' escape character, as are any preexisting escape characters. Then `string_quote` is added to both ends of `data_in`, resulting in a quoted string.
- `embed_string_quote` is only used if `data_in` is a sequence that contains strings. In this case, it is used as the enclosing quote for embedded strings.

Example 1:

```
include std/console.e
display(to_string(12))           --> 12
display(to_string("abc"))        --> abc
display(to_string("abc",'"'))    --> "abc"
display(to_string(`abc\"`,'"'))  --> "abc\\""
display(to_string({12,"abc",{4.5, -99}}))    --> {12, "abc", {4.5, -99}}
display(to_string({12,"abc",{4.5, -99}},,0)) --> {12, abc, {4.5, -99}}
```

---

datetime

Localized Variables

month_names
month_abbrs
day_names

Date and Time Type Accessors

These accessors can be used with the datetime type.

Intervals

These constant enums are to be used with the add and subtract routines.

---

datetime API

DATE

Date

Signature:

```
DATE
```

```
public enum
include datetime.e
namespace datetime
```

DAY

Day (1-31)

Signature:

```
DAY
```

```
public enum
include datetime.e
namespace datetime
```

DAYS

Days

Signature:

```
DAYS
```

```
public enum
include datetime.e
namespace datetime
```

HOUR

Hour (0-23)

Signature:

```
HOUR
```

```
public enum
include datetime.e
namespace datetime
```

HOURS

Hours

Signature:

```
HOURS
```

```
public enum
include datetime.e
namespace datetime
```

MINUTE

Minute (0-59)

Signature:

```
MINUTE
```

```
public enum
include datetime.e
namespace datetime
```

MINUTES

Minutes

Signature:

```
MINUTES
```

```
public enum
include datetime.e
namespace datetime
```

MONTH

Month (1-12)

Signature:

```
MONTH
```

```
public enum
include datetime.e
namespace datetime
```

MONTHS

Months

Signature:

```
MONTHS
```

```
public enum
include datetime.e
namespace datetime
```

SECOND

Second (0-59)

Signature:

```
SECOND
```

```
public enum
include datetime.e
namespace datetime
```

SECONDS

Signature: Seconds

```
SECONDS
```

```
public enum
include datetime.e
namespace datetime
```

WEEKS

Signature: Weeks

```
WEEKS
```

```
public enum
include datetime.e
namespace datetime
```

YEAR

Signature: Year (full year, for example: 2010, 1922)

```
YEAR
```

```
public enum
include datetime.e
namespace datetime
```

YEARS

Signature: Years

```
YEARS
```

```
public enum
include datetime.e
namespace datetime
```

add

Signature: adds a number of *intervals* to a datetime.

```
add(datetime dt, object qty, integer interval)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : the base datetime
≡ `qty` : the number of *intervals* to add. It should be positive.
≡ `interval` : which kind of interval to add.

Returns: A **sequence**, more precisely a **datetime** representing the new moment in time.

Comments: Please see Constants for Date and Time for a reference of valid intervals.

Do not confuse the item access constants such as YEAR, MONTH, DAY, etc... with the interval constants YEARS, MONTHS, DAYS, and so on.

When adding MONTHS, it is a calendar based addition. For instance, a date of 5/2/2008 with 5 MONTHS added will become 10/2/2008. MONTHS does not compute the number of days per each month and the average number of days per month.

When adding YEARS, leap year is taken into account. Adding 4 YEARS to a date may result in a different day of month number due to leap year.

See Also: subtract, diff

Example 1:

```
d2 = add(d1, 35, SECONDS) -- add 35 seconds to d1
d2 = add(d1, 7, WEEKS)    -- add 7 weeks to d1
d2 = add(d1, 19, YEARS)   -- add 19 years to d1
```

ampm

AM/PM

Signature:

```
ampm
```

```
public sequence
include datetime.e
namespace datetime
```

date

return a sequence with information on the current date.

Signature:

```
date()
```

```
<built-in> function
```

Returns: A **sequence** of length 8, laid out as follows: # year -- since 1900 # month -- January = 1 # day -- day of month, starting at 1 # hour -- 0 to 23 # minute -- 0 to 59 # second -- 0 to 59 # day of the week -- Sunday = 1 # day of the year -- January 1st = 1

Comments: The value returned for the year is actually the number of years since 1900 (not the last 2 digits of the year). In the year 2000 this value was 100. In 2001 it was 101, etc.

See Also: time, now

Example 1:

```
now = date()
-- now has: {95,3,24,23,47,38,6,83}
-- i.e. Friday March 24, 1995 at 11:47:38pm, day 83 of the year
```

datetime

is the datetime type.

Signature:

```
datetime(object o)
```

```
public type
include datetime.e
namespace datetime
```

Arguments: ≡ `obj` : any object, so no crash takes place.

Comments: A datetime type consists of a sequence of length 6 in the form `{year, month, day_of_month, hour, minute, second}`. Checks are made to guarantee those values are in range.

day_abbrs

Abbreviations of day names

Signature:

```
day_abbrs
```

```
public sequence
include datetime.e
namespace datetime
```

day_names

Names of the days

Signature:

```
day_names
```

```
public sequence
include datetime.e
namespace datetime
```

days_in_month

returns the number of days in the month of `dt`.

Signature:

```
days_in_month(datetime dt)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

Comments: This takes into account leap years.

See Also: is_leap_year

Example 1:

```
d = new(2008, 1, 1, 0, 0, 0)
? days_in_month(d) -- 31
d = new(2008, 2, 1, 0, 0, 0) -- Leap year
? days_in_month(d) -- 29
```

days_in_year

returns the number of days in the year of `dt`.

Signature:

```
days_in_year(datetime dt)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

See Also: is_leap_year, days_in_month

Example 1:

```
 d = new(2007, 1, 1, 0, 0, 0)
 ? days_in_year(d) -- 365
 d = new(2008, 1, 1, 0, 0, 0) -- leap year
 ? days_in_year(d) -- 366
```

diff

computes the difference, in seconds, between two dates.

Signature:

```
diff(datetime dt1, datetime dt2)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt1` : the end datetime
≡ `dt2` : the start datetime

Returns: An **atom**, the number of seconds elapsed from `dt2` to `dt1`.

Comments: `dt2` is subtracted from `dt1`, therefore, you can come up with a negative value.

See Also: add, subtract

Example 1:

```
 d1 = now()
 sleep(15)  -- sleep for 15 seconds
 d2 = now()

 i = diff(d1, d2) -- i is 15
```

format

formats the date according to the format pattern string.

Signature:

```
format(datetime d, sequence pattern = "%Y-%m-%d %H:%M:%S")
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `d` : a datetime which is to be printed out
≡ `pattern` : a format string, similar to the ones sprintf() uses, but with some Unicode encoding. The default is "%Y-%m-%d %H:%M:%S".

Returns: A **string**, with the date `d` formatted according to the specification in `pattern`.

Comments: Pattern string can include the following specifiers:

- `%%` -- a literal %
- `%a` -- locale's abbreviated weekday name (e.g., Sun)
- `%A` -- locale's full weekday name (e.g., Sunday)
- `%b` -- locale's abbreviated month name (e.g., Jan)
- `%B` -- locale's full month name (e.g., January)
- `%C` -- century; like %Y, except omit last two digits (e.g., 21)
- `%d` -- day of month (e.g, 01)
- `%H` -- hour (00..23)
- `%I` -- hour (01..12)
- `%j` -- day of year (001..366)
- `%k` -- hour (0..23)
- `%l` -- hour (1..12)
- `%m` -- month (01..12)
- `%M` -- minute (00..59)
- `%p` -- locale's equivalent of either AM or PM; blank if not known
- `%P` -- like %p, but lower case
- `%s` -- seconds since 1970-01-01 00:00:00 UTC
- `%S` -- second (00..60)
- `%u` -- day of week (1..7); 1 is Monday
- `%w` -- day of week (0..6); 0 is Sunday
- `%y` -- last two digits of year (00..99)
- `%Y` -- year

See Also: to_unix, parse

Example 1:

```
d = new(2008, 5, 2, 12, 58, 32)
s = format(d, "%Y-%m-%d %H:%M:%S")
-- s is "2008-05-02 12:58:32"
```

Example 2:

```
d = new(2008, 5, 2, 12, 58, 32)
s = format(d, "%A, %B %d '%y %H:%M%p")
-- s is "Friday, May 2 '08 12:58PM"
```

from_date

converts a sequence formatted according to the built-in `date`() function to a valid datetime

Signature:

```
from_date(sequence src)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `src` : a sequence which date() might have returned

Returns: A **sequence**, more precisely a **datetime** corresponding to the same moment in time.

See Also: date, from_unix, now, new

Example 1:

```
d = from_date(date())
-- d is the current date and time
```

from_unix

creates a datetime value from the *unix* numeric format (seconds since EPOCH).

Signature:

```
from_unix(atom unix)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `unix` : an atom, counting seconds elapsed since EPOCH.

Returns: A **sequence**, more precisely a **datetime** representing the same moment in time.

See Also: to_unix, from_date, now, new

Example 1:

```
d = from_unix(0)
-- d is 1970-01-01 00:00:00  (zero seconds since EPOCH)
```

is_leap_year

tests if `dt` falls within leap year.

Signature:

```
is_leap_year(datetime dt)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

Returns: An **integer**, of 1 if leap year, otherwise 0.

See Also: days_in_month

Example 1:

```
d = new(2008, 1, 1, 0, 0, 0)
? is_leap_year(d) -- prints 1
d = new(2005, 1, 1, 0, 0, 0)
? is_leap_year(d) -- prints 0
```

month_abbrs

Abbreviations of month names

Signature:

```
month_abbrs
```

```
public sequence
include datetime.e
namespace datetime
```

month_names

Names of the months

Signature:

```
month_names
```

```
public sequence
include datetime.e
namespace datetime
```

new

creates a new datetime value.

Signature:

```
new(integer year = 0, integer month = 0, integer day = 0, integer hour = 0,
integer minute = 0, atom second = 0)
```

```
public function
include datetime.e
namespace datetime
```

Arguments:
- ≡ `year` -- the full year.
- ≡ `month` -- the month (1-12).
- ≡ `day` -- the day of the month (1-31).
- ≡ `hour` -- the hour (0-23) (defaults to 0)
- ≡ `minute` -- the minute (0-59) (defaults to 0)
- ≡ `second` -- the second (0-59) (defaults to 0)

See Also: from_date, from_unix, now, new_time

Example 1:

```
dt = new(2010, 1, 1, 0, 0, 0)
-- dt is Jan 1st, 2010
```

new_time

creates a new datetime value with a date of zeros.

Signature:

```
new_time(integer hour, integer minute, atom second)
```

```
public function
include datetime.e
namespace datetime
```

Arguments:
- ≡ `hour` : is the hour (0-23)
- ≡ `minute` : is the minute (0-59)
- ≡ `second` : is the second (0-59)

See Also: from_date, from_unix, now, new

Example 1:

```
dt = new_time(10, 30, 55)
dt is 10:30:55 AM
```

now

creates a new datetime value initialized with the current date and time.

Signature:

```
now()
```

```
public function
include datetime.e
namespace datetime
```

Returns: A **sequence**, more precisely a **datetime** corresponding to the current moment in time.

See Also: from_date, from_unix, new, new_time, now_gmt

Example 1:

```
dt = now()
-- dt is the current date and time
```

now_gmt

creates a new datetime value that falls into the Greenwich Mean Time (GMT) timezone.

Signature:

```
now_gmt()
```

```
public function
include datetime.e
namespace datetime
```

See Also: now

Example 1:

```
dt = now_gmt()
-- If local time was July 16th, 2008 at 10:34pm CST
-- dt would be July 17th, 2008 at 03:34pm GMT
```

parse

parses a datetime string according to the given format.

Signature:

```
parse(sequence val, sequence fmt = "%Y-%m-%d %H:%M:%S",
integer yylower = - 80)
```

```
public function
include datetime.e
namespace datetime
```

Arguments:
≡ `val` : string datetime value
≡ `fmt` : datetime format. Default is "%Y-%m-%d %H:%M:%S"
≡ `yysplit` : Set the maximum difference from the current year when parsing a two digit year. Defaults to -80/+20.

Returns: A **datetime**, value.

Comments: Only a subset of the format specification is currently supported:

- `%d` -- day of month (e.g, 01)
- `%H` -- hour (00..23)
- `%m` -- month (01..12)
- `%M` -- minute (00..59)
- `%S` -- second (00..60)
- `%y` -- 2-digit year (YY)
- `%Y` -- 4-digit year (CCYY)

More format codes will be added in future versions.

All non-format characters in the format string are ignored and are not matched against the input string.

All non-digits in the input string are ignored.

Parsing Two Digit Years:

When parsing a two digit year `parse` has to make a decision if a given year is in the past or future. For example, 10/18/44. Is that Oct 18, 1944 or Oct 18, 2044. A common rule has come about for this purpose and that is the -80/+20 rule. Based on research it was found that more historical events are recorded than future events, thus it favors history rather than future. Some other applications may require a different rule, thus the `yylower` parameter can be supplied.

Assuming today is 12/22/2010 here is an example of the -80/+20 rule.

```
| YY | Diff | CCYY |
| 18 | -92/+8 | 2018 |
| 95 | -15/+85 | 1995 |
| 33 | -77/+23 | 1933 |
| 29 | -81/+19 | 2029 |
```

Another rule in use is the -50/+50 rule. Therefore, if you supply -50 to the `yylower` to set the lower bounds, some examples may be (given that today is 12/22/2010).

```
| YY | Diff | CCYY |
| 18 | -92/+8 | 2018 |
| 95 | -15/+85 | 1995 |
| 33 | -77/+23 | 2033 |
| 29 | -81/+19 | 2029 |
```

See Also: format

Example 1:

```
datetime d = parse("05/01/2009 10:20:30", "%m/%d/%Y %H:%M:%S")
-- d is { 2009, 5, 1, 10, 20, 30 }
```

Example 2:

```
datetime d = parse("05/01/44", "%m/%d/%y", -50) -- -50/+50 rule
-- d is { 2044, 5, 14, 0, 0, 0 }
```

subtract

subtracts a number of *intervals* to a base datetime.

Signature:

```
subtract(datetime dt, atom qty, integer interval)


public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : the base datetime
≡ `qty` : the number of *intervals* to subtract. It should be positive.
≡ `interval` : which kind of interval to subtract.

Returns: A **sequence**, more precisely a **datetime** representing the new moment in time.

Comments: Please see Constants for Date/Time for a reference of valid intervals.

See the function `add`() for more information on adding and subtracting date intervals

See Also: add, diff

Example 1:

```
dt2 = subtract(dt1, 18, MINUTES) -- subtract 18 minutes from dt1
dt2 = subtract(dt1, 7, MONTHS)   -- subtract 7 months from dt1
dt2 = subtract(dt1, 12, HOURS)   -- subtract 12 hours from dt1
```

time

return the number of seconds since some fixed point in the past.

Signature:

```
time()
```

```
<built-in> function
```

Returns: An **atom**, which represents an absolute number of seconds.

Comments: Take the difference between two readings of `time`(), to measure, for example, how long a section of code takes to execute.

On some machines, `time`() can return a negative number. However, you can still use the difference in calls to `time`() to measure elapsed time.

See Also: date, now

Example 1:

```
constant ITERATIONS = 1000000
integer p
atom t0, loop_overhead

t0 = time()
for i = 1 to ITERATIONS do
    -- time an empty loop
end for
loop_overhead = time() - t0

t0 = time()
for i = 1 to ITERATIONS do
    p = power(2, 20)
end for
? (time() - t0 - loop_overhead)/ITERATIONS
-- calculates time (in seconds) for one call to power
```

to_unix

converts a datetime value to the *unix* numeric format (seconds since `EPOCH_1970`).

Signature:

```
to_unix(datetime dt)
```

```
public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

Returns: An **atom**, so this will not overflow during the winter 2038-2039.

See Also: from_unix, format

Example 1:

```
secs_since_epoch = to_unix(now())
-- secs_since_epoch is equal to the current seconds since epoch
```

weeks_day

gets the day of week of the datetime dt.

Signature:

```
weeks_day(datetime dt)


public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

Returns: An **integer**, between 1 (Sunday) and 7 (Saturday).

Example 1:

```
 d = new(2008, 5, 2, 0, 0, 0)
 day = weeks_day(d) -- day is 6 because May 2, 2008 is a Friday.
```

years_day

gets the Julian day of year of the supplied date.

Signature:

```
years_day(datetime dt)


public function
include datetime.e
namespace datetime
```

Arguments: ≡ `dt` : a datetime to be queried.

Returns: An **integer**, between 1 and 366.

Comments: For dates earlier than 1800, this routine may give inaccurate results if the date applies to a country other than United Kingdom or a former colony thereof. The change from Julian to Gregorian calendar took place much earlier in some other European countries.

Example 1:

```
 d = new(2008, 5, 2, 0, 0, 0)
 day = years_day(d) -- day is 123
```

dll

C Type Constants

- C_CHAR
- C_BYTE
- C_UCHAR
- C_UBYTE
- C_SHORT
- C_WORD
- C_USHORT
- C_INT
- C_BOOL
- C_UINT
- C_LONG
- C_ULONG
- C_SIZE_T
- C_POINTER

C Type Constants These C type constants are used when defining external C functions in a shared library file.

Example 1:

See define_c_proc

See Also:

define_c_proc, define_c_func, define_c_var

External Euphoria Type Constants These are used for arguments to and the return value from a Euphoria shared library file (.dll, .so or .dylib).

---

dll API

C_BOOL

bool 32-bits

Signature:

```
C_BOOL
```

```
public constant
include dll.e
namespace dll
```

C_BYTE

byte 8-bits

Signature:

```
C_BYTE
```

```
public constant
include dll.e
namespace dll
```

C_CHAR

char 8-bits

Signature:

```
C_CHAR
```

```
public constant
include dll.e
namespace dll
```

C_DOUBLE

double 64-bits

Signature:

```
C_DOUBLE
```

```
public constant
include dll.e
namespace dll
```

C_DWORD

dword 32-bits

Signature:

```
C_DWORD
```

```
public constant
include dll.e
namespace dll
```

C_DWORDLONG

dwordlong 64-bits

Signature:

```
C_DWORDLONG
```

```
public constant
include dll.e
namespace dll
```

C_FLOAT

float 32-bits

Signature:

```
C_FLOAT
```

```
public constant
include dll.e
namespace dll
```

C_HANDLE

handle sizeof pointer

Signature:

```
C_HANDLE
```

```
public constant
include dll.e
namespace dll
```

C_HRESULT

hresult 32-bits

Signature:

```
C_HRESULT
```

```
public constant
include dll.e
namespace dll
```

C_HWND

hwnd sizeof pointer

Signature:

```
C_HWND
```

```
public constant
include dll.e
namespace dll
```

C_INT

int 32-bits

Signature:

```
C_INT
```

```
public constant
include dll.e
namespace dll
```

C_LONG

long 32-bits except on 64-bit *nix, where it is 64-bits

Signature:

```
C_LONG
```

```
public constant
include dll.e
namespace dll
```

C_LONGLONG

longlong 64-bits

Signature:

```
C_LONGLONG
```

```
public constant
include dll.e
namespace dll
```

C_LPARAM

lparam sizeof pointer

Signature:

```
C_LPARAM
```

```
public constant
include dll.e
namespace dll
```

C_POINTER

any valid pointer

Signature:

```
C_POINTER
```

```
public constant
include dll.e
namespace dll
```

C_SHORT

short 16-bits

Signature:

```
C_SHORT
```

```
public constant
include dll.e
```

```
namespace dll
```

C_SIZE_T

size_t unsigned long 32-bits except on 64-bit *nix, where it is 64-bits

Signature:

```
C_SIZE_T
```

```
public constant
include dll.e
namespace dll
```

C_UBYTE

ubyte 8-bits

Signature:

```
C_UBYTE
```

```
public constant
include dll.e
namespace dll
```

C_UCHAR

unsigned char 8-bits

Signature:

```
C_UCHAR
```

```
public constant
include dll.e
namespace dll
```

C_UINT

unsigned int 32-bits

Signature:

```
C_UINT
```

```
public constant
include dll.e
namespace dll
```

C_ULONG

unsigned long 32-bits except on 64-bit *nix, where it is 64-bits

Signature:

```
C_ULONG
```

```
public constant
```

```
include dll.e
namespace dll
```

C_USHORT

unsigned short 16-bits

Signature:

```
C_USHORT
```

```
public constant
include dll.e
namespace dll
```

C_WORD

word 16-bits

Signature:

```
C_WORD
```

```
public constant
include dll.e
namespace dll
```

C_WPARAM

wparam sizeof pointer

Signature:

```
C_WPARAM
```

```
public constant
include dll.e
namespace dll
```

E_ATOM

atom

Signature:

```
E_ATOM
```

```
public constant
include dll.e
namespace dll
```

E_INTEGER

integer

Signature:

```
E_INTEGER
```

```
public constant
include dll.e
namespace dll
```

E_OBJECT

Signature: object

```
E_OBJECT
```

```
public constant
include dll.e
namespace dll
```

E_SEQUENCE

Signature: sequence

```
E_SEQUENCE
```

```
public constant
include dll.e
namespace dll
```

NULL

Signature: C's NULL pointer

```
NULL
```

```
public constant
include dll.e
namespace dll
```

c_func

Signature: calls a C function, or machine code function, or translated or compiled Euphoria function by routine id.

```
c_func(integer rid, sequence args={})
```

```
<built-in> function
```

Arguments: ≡ `rid` : an integer, the routine_id of the external function being called.
≡ `args` : a sequence, the list of parameters to pass to the function

Returns: An **object**, whose type and meaning was defined on calling define_c_func().

Comments: `rid` must have been returned by define_c_func(), **not** by routine_id(). The type checks are different, and you would get a machine level exception in the best case.

If the function does not take any arguments then `args` should be `{}`.

If you pass an argument value which contains a fractional part, where the C function expects a C integer type, the argument will be rounded towards 0. e.g. 5.9 will be passed as 5, -5.9 will be passed as -5.

The function could be part of a .dll or .so created by the Euphoria To C Translator. In this case, a Euphoria atom or sequence could be returned. C and machine code functions can only return integers, or more generally, atoms (IEEE floating-point numbers).

See Also: c_proc, define_c_proc, open_dll, Platform-Specific Issues

Example 1:

```
atom user32, hwnd, ps, hdc
integer BeginPaint

-- open user32.dll - it contains the BeginPaint C function
user32 = open_dll("user32.dll")

-- the C function BeginPaint takes a C int argument and
-- a C pointer, and returns a C int as a result:
BeginPaint = define_c_func(u""BeginPaint
                           {C_INT, C_POINTER}, C_INT)

-- call BeginPaint, passing hwnd and ps as the arguments,
-- hdc is assigned the result:
hdc = c_func(BeginPaint, {hwnd, ps})
```

c_proc

calls a C void function, machine code function, or translated/compiled Euphoria procedure by routine id.

Signature:

```
c_proc(integer rid, sequence args={})
```

```
<built-in> procedure
```

Arguments: ≡ `rid` : an integer, the routine_id of the external function being called.
≡ `args` : a sequence, the list of parameters to pass to the function

Comments: `rid` must have been returned by define_c_proc(), **not** by routine_id(). The type checks are different, and you would get a machine level exception in the best case.

If the procedure does not take any arguments then `args` should be `{}`.

If you pass an argument value which contains a fractional part, where the C void function expects a C integer type, the argument will be rounded towards 0. (for example: 5.9 will be passed as 5, -5.9 will be passed as -5).

See Also: c_func, define_c_func, open_dll, Platform-Specific Issues

Example 1:

```
atom user32, hwnd, rect
integer GetClientRect

-- open user32.dll - it contains the GetClientRect C function
user32 = open_dll("user32.dll")

-- GetClientRect is a VOID C function that takes a C int
-- and a C pointer as its arguments:
GetClientRect = define_c_proc(user32, "GetClientRect",
                              {C_INT, C_POINTER})

-- pass hwnd and rect as the arguments
c_proc(GetClientRect, {hwnd, rect})
```

call_back

gets a machine address for an Euphoria procedure.

Signature:

```
call_back(object id)
```

```
public function
include dll.e
namespace dll
```

Arguments: ≡ `id` : an object, either the id returned by routine_id for the function/procedure, or a pair {'+', id}.

Returns: An **atom**, the address of the machine code of the routine. It can be used by *windows* or an external C routine in a *windows* .dll or *unix* shared library (.so), as a 32-bit "call-back" address for calling your Euphoria routine.

Comments: By default, your routine will work with the stdcall convention. On Windows, you can specify its id as {'+', id}, in which case it will work with the cdecl calling convention instead. On non-Microsoft platforms, you should only use simple IDs, as there is just one standard calling convention, i.e. cdecl.

You can set up as many call-back functions as you like, but they must all be Euphoria functions (or types) with 0 to 9 arguments. If your routine has nothing to return (it should really be a procedure), just return 0 (say), and the calling C routine can ignore the result.

When your routine is called, the argument values will all be 32-bit unsigned (positive) values. You should declare each parameter of your routine as atom, unless you want to impose tighter checking. Your routine must return a 32-bit integer value.

You can also use a call-back address to specify a Euphoria routine as an exception handler in the *unix* signal() function. For example, you might want to catch the SIGTERM signal, and do a graceful shutdown. Some Web hosts send a SIGTERM to a CGI process that has used too much CPU time.

A call-back routine that uses the cdecl convention and returns a floating-point result, might not work with euiw. This is because the Watcom C compiler (used to build euiw) has a non-standard way of handling cdecl floating-point return values.

See Also: routine_id

Example 1:

define_c_func

defines the characteristics of either a C function, or a machine-code routine that returns

Signature:

```
define_c_func(object lib, object routine_name, sequence arg_types,
atom return_type)
```

```
public function
include dll.e
namespace dll
```

Arguments: ≡ `lib` : an object, either an entry point returned as an atom by open_dll(), or "" to denote a routine the RAM address is known.
≡ `routine_name` : an object, either the name of a procedure in a shared object or the machine address of the procedure.
≡ `argtypes` : a sequence of type constants.
≡ `return_type` : an atom, indicating what type the function will return.

Returns: A small **integer**, known as a routine id, will be returned.

Comments: Use the returned routine id as the first argument to c_proc() when you wish to call the routine from Euphoria.

A returned value of -1 indicates that the procedure could not be found or linked to.

On *windows* you can add a '+' character as a prefix to the function name. This indicates to Euphoria that the function uses the cdecl calling convention. By default, Euphoria assumes that C routines accept the stdcall convention.

When defining a machine code routine, x1 must be the empty sequence, "" or {}, and x2 indicates the address of the machine code routine. You can poke the bytes of machine code into a block of memory reserved using `allocate`(). On *windows* the machine code routine is normally expected to follow the stdcall calling convention, but if you wish to use the cdecl convention instead, you can code {'+', address} instead of address for x2.

The C function that you define could be one created by the Euphoria To C Translator, in which case you can pass Euphoria data to it, and receive Euphoria data back. A list of Euphoria types is contained in dll.e:

- E_INTEGER = #06000004
- E_ATOM = #07000004
- E_SEQUENCE= #08000004
- E_OBJECT = #09000004

You can pass or return any C integer type or pointer type. You can also pass a Euphoria atom as a C double or float, and get a C double or float returned to you as a Euphoria atom.

Parameter types which use 4 bytes or less are all passed the same way, so it is not necessary to be exact when choosing a 4-byte parameter type. However the distinction between signed and unsigned may be important when you specify the return type of a function.

Currently, there is no way to pass a C structure by value or get a C structure as a return result. You can only pass a pointer to a structure and get a pointer to a structure as a result. However, you can pass a 64 bit integer as two C_LONG instead. On calling the routine, pass low doubleword first, then high doubleword.

If you are not interested in using the value returned by the C function, you should instead define it with define_c_proc() and call it with c_proc().

If you use euiw to call a cdecl C routine that returns a floating-point value, it might not work. This is because the Watcom C compiler (used to build euiw) has a non-standard way of handling cdecl floating-point return values.

Passing floating-point values to a machine code routine will be faster if you use `c_func`() rather than `call`() to call the routine, since you will not have to use `atom_to_float64`() and `poke`() to get the floating-point values into memory.

See Also: `demo\callmach.ex`, c_func, define_c_proc, c_proc, open_dll

Example 1:

```
atom user32
integer LoadIcon

-- open user32.dll - it contains the LoadIconA C function
user32 = open_dll("user32.dll")

-- It takes a C pointer and a C int as parameters.
-- It returns a C int as a result.
LoadIcon = define_c_func(u""LoadIconA
                                {C_POINTER, C_INT}, C_INT)
-- We use "LoadIconA" here because we know that LoadIconA
-- needs the stdcall convention, as do
-- all standard .dll routines in the WINDOWS API.
-- To specify the cdecl convention, we would have used "+LoadIconA".

if LoadIcon = -1 then
    puts(1, "LoadIconA could not be found!\n")
end if
```

define_c_proc

defines the characteristics of either a C function or a machine-code routine, that you

Signature:

```
define_c_proc(object lib, object routine_name, sequence arg_types)
```

```
public function
include dll.e
namespace dll
```

Arguments:
≡ `lib` : an object, either an entry point returned as an atom by open_dll(), or "" to denote a routine the RAM address is known.
≡ `routine_name` : an object, either the name of a procedure in a shared object or the machine address of the procedure.
≡ `argtypes` : a sequence of type constants.

Returns: A small **integer**, known as a routine id, will be returned.

Comments: Use the returned routine id as the first argument to c_proc() when you wish to call the routine from Euphoria.

A returned value of -1 indicates that the procedure could not be found or linked to.

On *windows* you can add a '+' character as a prefix to the procedure name. This tells Euphoria that the function uses the cdecl calling convention. By default, Euphoria assumes that C routines accept the stdcall convention.

When defining a machine code routine, `lib` must be the empty sequence, "" or {}, and `routine_name` indicates the address of the machine code routine. You can poke the bytes of machine code into a block of memory reserved using allocate(). On *windows* the machine code routine is normally expected to follow the stdcall calling convention, but if you wish to use the cdecl convention instead, you can code {'+', address} instead of address.

`argtypes` is made of type constants, which describe the C types of arguments to the procedure. They may be used to define machine code parameters as well.

The C function that you define could be one created by the Euphoria To C Translator, in which case you can pass Euphoria data to it, and receive Euphoria data back. A list of Euphoria types is shown above.

You can pass any C integer type or pointer type. You can also pass a Euphoria atom as a C double or float.

Parameter types which use 4 bytes or less are all passed the same way, so it is not necessary to be exact.

Currently, there is no way to pass a C structure by value. You can only pass a pointer to a structure. However, you can pass a 64 bit integer by pretending to pass two C_LONG instead. When calling the routine, pass low doubleword first, then high doubleword.

The C function can return a value but it will be ignored. If you want to use the value returned by the C function, you must instead define it with define_c_func() and call it with c_func().

See Also: c_proc, define_c_func, c_func, open_dll

Example 1:

```
atom user32
integer ShowWindow

-- open user32.dll - it contains the ShowWindow C function
user32 = open_dll("user32.dll")

-- It has 2 parameters that are both C int.
ShowWindow = define_c_proc(user32, "ShowWindow", {C_INT, C_INT})
-- If ShowWindow used the cdecl convention,
-- we would have coded "+ShowWindow" here

if ShowWindow = -1 then
    puts(1, "ShowWindow not found!\n")
end if
```

define_c_var

gets the address of a symbol in a shared library or in RAM.

Signature:

```
define_c_var(atom lib, sequence variable_name)
```

```
public function
include dll.e
namespace dll
```

Arguments: ≡ `lib` : an atom, the address of a Linux or FreeBSD shared library, or Windows .dll, as returned by open_dll().
≡ `variable_name` : a sequence, the name of a public C variable defined within the library.

Returns: An **atom**, the memory address of `variable_name`.

Comments: Once you have the address of a C variable, and you know its type, you can use `peek`() and `poke`() to read or write the value of the variable. You can in the same way obtain the address of a C function and pass it to any external routine that requires a callback address.

See Also: c_proc, define_c_func, c_func, open_dll

Example 1:

open_dll

opens a *windows* dynamic link library (.dll) file, or a *unix* shared library

Signature:

```
open_dll(sequence file_name)
```

```
public function
include dll.e
namespace dll
```

Arguments: ≡ `file_name` : a sequence, the name of a shared libray file or a sequence of names of shared library files to be opened.

Returns: An **atom**, actually a 32-bit address. 0 is returned if the .dll can not be found.

Comments: `file_name` can be a relative or an absolute file name. Most operating systems will use the normal search path for locating non-relative files.

`file_name` can be a list of file names to try. On different Linux platforms especially, the filename will not always be the same. For instance, you may wish to try opening libmylib.so, libmylib.so.1, libmylib.so.1.0, libmylib.so.1.0.0. If given a sequence of file names to try, the first successful library loaded will be returned. If no library could be loaded, 0 will be returned after exhausting the entire list of file names.

The value returned by `open_dll`() can be passed to `define_c_proc`(), `define_c_func`(), or `define_c_var`().

You can open the same .dll or .so file multiple times. No extra memory is used and you will get the same number returned each time.

Euphoria will close the .dll or .so for you automatically at the end of execution.

See Also: define_c_func, define_c_proc, define_c_var, c_func, c_proc

Example 1:

```
atom user32
user32 = open_dll("user32.dll")
if user32 = 0 then
   puts(1, "Couldn't open user32.dll!\n")
end if
```

Example 2:

```
atom mysql_lib
mysql_lib = open_dll({"libmysqlclient.so", "libmysqlclient.so.15",
                     "libmysqlclient.so.15.0"})
if mysql_lib = 0 then
  puts(1, "Couldn't find the mysql client library\n")
end if
```

---

eds

Error Status Constants

- DB_OK
- DB_OPEN_FAIL
- DB_EXISTS_ALREADY
- DB_LOCK_FAIL
- DB_BAD_NAME
- DB_FATAL_FAIL

Lock Type Constants

- DB_LOCK_NO
- DB_LOCK_SHARED
- DB_LOCK_EXCLUSIVE
- DB_LOCK_READ_ONLY

Error Code Constants

- MISSING_END
- NO_DATABASE
- BAD_SEEK
- NO_TABLE
- DUP_TABLE
- BAD_RECNO
- INSERT_FAILED
- LAST_ERROR_CODE
- BAD_FILE

Indexes for connection option structure.

---

eds API

BAD_FILE

bad file

Signature:

```
BAD_FILE
```

```
public enum
include eds.e
namespace eds
```

BAD_RECNO

unknown key_location index was supplied.

Signature:

```
BAD_RECNO
```

```
public enum
include eds.e
namespace eds
```

BAD_SEEK

io:seek() failed.

Signature:

```
BAD_SEEK
```

```
public enum
include eds.e
namespace eds
```

CONNECTION

Fetch the details about the alias

Signature:

```
CONNECTION
```

```
public constant
include eds.e
namespace eds
```

CONNECT_FREE

Initial number of free pointers to create

Signature:

```
CONNECT_FREE
```

```
public enum
include eds.e
namespace eds
```

CONNECT_LOCK

Locking method

Signature:

```
CONNECT_LOCK
```

```
public enum
include eds.e
namespace eds
```

CONNECT_TABLES

Initial number of tables to create

Signature:

```
CONNECT_TABLES
```

```
public enum
include eds.e
namespace eds
```

DB_BAD_NAME

An invalid name suppled when creating a table.

Signature:

```
DB_BAD_NAME
```

```
public enum
include eds.e
namespace eds
```

DB_EXISTS_ALREADY

The database could not be created, it already exists.

Signature:

```
DB_EXISTS_ALREADY
```

```
public enum
include eds.e
namespace eds
```

DB_FATAL_FAIL

A fatal error has occurred.

Signature:

```
DB_FATAL_FAIL
```

```
public enum
include eds.e
namespace eds
```

DB_LOCK_EXCLUSIVE

Open the database with read and write access.

Signature:

```
DB_LOCK_EXCLUSIVE
```

```
public enum
include eds.e
namespace eds
```

DB_LOCK_FAIL

A lock could not be gained on the database.

Signature:

```
DB_LOCK_FAIL
```

```
public enum
include eds.e
namespace eds
```

DB_LOCK_NO

Do not lock the file.

Signature:

```
DB_LOCK_NO
```

```
public enum
include eds.e
namespace eds
```

DB_LOCK_READ_ONLY

Open the database with read-only access and ignore others updating it

Signature:

```
DB_LOCK_READ_ONLY
```

```
public enum
include eds.e
namespace eds
```

DB_LOCK_SHARED

Open the database with read-only access but allow others to update it.

Signature:

```
DB_LOCK_SHARED
```

```
public enum
include eds.e
namespace eds
```

DB_OK

Database is OK, not error has occurred.

Signature:

```
DB_OK
```

```
public enum
include eds.e
namespace eds
```

DB_OPEN_FAIL

The database could not be opened.

Signature:

```
DB_OPEN_FAIL
```

```
public enum
include eds.e
namespace eds
```

DISCONNECT

Disconnect a connected database

Signature:

```
DISCONNECT
```

```
public constant
include eds.e
namespace eds
```

DUP_TABLE

this table already exists.

Signature:

```
DUP_TABLE
```

```
public enum
include eds.e
namespace eds
```

INIT_FREE

The initial number of free space pointers to reserve space for when creating a database.

Signature:

```
INIT_FREE
```

```
public constant
include eds.e
namespace eds
```

INIT_TABLES

The initial number of tables to reserve space for when creating a database.

Signature:

```
INIT_TABLES
```

```
public constant
include eds.e
namespace eds
```

INSERT_FAILED

could not insert a new record.

Signature:

```
INSERT_FAILED
```

```
public enum
include eds.e
namespace eds
```

LAST_ERROR_CODE

last error code

Signature:

```
LAST_ERROR_CODE
```

```
public enum
include eds.e
namespace eds
```

LOCK_METHOD

Locking method to use

Signature:

```
LOCK_METHOD
```

```
public constant
include eds.e
namespace eds
```

MISSING_END

Missing 0 terminator

Signature:

```
MISSING_END
```

```
public enum
include eds.e
```

```
namespace eds
```

NO_DATABASE

current_db is not set

Signature:

```
NO_DATABASE
```

```
public enum
include eds.e
namespace eds
```

NO_TABLE

no table was found.

Signature:

```
NO_TABLE
```

```
public enum
include eds.e
namespace eds
```

check_free_list

detects corruption of the free list in a Euphoria database.

Signature:

```
check_free_list()
```

```
public procedure
include eds.e
namespace eds
```

Comments: This is a debug routine used by RDS to detect corruption of the free list. Users do not normally call this.

db_cache_clear

forces the database index cache to be cleared.

Signature:

```
db_cache_clear()
```

```
public procedure
include eds.e
namespace eds
```

Comments:

- This is not normally required to the run. You might run it to set up a predetermined state for performance timing, or to release some memory back to the application.

Example 1:

```
 db_cache_clear() -- Clear the cache.
```

db_clear_table

clears a table of all its records, in the current database.

Signature:

```
db_clear_table(sequence name, integer init_records = DEF_INIT_RECORDS)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `name` : a sequence, the name of the table to clear.

Comments: If there is no table with the name given by name, then nothing happens. On success, all records are deleted and all space used by the table is freed up. If this is the current table, after this operation it will still be the current table.

See Also: db_table_list, db_select_table, db_delete_table

db_close

unlocks and closes the current database.

Signature:

```
db_close()
```

```
public procedure
include eds.e
namespace eds
```

Comments: Call this procedure when you are finished with the current database. Any lock will be removed, allowing other processes to access the database file. The current database becomes undefined.

db_compress

compresses the current database.

Signature:

```
db_compress()
```

```
public function
include eds.e
namespace eds
```

Returns: An **integer**, either DB_OK on success or an error code on failure.

Comments: The current database is copied to a new file such that any blocks of unused space are eliminated. If successful, the return value will be set to `DB_OK`, and the new compressed database file will retain the same name. The current table will be undefined. As a backup, the original, uncompressed file will be renamed with an extension of .t0 (or .t1, .t2, ..., .t99). In the highly unusual case that the compression is unsuccessful, the database will be left unchanged, and no backup will be made.

When you delete items from a database, you create blocks of free space within the database file. The system keeps track of these blocks and tries to use them for storing new data that you insert. db_compress() will copy the current database without copying these free areas. The size of the database file may therefore be reduced. If the backup filenames reach .t99 you will have to delete some of them.

Example 1:

```
if db_compress() != DB_OK then
    puts(2, "compress failed!\n")
end if
```

db_connect

defines a symbolic name for a database, and its default attributes.

Signature:

```
db_connect(sequence dbalias, sequence path = "", sequence dboptions = {})
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `dbalias` : a sequence. This is the symbolic name that the database can be referred to by.
≡ `path` : a sequence, the path to the file that will contain the database.
≡ `dboptions`: a sequence. Contains the set of attributes for the database. The default is {} meaning it will use the various EDS default values.

Returns: An **integer**, status code, either DB_OK if creation successful or anything else on an error.

Comments:

• This does not create or open a database. It only associates a symbolic name with a database path. This name can then be used in the calls to db_create(), db_open(), and db_select() instead of the physical database name.
• If the file in the path does not have an extention, ".edb" will be added automatically.
• The `dboptions` can contain any of the options detailed below. These can be given as a single string of the form `"option=value, option=value, ..."` or as as sequence containing option-value pairs, `{ {option,value}, {option,value}, ... }`

See Also: db_create, db_open, db_select

Example 1:

```
db_connect("myDB", "/usr/data/myapp/customer.edb", {{LOCK_METHOD,DB_LOCK_NO},
                                                     {INIT_TABLES,1}})
db_open("myDB")
```

Example 2:

```
db_connect("myDB", "/usr/data/myapp/customer.edb",
                           sprintf("init_tables=1,lock_method=%d",DB_LOCK_NO))
db_open("myDB")
```

Example 3:

```
db_connect("myDB", "/usr/data/myapp/customer.edb",
                           sprintf("init_tables=1,lock_method=%d",DB_LOCK_NO))
db_connect("myDB",,CONNECTION) --> {"/usr/data/myapp/customer.edb", {0,1,1}}
db_connect("myDB",,DISCONNECT) -- The name 'myDB' is removed from EDS.
```

db_create

creates a new database, given a file path and a lock method.

Signature:

```
db_create(sequence path, integer lock_method = DB_LOCK_NO,
integer init_tables = DEF_INIT_TABLES, integer init_free = DEF_INIT_FREE)
```

```
public function
include eds.e
```

```
namespace eds
```

Arguments: ≡ `path` : a sequence, the path to the file that will contain the database.
≡ `lock_method` : an integer specifying which type of access can be granted to the database. The value of `lock_method` can be either `DB_LOCK_NO` (no lock) or `DB_LOCK_EXCLUSIVE` (exclusive lock).
≡ `init_tables` : an integer giving the initial number of tables to reserve space for. The default is 5 and the minimum is 1.
≡ `init_free` : an integer giving the initial amount of free space pointers to reserve space for. The default is 5 and the minimum is 0.

Returns: An **integer**, status code, either DB_OK if creation successful or anything else on an error.

Comments: On success, the newly created database becomes the **current database** to which all other database operations will apply.

If the file in the path does not have an extention, ".edb" will be added automatically.

A version number is stored in the database file so future versions of the database software can recognize the format, and possibly read it and deal with it in some way.

If the database already exists, it will not be overwritten. db_create() will return DB_EXISTS_ALREADY.

See Also: db_open, db_select

Example 1:

```
if db_create("mydata", DB_LOCK_NO) != DB_OK then
    puts(2, "Couldn't create the database!\n")
    abort(1)
end if
```

db_create_table

creates a new table within the current database.

Signature:

```
db_create_table(sequence name, integer init_records = DEF_INIT_RECORDS)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `name` : a sequence, the name of the new table.
≡ `init_records` : The number of records to initially reserve space for. (Default is 50)

Returns: An **integer**, either DB_OK on success or DB_EXISTS_ALREADY on failure.

Comments:

• The supplied name must not exist already on the current database.
• The table that you create will initially have 0 records. However it will reserve some space for a number of records, which will improve the initial data load for the table.
• It becomes the current table.

See Also: db_select_table, db_table_list

Example 1:

```
if db_create_table("my_new_table") != DB_OK then
    puts(2, "Could not create my_new_table!\n")
end if
```

db_current

gets the name of the currently selected database.

Signature:

```
db_current()


public function
include eds.e
namespace eds
```

Returns: A **sequence**, the name of the current database. An empty string means that no database is currently selected.

Comments: The actual name returned is the *path* as supplied to the db_open routine.

See Also: db_select

Example 1:

```
 s = db_current_database()
```

db_current_table

gets the name of the currently selected table.

Signature:

```
db_current_table()


public function
include eds.e
namespace eds
```

Arguments: # None.

Returns: A **sequence**, the name of the current table. An empty string means that no table is currently selected.

See Also: db_select_table, db_table_list

Example 1:

```
 s = db_current_table()
```

db_delete_record

deletes a record number key_location from the current table.

Signature:

```
db_delete_record(integer key_location,
object table_name = current_table_name)


public procedure
include eds.e
namespace eds
```

See Also: db_find_key

Example 1:

```
 db_delete_record(55)
```

db_delete_table

deletes a table in the current database.

Signature:

```
db_delete_table(sequence name)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `name` : a sequence, the name of the table to delete.

Comments: If there is no table with the name given by name, then nothing happens. On success, all records are deleted and all space used by the table is freed up. If the table was the current table, the current table becomes undefined.

See Also: db_table_list, db_select_table, db_clear_table

db_dump

prints the current database in readable form to file fn.

Signature:

```
db_dump(object file_id, integer low_level_too = 0)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `fn` : the destination file for printing the current Euphoria database.
≡ `low_level_too` : a boolean. If TRUE, a byte-by-byte binary dump is presented as well; otherwise this step is skipped. If omitted, FALSE is assumed.

Comments:
• All records in all tables are shown.
• If low_level_too is non-zero, then a low-level byte-by-byte dump is also shown. The low-level dump will only be meaningful to someone who is familiar with the internal format of a Euphoria database.

Example 1:

```
if db_open("mydata", DB_LOCK_SHARED) != DB_OK then
    puts(2, "Couldn't open the database!\n")
    abort(1)
end if
fn = open("db.txt", "w")
db_dump(fn) -- Simple output
db_dump("lowlvl_db.txt", 1) -- Full low-level dump created.
```

db_fatal_id

Exception handler

Signature:

```
db_fatal_id
```

```
public integer
include eds.e
namespace eds
```

Comments: Set this to a valid routine_id value for a procedure that will be called whenever the library detects a serious error. You procedure will be passed a single text string that describes the error. It may also call db_get_errors to get more detail about the cause of the error.

db_fetch_record

returns the data for the record with supplied key.

Signature:

```
db_fetch_record(object key, object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `key` : the identifier of the record to be looked up.
≡ `table_name` : optional name of table to find key in

Returns: An **integer**,
• If less than zero, the record was not found. The returned integer is the opposite of what the record number would have been, had the record been found.
• If equal to zero, an error occured. A sequence, the data for the record.

Comments: Each record in a Euphoria database consists of a key portion and a data portion. Each of these can be any Euphoria atom or sequence. **NOTE** This function does not support records that data consists of a single non-sequence value. In those cases you will need to use db_find_key and db_record_data.

See Also: db_find_key, db_record_data

Example 1:

```
printf(1, "The record['%s'] has data value:\n", {"foo"})
? db_fetch_record("foo")
```

db_find_key

finds the record in the current table with supplied key.

Signature:

```
db_find_key(object key, object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `key` : the identifier of the record to be looked up.
≡ `table_name` : optional name of table to find key in

Returns: An **integer**, either greater or less than zero:
• If above zero, the record identified by `key` was found on the current table, and the returned integer is its record number.
• If less than zero, the record was not found. The returned integer is the opposite of what the record number would have been, had the record been found.
• If equal to zero, an error occured.

Comments: A fast binary search is used to find the key in the current table. The number of comparisons is proportional to the log of the number of records in the table. The key is unique--a table is more like a dictionary than like a spreadsheet.

You can select a range of records by searching for the first and last key values in the range. If those key values do not exist, you will at least get a negative value showing io:where they would be, if they existed.

For example: suppose you want to know which records have keys greater than "GGG" and less than "MMM". If -5 is returned for key "GGG", it means a record with "GGG" as a key would be inserted as record number 5. -27 for "MMM" means a record with "MMM" as its key would be inserted as record number 27. This quickly tells you that all records, >= 5 and < 27 qualify.

See Also: db_insert, db_replace_data, db_delete_record, db_get_recid

Example 1:

```
 rec_num = db_find_key("Millennium")
 if rec_num > 0 then
     ? db_record_key(rec_num)
     ? db_record_data(rec_num)
 else
     puts(2, "Not found, but if you insert it,\n")

     printf(2, "it will be #%d\n", -rec_num)
 end if
```

db_get_errors

fetches the most recent set of errors recorded by the library.

Signature:

```
db_get_errors(integer clearing = 1)


public function
include eds.e
namespace eds
```

Arguments: ≡ `clearing` : if zero the set of errors is not reset, otherwise it will be cleared out. The default is to clear the set.

Returns: A **sequence**, each element is a set of four fields. # Error Code. # Error Text. # Name of library routine that recorded the error. # Parameters passed to that routine.

Comments:

• A number of library routines can detect errors. If the routine is a function, it usually returns an error code. However, procedures that detect an error can not do that. Instead, they record the error details and you can query that after calling the library routine.
• Both functions and procedures that detect errors record the details in the `Last Error Set`, which is fetched by this function.

Example 1:

```
 db_replace_data(recno, new_data)
 errs = db_get_errors()
 if length(errs) != 0 then
     display_errors(errs)
     abort(1)
 end if
```

db_get_recid

returns the unique record identifier (`recid`) value for the record.

Signature:

```
db_get_recid(object key, object table_name = current_table_name)


public function
include eds.e
namespace eds
```

Arguments: ≡ `key` : the identifier of the record to be looked up.
≡ `table_name` : optional name of table to find key in

Returns: An **atom**, either greater or equal to zero:
• If above zero, it is a `recid`.
• If less than zero, the record was not found.
• If equal to zero, an error occured.

Comments: A **`recid`** is a number that uniquely identifies a record in the database. No two records in a database has the same `recid` value. They can be used instead of keys to *quickly* refetch a record, as they avoid the overhead of looking for a matching record key. They can also be used without selecting a table first, as the `recid` is unique to the database and not just a table. However, they only remain valid while a database is open and so long as it does not get compressed. Compressing the database will give each record a new `recid` value.

Because it is faster to fetch a record with a `recid` rather than with its key, these are used when you know you have to **refetch** a record.

See Also: db_insert, db_replace_data, db_delete_record, db_find_key

Example 1:

```
rec_num = db_get_recid("Millennium")
if rec_num > 0 then
    ? db_record_recid(rec_num) -- fetch key and data.
else
    puts(2, "Not found\n")
end if
```

db_insert

inserts a new record into the current table.

Signature:

```
db_insert(object key, object data, object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `key` : an object, the record key, which uniquely identifies it inside the current table
≡ `data` : an object, associated to `key`.
≡ `table_name` : optional table name to insert record into

Returns: An **integer**, either DB_OK on success or an error code on failure.

Comments: Within a table, all keys must be unique. `db_insert`() will fail with `DB_EXISTS_ALREADY` if a record already exists on current table with the same key value.

Both key and data can be any Euphoria data objects, atoms or sequences.

See Also: db_replace_data, db_delete_record

Example 1:

```
if db_insert("Smith", {"Peter", 100, 34.5}) != DB_OK then
    puts(2, "insert failed!\n")
end if
```

db_open

opens an existing Euphoria database.

Signature:

```
db_open(sequence path, integer lock_method = DB_LOCK_NO)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `path` : a sequence, the path to the file containing the database
≡ `lock_method` : an integer specifying which sort of access can be granted to the database. The types of lock that you can use are: #
≡ `DB_LOCK_NO` : (no lock) - The default #
≡ `DB_LOCK_SHARED` : (shared lock for read-only access) #

≡ `DB_LOCK_EXCLUSIVE` : (for read/write access).

Returns: An **integer**, status code, either `DB_OK` if creation successful or anything else on an error.

Comments: `DB_LOCK_SHARED` is only supported on *unix* platforms. It allows you to read the database, but not write anything to it. If you request `DB_LOCK_SHARED` on *windows* it will be treated as if you had asked for DB_LOCK_EXCLUSIVE.

If the lock fails, your program should wait a few seconds and try again. Another process might be currently accessing the database.

See Also: db_create, db_select

Example 1:

```
tries = 0
while 1 do
    err = db_open("mydata", DB_LOCK_SHARED)
    if err = DB_OK then
        exit
    elsif err = DB_LOCK_FAIL then
        tries += 1
        if tries > 10 then
            puts(2, "too many tries, giving up\n")
            abort(1)
        else
            sleep(5)
        end if
    else
        puts(2, "Couldn't open the database!\n")
        abort(1)
    end if
end while
```

db_record_data

returns the data in a record queried by position.

Signature:

```
db_record_data(integer key_location,
object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `key_location` : the index of the record the data of which is being fetched.
≡ `table_name` : optional table name to get record data from.

Returns: An **object**, the data portion of requested record.
NOTE This function calls `fatal`() and returns a value of -1 if an error prevented the correct data being returned.

Comments: Each record in a Euphoria database consists of a key portion and a data portion. Each of these can be any Euphoria atom or sequence.

See Also: db_find_key, db_replace_data

Example 1:

```
puts(1, "The 6th record has data value: ")
? db_record_data(6)
```

db_record_key

Signature:

```
db_record_key(integer key_location,
object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Comments: Each record in a Euphoria database consists of a key portion and a data portion. Each of these can be any Euphoria atom or sequence.

See Also: db_record_data

Example 1:

```
puts(1, "The 6th record has key value: ")
? db_record_key(6)
```

db_record_recid

returns the key and data in a record queried by `recid`.

Signature:

```
db_record_recid(integer recid)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `recid` : the `recid` of the required record, which has been previously fetched using db_get_recid.

Returns: An **sequence**, the first element is the key and the second element is the data portion of requested record.

Comments:

• This is much faster than calling db_record_key and db_record_data.
• This does no error checking. It assumes the database is open and valid.
• This function does not need the requested record to be from the current table. The `recid` can refer to a record in any table.

See Also: db_get_recid, db_replace_recid

Example 1:

```
rid = db_get_recid("SomeKey")
? db_record_recid(rid)
```

db_rename_table

renames a table in the current database.

Signature:

```
db_rename_table(sequence name, sequence new_name)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `name` : a sequence, the name of the table to rename
≡ `new_name` : a sequence, the new name for the table

Comments: The table to be renamed can be the current table, or some other table in the current database.

See Also: db_table_list

db_replace_data

replaces the data portion of a record with new data in the current table.

Signature:

```
db_replace_data(integer key_location, object data,
object table_name = current_table_name)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `key_location`: an integer, the index of the record the data is to be altered.
≡ `data`: an object , the new value associated to the key of the record.
≡ `table_name`: optional table name of record to replace data in.

Comments: `key_location` must be from 1 to the number of records in the current table. `data` is an Euphoria object of any kind, atom or sequence.

See Also: db_find_key

Example 1:

```
 db_replace_data(67, {"Peter", 150, 34.5})
```

db_replace_recid

replaces the data portion of a record with new data In the current database.

Signature:

```
db_replace_recid(integer recid, object data)
```

```
public procedure
include eds.e
namespace eds
```

Arguments: ≡ `recid` : an atom, the `recid` of the record to be updated.
≡ `data` : an object, the new value of the record.

Comments: This procedure be used to quickly update records that have already been located by calling db_get_recid. This operation is faster than using db_replace_data

- `recid` must be fetched using db_get_recid first.
- `data` is an Euphoria object of any kind, atom or sequence.
- The `recid` does not have to be from the current table.
- This does no error checking. It assumes the database is open and valid.

See Also: db_replace_data, db_find_key, db_get_recid

Example 1:

```
 rid = db_get_recid("Peter")
 rec = db_record_recid(rid)
 rec[2][3] *= 1.10
 db_replace_recid(rid, rec[2])
```

db_select

chooses a new, already open, database to be the current database.

Signature:

```
db_select(sequence path, integer lock_method = - 1)
```

```
public function
include eds.e
```

```
namespace eds
```

Arguments:
≡ `path` : a sequence, the path to the database to be the new current database.
≡ `lock_method` : an integer. Optional locking method.

Returns: An **integer**, `DB_OK` on success or an error code.

Comments:

• Subsequent database operations will apply to this database. path is the path of the database file as it was originally opened with `db_open`() or `db_create`().

• When you create (db_create) or open (db_open) a database, it automatically becomes the current database. Use `db_select`() when you want to switch back and forth between open databases, perhaps to copy records from one to the other. After selecting a new database, you should select a table within that database using `db_select_table`().
• If the `lock_method` is omitted and the database has not already been opened, this function will fail. However, if `lock_method` is a valid lock type for db_open and the database is not open yet, this function will attempt to open it. It may still fail if the database cannot be opened.

See Also: db_open, db_select

Example 1:

```
 if db_select("employees") != DB_OK then
     puts(2, "Could not select employees database\n")
 end if
```

Example 2:

```
 if db_select("customer", DB_LOCK_SHARED) != DB_OK then
     puts(2, "Could not open or select Customer database\n")
 end if
```

db_select_table

Signature:

```
db_select_table(sequence name)


public function
include eds.e
namespace eds
```

Returns: An **integer**, either DB_OK on success or DB_OPEN_FAIL otherwise.

Comments: All record-level database operations apply automatically to the current table.

See Also: db_table_list

Example 1:

```
 if db_select_table("salary") != DB_OK then
     puts(2, "Couldn't find salary table!\n")
     abort(1)
 end if
```

db_set_caching

sets the key cache behavior.

Signature:

```
db_set_caching(atom new_setting)


public function
include eds.e
```

```
namespace eds
```

Arguments: ≡ `integer` : 0 will turn of caching, 1 will turn it back on.

Returns: An **integer**, the previous setting of the option.

Comments: Initially, the cache option is turned on. This means that when possible, the keys of a table are kept in RAM rather than read from disk each time `db_select_table`() is called. For most databases, this will improve performance when you have more than one table in it.

Notes: When caching is turned off, the current cache contents is totally cleared.

Example 1:

```
x = db_set_caching(0) -- Turn off key caching.
```

db_table_list

lists all tables on the current database.

Signature:

```
db_table_list()
```

```
public function
include eds.e
namespace eds
```

Returns: A **sequence**, of all the table names in the current database. Each element of this sequence is a sequence, the name of a table.

See Also: db_select_table, db_create_table

Example 1:

```
sequence names = db_table_list()
for i = 1 to length(names) do
    puts(1, names[i] & '\n')
end for
```

db_table_size

gets the size (number of records) of the default table.

Signature:

```
db_table_size(object table_name = current_table_name)
```

```
public function
include eds.e
namespace eds
```

Arguments: ≡ `table_name` : optional table name to get the size of.

Returns An **integer**, the current number of records in the current table. If a value less than zero is returned, it means that an error occured.

See Also: db_replace_data

Example 1:

```
-- look at all records in the current table
for i = 1 to db_table_size() do
    if db_record_key(i) = 0 then
     puts(1, "0 key found\n")
     exit
    end if
end for
```

error

Routines

error API

abort

aborts execution of the program.

Signature:

```
abort(atom error)
```

```
<built-in> procedure
```

Arguments: ≡ `error` : an integer, the exit code to return.

Comments: `error` is expected to lie in the 0..255 range. 0 is usually interpreted as the sign of a successful completion.

Other values can indicate various kinds of errors. Windows batch (.bat) programs can read this value using the errorlevel feature. Non integer values are rounded down. A Euphoria program can read this value using system_exec().

`abort`() is useful when a program is many levels deep in subroutine calls, and execution must end immediately, perhaps due to a severe error that has been detected.

If you do not use `abort`(), the interpreter will normally return an exit status code of 0. If your program fails with a Euphoria-detected compile-time or run-time error then a code of 1 is returned.

See Also: crash_message, system_exec

Example 1:

```
if x = 0 then
    puts(ERR, "can't divide by 0 !!!\n")
    abort(1)
else
    z = y / x
end if
```

crash

crashes the running program and displays a formatted error message.

Signature:

```
crash(sequence fmt, object data = {})
```

```
public procedure
include error.e
namespace error
```

Arguments: ≡ `fmt` : a sequence representing the message text. It may have format specifiers in it
≡ `data` : an object, defaulted to {}.

Comments: Formatting follows the conventions that `printf`() does.

The actual message being shown, both on standard error and in ex.err (or whatever file last passed to crash_file()), is `sprintf(fmt, data)`. The program terminates as for any runtime error.

See Also: crash_file, crash_message, printf

Example 1:

```
if PI = 3 then
    crash("The structure of universe just changed -- reload solar_system.ex")
end if
```

Example 2:

```
if token = end_of_file then
    crash("Test file #%d is bad, text read so far is %s\n",
                                             {file_number, read_so_far})
end if
```

crash_file

specifies a file path name in place of "ex.err" where you want

Signature:

```
crash_file(sequence file_path)
```

```
public procedure
include error.e
namespace error
```

Arguments: ≡ `file_path` : a sequence, the new error and traceback file path.

Comments: There can be as many calls to `crash_file`() as needed. Whatever was defined last will be used in case of an error at runtime, whether it was triggered by crash() or not.

See Also: crash, crash_message

crash_message

specifies a final message to display for your user, in the event

Signature:

```
crash_message(sequence msg)
```

```
public procedure
include error.e
namespace error
```

Arguments: ≡ `msg` : a sequence to display. It must only contain printable characters.

Comments: There can be as many calls to `crash_message`() as needed in a program. Whatever was defined last will be used in case of a runtime error.

See Also: crash, crash_file

Example 1:

```
 crash_message("The password you entered must have at least 8 characters.")
 pwd_key = input_text[1..8]
 -- if ##input_text## is too short,
 -- user will get a more meaningful message than
 -- "index out of bounds".
```

crash_routine

specifies a function to be called when an error takes place at run time.

Signature:

```
crash_routine(integer func)
```

```
public procedure
include error.e
namespace error
```

Arguments: ≡ `func` : an integer, the routine_id of the function to link in.

Comments: The supplied function must have only one parameter, which should be integer or more general. Defaulted parameters in crash routines are not supported yet.

Euphoria maintains a linked list of routines to execute upon a crash. `crash_routine`() adds a new function to the list. The routines defined first are executed last. You cannot unlink a routine once it is linked, nor inspect the crash routine chain.

Currently, the crash routines are passed 0. Future versions may attempt to convey more information to them. If a crash routine returns anything else than 0, the remaining routines in the chain are skipped.

crash routines are not full fledged exception handlers, and they cannot resume execution at current or next statement. However, they can read the generated crash file, and might perform any action, including restarting the program.

See Also: crash_file, routine_id, Debugging and Profiling

Example 1:

```
 function report_error(integer dummy)
   mylib:email("maintainer@remote_site.org", "ex.err")
    return 0 and dummy
 end function
 crash_routine(routine_id("report_error"))
```

warning

causes the specified warning message to be displayed as a regular warning.

Signature:

```
warning(sequence message)
```

```
<built-in> procedure
```

Arguments: ≡ `message` : a double quoted literal string, the text to display.

Comments: Writing a library has specific requirements, since the code you write will be mainly used inside code you did not write. It may be desirable then to influence, from inside the library, that code you did not write.

This is what `warning`(), in a limited way, does. It enables to generate custom warnings in code that will include yours. Of course, you can also generate warnings in your own code, for instance as a kind of memo. The On/off options|without warning top level statement disables such warnings.

The warning is issued with the `custom_warning` level. This level is enabled by default, but can be turned off any time.

Using any kind of expression in `message` will result in a blank warning text.

See Also: warning_file

Example 1:

```
-- mylib.e
procedure foo(integer n)
    warning("The foo() procedure is obsolete, use bar() instead.")
    ? n
end procedure

-- some_app.exw
include mylib.e
foo(123)
```

will result, when some_app.exw is run with warning, in the following text being displayed in the console window

```
123
Warning: ( custom_warning ):
The foo() procedure is obsolete, use bar() instead.

Press Enter...
```

warning_file

specifies a file path where to output warnings.

Signature:

```
warning_file(object file_path)
```

```
public procedure
include error.e
namespace error
```

Arguments: ≡ `file_path` : an object indicating where to dump any warning that were produced.

Comments: By default, warnings are displayed on the standard error, and require pressing the Enter key to keep going. Redirecting to a file enables skipping the latter step and having a console window open, while retaining ability to inspect the warnings in case any was issued.

Any atom >= 0 causes standard error to be used, thus reverting to default behaviour.

Any atom < 0 suppresses both warning generation and output. Use this latter in extreme cases only.

On an error, some output to the console is performed anyway, so that whatever warning file was specified is ignored then.

See Also: without warning, warning

Example 1:

```
warning_file("warnings.lst")
-- some code
warning_file(0)
-- changed opinion: warnings will go to standard error as usual
```

eumem

eumem API

free

deallocates a block of (pseudo) memory.

Signature:

```
free(atom mem_p)
```

```
export procedure
include eumem.e
namespace eumem
```

Arguments: ≡ `mem_p` : The handle to a previously acquired ram_space location.

Comments: This allows the location to be used by other parts of your application. You should no longer access this location again because it could be acquired by some other process in your application. This routine should only be called if you passed 0 as `cleanup_p` to malloc.

Example 1:

```
my_spot = malloc(1,0)
 ram_space[my_spot] = my_data
     -- . . . do some processing  . . .
 free(my_spot)
```

malloc

allocates a block of (pseudo) memory.

Signature:

```
malloc(object mem_struct_p = 1, integer cleanup_p = 1)
```

```
export function
include eumem.e
namespace eumem
```

Arguments: ≡ `mem_struct_p` : The initial structure (sequence) to occupy the allocated block. If this is an integer, a sequence of zero this long is used. The default is the number 1, meaning that the default initial structure is {0}
≡ `cleanup_p` : Identifies whether the memory should be released automatically when the reference count for the handle for the allocated block drops to zero, or when passed to `delete()`. If 0, then the block must be freed using the free procedure.

Returns: A **handle**, to the acquired block. Once you acquire this, you can use it as you need to. Note that if `cleanup_p` is 1, then the variable holding the handle must be capable of storing an atom as a double floating point value (i.e., not an integer).

Example 1:

```
my_spot = malloc()
 ram_space[my_spot] = my_data
```

ram_space

is the (pseudo) RAM heap space.

Signature:

```
ram_space
```

```
export sequence
include eumem.e
namespace eumem
```

Comments: Use malloc to gain ownership to a heap location and free to release it back to the system.

valid

validates a block of (pseudo) memory.

Signature:

```
valid(object mem_p, object mem_struct_p = 1)
```

```
export function
include eumem.e
namespace eumem
```

Arguments: ≡ `mem_p` : The handle to a previously acquired ram_space location.
≡ `mem_struct_p` : If an integer, this is the length of the sequence that should be occupying the ram_space location pointed to by `mem_p`.

Returns: An **integer**,
0 if either the `mem_p` is invalid or if the sequence at that location is the wrong length.
1 if the handle and contents is okay.

Comments: This can only check the length of the contents at the location. Nothing else is checked at that location.

Example 1:

```
my_spot = malloc()
 ram_space[my_spot] = my_data

 if valid(my_spot, length(my_data)) then
     free(my_spot)
 end if
```

filesys

Constants

SLASH
SLASHES
EOLSEP
EOL
PATHSEP
NULLDEVICE
SHARED_LIB_EXT

Directory Handling

D_NAME
D_ATTRIBUTES

D_SIZE
D_YEAR
D_MONTH
D_DAY
D_HOUR
D_MINUTE
D_SECOND
D_MILLISECOND
D_ALTNAME

Bad path error code. See walk_dir

W_BAD_PATH
dir
current_dir
chdir
my_dir
walk_dir
create_directory
create_file
delete_file
curdir
init_curdir
clear_directory
remove_directory

File name parsing

PATH_DIR
PATH_FILENAME
PATH_BASENAME
PATH_FILEEXT
PATH_DRIVEID
pathinfo
dirname
pathname
filename
filebase
fileext
driveid
defaultext
absolute_path
AS_IS
TO_LOWER
CORRECT
TO_SHORT
case_flagset_type
canonical_path
abbreviate_path
split_path
join_path

File Types

FILETYPE_UNDEFINED
FILETYPE_NOT_FOUND
FILETYPE_FILE
FILETYPE_DIRECTORY

filesys API

AS_IS

Signature:

```
AS_IS
```

```
public enum
include filesys.e
namespace filesys
```

BYTES_PER_SECTOR

Signature:

```
BYTES_PER_SECTOR
```

```
public enum
include filesys.e
namespace filesys
```

CORRECT

Signature:

```
CORRECT
```

```
public enum
include filesys.e
namespace filesys
```

COUNT_DIRS

Signature:

```
COUNT_DIRS
```

```
public enum
include filesys.e
namespace filesys
```

COUNT_FILES

Signature:

```
COUNT_FILES
```

```
public enum
include filesys.e
namespace filesys
```

COUNT_SIZE

Signature:

```
COUNT_SIZE
```

```
public enum
include filesys.e
namespace filesys
```

COUNT_TYPES

Signature:

```
COUNT_TYPES
```

```
public enum
include filesys.e
namespace filesys
```

D_ALTNAME

Signature:

```
D_ALTNAME
```

```
public enum
include filesys.e
namespace filesys
```

D_ATTRIBUTES

Signature:

```
D_ATTRIBUTES
```

```
public enum
include filesys.e
namespace filesys
```

D_DAY

Signature:

```
D_DAY
```

```
public enum
include filesys.e
namespace filesys
```

D_HOUR

Signature:

```
D_HOUR
```

```
public enum
include filesys.e
namespace filesys
```

D_MILLISECOND

Signature:

```
D_MILLISECOND
```

```
public enum
include filesys.e
namespace filesys
```

D_MINUTE

Signature:

```
D_MINUTE
```

```
public enum
include filesys.e
namespace filesys
```

D_MONTH

Signature:

```
D_MONTH
```

```
public enum
include filesys.e
namespace filesys
```

D_NAME

Signature:

```
D_NAME
```

```
public enum
include filesys.e
namespace filesys
```

D_SECOND

Signature:

```
D_SECOND
```

```
public enum
include filesys.e
namespace filesys
```

D_SIZE

Signature:

```
D_SIZE
```

```
public enum
include filesys.e
namespace filesys
```

D_YEAR

Signature:

```
D_YEAR
```

```
public enum
include filesys.e
namespace filesys
```

EOL

All platform's newline character: `'\n'`. When text lines are read the native

Signature:

```
EOL
```

```
public constant
```

EOLSEP

is the newline string for the current platform.

Signature:

```
EOLSEP
```

```
public constant
```

Comments: `"\n"` on *unix* or else `"\r\n"` on *windows*.

EXT_COUNT

Signature:

```
EXT_COUNT
```

```
public enum
include filesys.e
namespace filesys
```

EXT_NAME

Signature:

```
EXT_NAME
```

```
public enum
include filesys.e
namespace filesys
```

EXT_SIZE

Signature:

```
EXT_SIZE
```

```
public enum
include filesys.e
namespace filesys
```

FILETYPE_DIRECTORY

Signature:

```
FILETYPE_DIRECTORY
```

```
public enum
```

```
include filesys.e
namespace filesys
```

FILETYPE_FILE

Signature:

```
FILETYPE_FILE
```

```
public enum
include filesys.e
namespace filesys
```

FILETYPE_NOT_FOUND

Signature:

```
FILETYPE_NOT_FOUND
```

```
public enum
include filesys.e
namespace filesys
```

FILETYPE_UNDEFINED

Signature:

```
FILETYPE_UNDEFINED
```

```
public enum
include filesys.e
namespace filesys
```

FREE_BYTES

Signature:

```
FREE_BYTES
```

```
public enum
include filesys.e
namespace filesys
```

NULLDEVICE

is the null device path for the current platform.

Signature:

```
NULLDEVICE
```

```
public constant
```

Comments: `/dev/null` on *unix* or else `NUL:` on *windows*.

NUMBER_OF_FREE_CLUSTERS

Signature:

```
NUMBER_OF_FREE_CLUSTERS
```

```
public enum
include filesys.e
namespace filesys
```

PATHSEP

is the path separator character for the current platform.

Signature:

```
PATHSEP
```

```
public constant
```

Comments: `:` on *unix*, else `;` on *windows*.

PATH_BASENAME

Signature:

```
PATH_BASENAME
```

```
public enum
include filesys.e
namespace filesys
```

PATH_DIR

Signature:

```
PATH_DIR
```

```
public enum
include filesys.e
namespace filesys
```

PATH_DRIVEID

Signature:

```
PATH_DRIVEID
```

```
public enum
include filesys.e
namespace filesys
```

PATH_FILEEXT

Signature:

```
PATH_FILEEXT
```

```
public enum
include filesys.e
namespace filesys
```

PATH_FILENAME

Signature:

```
PATH_FILENAME
```

```
public enum
include filesys.e
namespace filesys
```

SECTORS_PER_CLUSTER

Signature:

```
SECTORS_PER_CLUSTER
```

```
public enum
include filesys.e
namespace filesys
```

SHARED_LIB_EXT

is the shared library extension for the current platform.

Signature:

```
SHARED_LIB_EXT
```

```
public constant
```

Comments: For instance it can be `dll` (*windows*), `so` (*unix*) or `dylib` (*os x*) depending on the platform.

SLASH

is the path separator character for the current.

Signature:

```
SLASH
```

```
public constant
```

Comments: When on *windows*, '\\'. When on *unix*, '/'.

SLASHES

are the possible path separators for the current platform.

Signature:

```
SLASHES
```

```
public constant
```

Comments: On *unix* systems, it only contains slash (`/`).

On *windows* the path separators are: the traditional backslash which must be written as a double slash (`\\`) in a string sequence, the (`:`), and on newer versions of *windows* the slash (`/`).

TOTAL_BYTES

Signature:

```
TOTAL_BYTES
```

```
public enum
include filesys.e
namespace filesys
```

TOTAL_NUMBER_OF_CLUSTERS

Signature:

```
TOTAL_NUMBER_OF_CLUSTERS
```

```
public enum
include filesys.e
namespace filesys
```

TO_LOWER

Signature:

```
TO_LOWER
```

```
public enum
include filesys.e
namespace filesys
```

TO_SHORT

Signature:

```
TO_SHORT
```

```
public enum
include filesys.e
namespace filesys
```

USED_BYTES

Signature:

```
USED_BYTES
```

```
public enum
include filesys.e
namespace filesys
```

W_BAD_PATH

Signature:

```
W_BAD_PATH
```

```
public constant
include filesys.e
namespace filesys
```

abbreviate_path

returns a path string to the supplied file which is shorter than the

Signature:

```
abbreviate_path(sequence orig_path, sequence base_paths = {})
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `orig_path` : A sequence. This is the path to a file.
≡ `base_paths` : A sequence. This is an optional list of paths that may prefix the original path. The default is an empty list.

Returns: A **sequence**, an equivalent path to `orig_path` which is shorter than the supplied path. If a shorter one cannot be formed, then the original path is returned.

Comments:

- This function is primarily used to get the shortest form of a file path for output to a file or screen.
- It works by first trying to find if the `orig_path` begins with any of the `base_paths`. If so it returns the parameter minus the base path prefix.
- Next it checks if the `orig_path` begins with the current directory path. If so it returns the parameter minus the current directory path.
- Next it checks if it can form a relative path from the current directory to the supplied file which is shorter than the parameter string.
- Failing all of that, it returns the original parameter.
- In *windows*, the shorter result has all '/' characters are replaced by '\' characters.
- The supplied path does not have to actually exist.
- `orig_path` can be enclosed in quotes, which will be stripped off.
- If `orig_path` begins with a tilde '~' then that is replaced by the contents of $HOME in *unix* platforms and %HOMEDRIVE%%HOMEPATH% in *windows*.

Example 1:

```
-- Assuming the current directory is "/usr/foo/bar"
res = abbreviate_path("/usr/foo/abc.def")
-- res is now "../abc.def"
res = abbreviate_path("/usr/foo/bar/inc/abc.def")
-- res is now "inc/abc.def"
res = abbreviate_path("abc.def", {"/usr/foo"})
-- res is now "bar/abc.def"
```

absolute_path

determines if the supplied string is an absolute path or a relative path.

Signature:

```
absolute_path(sequence filename)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `filename` : a sequence, the name of the file path

Returns: An **integer**, 0 if `filename` is a relative path or 1 otherwise.

Example 1:

```
 ? absolute_path("") -- returns 0
 ? absolute_path("/usr/bin/abc") -- returns 1
 ? absolute_path("\\temp\\somefile.doc") -- returns 1
 ? absolute_path("../abc") -- returns 0
 ? absolute_path("local/abc.txt") -- returns 0
 ? absolute_path("abc.txt") -- returns 0
 ? absolute_path("c:..\\abc") -- returns 0

 -- The next two examples return
 -- 0 on Unix platforms and
 -- 1 on Microsoft platforms
 ? absolute_path("c:\\windows\\system32\\abc")
 ? absolute_path("c:/windows/system32/abc")
```

canonical_path

returns the full path and file name of the supplied file name.

Signature:

```
canonical_path(sequence path_in, integer directory_given = 0,
case_flagset_type case_flags = AS_IS)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path_in` : A sequence. This is the file name whose full path you want. # `directory_given` : An integer. This is zero if `path_in` is to be interpreted as a file specification otherwise it is assumed to be a directory specification. The default is zero.
≡ `case_flags` : An integer. This is a combination of flags.

Returns: A **sequence**, the full path and file name.

Comments: Flags for the `case_flags` argument:

AS_IS = Includes no flags TO_LOWER = If passed will convert the part of the path not affected by other case flags to lowercase. CORRECT = If passed will correct the parts of the filepath that exist in the current filesystem in parts of the filesystem that is case insensitive. This should work on WINDOWS or SMB mounted volumes on UNIX and all Mac OS filesystems.

TO_LOWER = If passed alone the entire path is converted to lowercase. or_bits(TO_LOWER,CORRECT) = If these flags are passed together the the part that exists has the case of that of the filesystem. The part that does not is converted to lower case. TO_SHORT = If passed the elements of the path that exist are also converted to their WINDOWS short names if avaliable.

- The supplied file or directory does not have to actually exist.

• `path_in` can be enclosed in quotes, which will be stripped off.
• If `path_in` begins with a tilde '~~' then that is replaced by the contents of $HOME in *unix* platforms and %HOMEDRIVE%%HOMEPATH% in *windows*.
• In *windows* all '/' characters are replaced by '\' characters.
• Does not (yet) handle UNC paths or *unix* links.

Example 1:

```
-- Assuming the current directory is "/usr/foo/bar"
res = canonical_path("../abc.def")
-- res is now "/usr/foo/abc.def"
```

Example 2:

```
-- res is "C:\Program Files" on systems that have that directory.
res = canonical_path(
-- on Windows Vista this would be "c:\Program Files"
-- since Vista uses lowercase for its drives.
```

case_flagset_type

Signature:

```
case_flagset_type(integer x)


public type
include filesys.e
namespace filesys
```

chdir

sets a new value for the current directory.

Signature:

```
chdir(sequence newdir)


public function
include filesys.e
namespace filesys
```

Arguments: `newdir` : a sequence, the name for the new working directory.

Returns: An **integer**, 0 on failure, 1 on success.

Comments: By setting the current directory, you can refer to files in that directory using just the file name.

The current_dir() function will return the name of the current directory.

On *windows* the current directory is a public property shared by all the processes running under one shell. On *unix* a subprocess can change the current directory for itself, but this will not affect the current directory of its parent process.

See Also: current_dir, dir

Example 1:

```
if chdir("c:\\euphoria") then
    f = open("readme.doc", "r")
else
    puts(STDERR, "Error: No euphoria directory?\n")
end if
```

checksum

returns a checksum value for the specified file.

Signature:

```
checksum(sequence filename, integer size = 4, integer usename = 0,
integer return_text = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `filename` : A sequence. The name of the file whose checksum you want.
≡ `size` : An integer. The number of atoms to return. Default is 4
≡ `usename`: An integer. If not zero then the actual text of `filename` will affect the resulting checksum. The default (0) will not use the name of the file.
≡ `return_text`: An integer. If not zero, the check sum is returned as a text string of hexadecimal digits otherwise (the default) the check sum is returned as a sequence of `size` atoms.

Returns: A **sequence** containing `size` atoms.

Comments:

• The larger the `size` value, the more unique will the checksum be. For most files and uses, a single atom will be sufficient as this gives a 32-bit file signature. However, if you require better proof that the content of two files are different then use higher values for `size`. For example, `size = 8` gives you 256 bits of file signature.
• If `size` is zero or negative, an empty sequence is returned.
• All files of zero length will return the same checksum value when `usename` is zero.

Example 1:

```
 -- Example values. The exact values depend on the contents of the file.
  include std/console.e
  display( checksum("myfile", 1) )   --> {92837498}
  display( checksum("myfile", 2) )   --> {1238176, 87192873}
  display( checksum("myfile", 2,,1)) --> "0012E480 05327529"
  display( checksum("myfile", 4) )   --> {23448, 239807, 79283749, 427370}
  display( checksum("myfile") )      --> {23448, 239807, 79283749, 427370}
                                     -- default
```

clear_directory

clears (deletes) a directory of all its files, but retains the sub-directories.

Signature:

```
clear_directory(sequence path, integer recurse = 1)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name of the directory whose files you want to remove.
≡ `recurse` : an integer, whether or not to remove files in the directory's sub-directories. If 0 then this function is identical to remove_directory(). If 1, then we recursively delete the directory and its contents. Defaults to 1.

Returns: An **integer**, 0 on failure, otherwise the number of files plus 1.

Comments: This never removes a directory; it only removes files. It is used to clear a directory structure of all existing files, leaving the structure intact.

See Also: remove_directory, delete_file

Example 1:

```
 integer cnt = clear_directory("the_old_folder")
 if cnt = 0 then
  crash("Filesystem problem - could not remove one or more of the files.")
 end if
 printf(1, "Number of files removed: %d\n", cnt - 1)
```

copy_file

copies a file.

Signature:

```
copy_file(sequence src, sequence dest, integer overwrite = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `src` : a sequence, the name of the file or directory to copy
≡ `dest` : a sequence, the new name or location of the file
≡ `overwrite` : an integer; 0 (the default) will prevent an existing destination file from being overwritten. Non-zero will overwrite the destination file.

Returns: An **integer**, 0 on failure, 1 on success.

Comments: If `overwrite` is true, and if dest file already exists, the function overwrites the existing file and succeeds.

See Also: move_file, rename_file

create_directory

creates a new directory.

Signature:

```
create_directory(sequence name, integer mode = 448, integer mkparent = 1)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name of the new directory to create
≡ `mode` : on *unix* systems, permissions for the new directory. Default is 448 (all rights for owner, none for others).
≡ `mkparent` : If true (default) the parent directories are also created if needed.

Returns: An **integer**, 0 on failure, 1 on success.

Comments: `mode` is ignored on non-unix platforms.

See Also: remove_directory, chdir

Example 1:

```
 if not create_directory("the_new_folder") then
  crash("Filesystem problem - could not create the new folder")
 end if

 -- This example will also create "myapp/" and "myapp/interface/"
 -- if they don't exist.
 if not create_directory("myapp/interface/letters") then
  crash("Filesystem problem - could not create the new folder")
 end if

 -- This example will NOT create "myapp/" and "myapp/interface/"
 -- if they don't exist.
 if not create_directory("myapp/interface/letters",,0) then
  crash("Filesystem problem - could not create the new folder")
 end if
```

create_file

creates a new file.

Signature:

```
create_file(sequence name)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name of the new file to create

Returns: An **integer**, 0 on failure, 1 on success.

Comments:

• The created file will be empty, that is it has a length of zero.
• The created file will not be open when this returns.

See Also: create_directory

Example 1:

```
 if not create_file("the_new_file") then
  crash("Filesystem problem - could not create the new file")
 end if
```

curdir

returns the current directory, with a trailing SLASH.

Signature:

```
curdir(integer drive_id = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `drive_id` : For non-Unix systems only. This is the Drive letter to to get the current directory of. If omitted, the current drive is used.

Returns: A **sequence**, the current directory.

Example 1:

```
 res = curdir('D') -- Find the current directory on the D: drive.
 -- res might be "D:\backup\music\"
 res = curdir()    -- Find the current directory on the current drive.
 -- res might be "C:\myapp\work\"
```

current_dir

returns the name of the current working directory.

Signature:

```
current_dir()
```

```
public function
include filesys.e
namespace filesys
```

Returns: A **sequence**, the name of the current working directory

Comments: There will be no slash or backslash on the end of the current directory, except under *windows*, at the top-level of a drive, e.g. C:\

See Also: dir, chdir

Example 1:

```
sequence s
s = current_dir()
-- s would have "C:\EUPHORIA\DOC" if you were in that directory
```

defaultext

returns the supplied filepath with the supplied extension, if

Signature:

```
defaultext(sequence path, sequence defext)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path to check for an extension.
≡ `defext` : the extension to add if `path` does not have one.

Returns: A **sequence**, the path with an extension.

See Also: pathinfo

Example 1:

```
-- ensure that the supplied path has an extension,
 -- but if it doesn't use "tmp".
theFile = defaultext(UserFileName, "tmp")
```

delete_file

deletes a file.

Signature:

```
delete_file(sequence name)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name of the file to delete.

Returns: An **integer**, 0 on failure, 1 on success.

dir

returns directory information for the specified file or directory.

Signature:

```
dir(sequence name)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name to be looked up in the file system.

Returns: An **object**, -1 if no match found, else a sequence of sequence entries

Comments: `name` can also contain * and ? wildcards to select multiple files.

The returned information is similar to what you would get from the DIR command. A sequence is returned where each element is a sequence that describes one file or subdirectory.

If `name` refers to a **directory** you may have entries for "." and "..", just as with the DIR command. If it refers to an existing **file**, and has no wildcards, then the returned sequence will have just one entry, i.e. its length will be 1. If `name` contains wildcards you may have multiple entries.

Each entry contains the name, attributes and file size as well as the time of the last modification.

You can refer to the elements of an entry with the following constants:

See Also: walk_dir

Example 1:

```
d = dir(current_dir())

-- d might have:
--  {
--    {".",    "d",      0  1994, 1, 18,  9, 30, 02},
--    {"..",   "d",      0  1994, 1, 18,  9, 20, 14},
--    {"fred", "ra", 2350, 1994, 1, 22, 17, 22, 40},
--    {"sub",  "d" ,     0, 1993, 9, 20,  8, 50, 12}
--  }

d[3][D_NAME] would be "fred"
```

dir_size

returns the amount of space used by a directory.

Signature:

```
dir_size(sequence dir_path, integer count_all = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments:
≡ `dir_path` : A sequence. This is the path that identifies the directory to inquire upon.
≡ `count_all` : An integer. Used by Windows systems. If zero (the default) it will not include *system* or *hidden* files in the count, otherwise they are included.

Returns: A **sequence**, containing four elements; the number of sub-directories [COUNT_DIRS], the number of files [COUNT_FILES], the total space used by the directory [COUNT_SIZE], and breakdown of the file contents by file extension [COUNT_TYPES].

Comments:

• The total space used by the directory does not include space used by any sub-directories.
• The file breakdown is a sequence of three-element sub-sequences. Each sub-sequence contains the extension [EXT_NAME], the number of files of this extension [EXT_COUNT], and the space used by these files [EXT_SIZE]. The sub-sequences are presented in extension name order. On Windows the extensions are all in lowercase.

Example 1:

```
res = dir_size("/usr/localbin")
printf(1, "Directory %s contains %d files\n", {
        "/usr/localbin", res[COUNT_FILES]
    })
for i = 1 to length(res[COUNT_TYPES]) do
    printf(1, "Type: %s (%d files %d bytes)\n", {
        res[COUNT_TYPES][i][EXT_NAME],
        res[COUNT_TYPES][i][EXT_COUNT],
        res[COUNT_TYPES][i][EXT_SIZE]
    })
end for
```

dirname

returns the directory name of a fully qualified filename.

Signature:

```
dirname(sequence path, integer pcd = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information
≡ `pcd` : If not zero and there is no directory name in `path` then "." is returned. The default (0) will just return any directory name in `path`.

Returns: A **sequence**, the full file name part of `path`.

Comments: The host operating system path separator is used.

See Also: driveid, filename, pathinfo

Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

disk_metrics

returns some information about a disk drive.

Signature:

```
disk_metrics(object disk_path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `disk_path` : A sequence. This is the path that identifies the disk to inquire upon.

Returns: A **sequence**, containing `SECTORS_PER_CLUSTER`, `BYTES_PER_SECTOR`, `NUMBER_OF_FREE_CLUSTERS`, and `TOTAL_NUMBER_OF_CLUSTERS`

Example 1:

```
res = disk_metrics("C:\\")
min_file_size = res[SECTORS_PER_CLUSTER] * res[BYTES_PER_SECTOR]
```

disk_size

returns the amount of space for a disk drive.

Signature:

```
disk_size(object disk_path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `disk_path` : A sequence. This is the path that identifies the disk to inquire upon.

Returns: A **sequence**, containing TOTAL_BYTES, USED_BYTES, FREE_BYTES, and a string which represents the filesystem name

Example 1:

```
res = disk_size("C:\\")
printf(1, "Drive %s has %3.2f%% free space\n", {
    "C:", res[FREE_BYTES] / res[TOTAL_BYTES]
})
```

driveid

returns the drive letter of the path on *windows* platforms.

Signature:

```
driveid(sequence path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information

Returns: A **sequence**, the file extension part of `path`.

Platform: *windows*

TODO: Test

See Also: pathinfo, dirname, filename

Example 1:

```
letter = driveid("C:\\EUPHORIA\\Readme.txt")
-- letter is "C"
```

file_exists

tests if a file exists.

Signature:

```
file_exists(object name)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : filename to check existence of

Returns: An **integer**, 1 on yes, 0 on no

Example 1:

```
if file_exists("abc.e") then
    puts(1, "abc.e exists already\n")
end if
```

file_length

returns the size of a file.

Signature:

```
file_length(sequence filename)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `filename` : the name of the queried file

Returns: An **atom**, the file size, or -1 if file is not found.

Comments: This function does not compute the total size for a directory, and returns 0 instead.

See Also: dir

file_timestamp

gets the timestamp of the file.

Signature:

```
file_timestamp(sequence fname)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `name` : the filename to get the date of

Returns: A valid **datetime type**, representing the files date and time or -1 if the file's date and time could not be read.

file_type

gets the type of a file.

Signature:

```
file_type(sequence filename)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `filename` : the name of the file to query. It must not have wildcards.

Returns: An **integer**,
- -1 if file could be multiply defined
- 0 if filename does not exist
- 1 if filename is a file
- 2 if filename is a directory

See Also: dir, FILETYPE_DIRECTORY, FILETYPE_FILE, FILETYPE_NOT_FOUND, FILETYPE_UNDEFINED

filebase

returns the base filename of path.

Signature:

```
filebase(sequence path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information

Returns: A **sequence**, the base file name part of `path`.

TODO: Test

See Also: pathinfo, filename, fileext

Example 1:

```
base = filebase("/opt/euphoria/readme.txt")
-- base is "readme"
```

fileext

returns the file extension of a fully qualified filename.

Signature:

```
fileext(sequence path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information

Returns: A **sequence**, the file extension part of `path`.

Comments: The host operating system path separator is used.

See Also: pathinfo, filename, filebase

Example 1:

```
fname = fileext("/opt/euphoria/docs/readme.txt")
-- fname is "txt"
```

filename

returns the file name portion of a fully qualified filename.

Signature:

```
filename(sequence path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information

Returns: A **sequence**, the file name part of `path`.

Comments: The host operating system path separator is used.

See Also: pathinfo, filebase, fileext

Example 1:

```
fname = filename("/opt/euphoria/docs/readme.txt")
-- fname is "readme.txt"
```

init_curdir

returns the original current directory.

Signature:

```
init_curdir()
```

```
public function
include filesys.e
namespace filesys
```

Returns: A **sequence**, the current directory at the time the program started running.

Example 1:

```
res = init_curdir() -- Find the original current directory.
```

join_path

joins multiple path segments into a single path filename.

Signature:

```
join_path(sequence path_elements)
```

```
public function
include filesys.e
namespace filesys
```

Arguments:

- `path_elements` - Sequence of path elements

Returns: A string representing the path elements on the given platform

See Also: split_path

Example 1:

```
sequence fname = join_path({ "usr", "home", "john", "hello.txt" })
-- fname would be "/usr/home/john/hello.txt" on Unix
-- fname would be "\\usr\\home\\john\\hello.txt" on Windows
```

locate_file

locates a file by searching in a set of directories for it.

Signature:

```
locate_file(sequence filename, sequence search_list = {},
sequence subdir = {})
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `filename` : a sequence, the name of the file to search for.
≡ `search_list` : a sequence, the list of directories to look in. By default this is "", meaning that a predefined set of directories is scanned. See comments below.
≡ `subdir` : a sequence, the sub directory within the search directories to check. This is optional.

Returns: A **sequence**, the located file path if found, else the original file name.

Comments: If `filename` is an absolute path, it is just returned and no searching takes place.

If `filename` is located, the full path of the file is returned.

If `search_list` is supplied, it can be either a sequence of directory names, of a string of directory names delimited by ':' in UNIX and ';' in Windows.

If the `search_list` is omitted or "", this will look in the following places...
- The current directory
- The directory that the program is run from.
- The directory in $HOME ($HOMEDRIVE & $HOMEPATH in Windows)
- The parent directory of the current directory
- The directories returned by include_paths()
- $EUDIR/bin
- $EUDIR/docs

• $EUDIST/
• $EUDIST/etc
• $EUDIST/data
• The directories listed in $USERPATH
• The directories listed in $PATH

If the `subdir` is supplied, the function looks in this sub directory for each of the directories in the search list.

Example 1:

```
 res = locate_file("abc.def", {"/usr/bin", "/u2/someapp", "/etc"})
  res = locate_file("abc.def", "/usr/bin:/u2/someapp:/etc")
  res = locate_file("abc.def")
        -- Scan default locations.
  res = locate_file("abc.def", , "app")
        -- Scan the 'app' sub directory in the default locations.
```

move_file

moves a file to another location.

Signature:

```
move_file(sequence src, sequence dest, integer overwrite = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `src` : a sequence, the name of the file or directory to move
≡ `dest` : a sequence, the new location for the file
≡ `overwrite` : an integer, 0 (the default) to prevent overwriting an existing destination file, 1 to overwrite existing destination file

Returns: An **integer**, 0 on failure, 1 on success.

Comments: If `overwrite` was requested but the move fails, any existing destination file is preserved.

See Also: rename_file, copy_file

my_dir

is **Deprecated** (therefore not documented).

Signature:

```
my_dir
```

```
public integer
include filesys.e
namespace filesys
```

pathinfo

parses a fully qualified pathname.

Signature:

```
pathinfo(sequence path, integer std_slash = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : a sequence, the path to parse

Returns: A **sequence**, of length 5. Each of these elements is a string:
• The path name. For Windows, this excludes the drive id.
• The full unqualified file name
• the file name, without extension
• the file extension
• the drive id

Comments: The host operating system path separator is used in the parsing.

See Also: driveid, dirname, filename, fileext, PATH_BASENAME, PATH_DIR, PATH_DRIVEID, PATH_FILEEXT, PATH_FILENAME

Example 1:

```
-- WINDOWS
info = pathinfo("C:\\euphoria\\docs\\readme.txt")
-- info is {"C:\\euphoria\\docs", "readme.txt", "readme", "txt", "C"}
```

Example 2:

```
-- Unix variants
info = pathinfo("/opt/euphoria/docs/readme.txt")
-- info is {"/opt/euphoria/docs", "readme.txt", "readme", "txt", ""}
```

Example 3:

```
-- no extension
info = pathinfo("/opt/euphoria/docs/readme")
-- info is {"/opt/euphoria/docs", "readme", "readme", "", ""}
```

pathname

returns the directory name of a fully qualified filename.

Signature:

```
pathname(sequence path)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path` : the path from which to extract information
≡ `pcd` : If not zero and there is no directory name in `path` then "." is returned. The default (0) will just return any directory name in `path`.

Returns: A **sequence**, the full file name part of `path`.

Comments: The host operating system path separator is used.

See Also: driveid, filename, pathinfo

Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

remove_directory

removes a directory.

Signature:

```
remove_directory(sequence dir_name, integer force = 0)
```

```
public function
include filesys.e
```

```
namespace filesys
```

Arguments: ≡ `name` : a sequence, the name of the directory to remove.
≡ `force` : an integer, if 1 this will also remove files and sub-directories in the directory. The default is 0, which means that it will only remove the directory if it is already empty.

Returns: An **integer**, 0 on failure, 1 on success.

See Also: create_directory, chdir, clear_directory

Example 1:

```
 if not remove_directory("the_old_folder") then
  crash("Filesystem problem - could not remove the old folder")
 end if
```

rename_file

renames a file.

Signature:

```
rename_file(sequence old_name, sequence new_name, integer overwrite = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `old_name` : a sequence, the name of the file or directory to rename.
≡ `new_name` : a sequence, the new name for the renamed file
≡ `overwrite` : an integer, 0 (the default) to prevent renaming if destination file exists, 1 to delete existing destination file first

Returns: An **integer**, 0 on failure, 1 on success.

Comments: * If `new_name` contains a path specification, this is equivalent to moving the file, as well as possibly changing its name. However, the path must be on the same drive for this to work.
• If `overwrite` was requested but the rename fails, any existing destination file is preserved.

See Also: move_file, copy_file

split_path

splits a filename into path segments.

Signature:

```
split_path(sequence fname)
```

```
public function
include filesys.e
namespace filesys
```

Arguments:

• `fname` - Filename to split

Returns: A sequence of strings representing each path element found in `fname`.

See Also: join_path

Example 1:

```
 sequence path_elements = split_path("/usr/home/john/hello.txt")
 -- path_elements would be { "usr", "home", "john", "hello.txt" }
```

temp_file

returns a file name that can be used as a temporary file.

Signature:

```
temp_file(sequence temp_location = "", sequence temp_prefix = "",
sequence temp_extn = "_T_", integer reserve_temp = 0)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `temp_location` : A sequence. A directory where the temporary file is expected to be created.
…… ♦ If omitted (the default) the 'temporary' directory will be used. The temporary directory is defined in the "TEMP" environment symbol, or failing that the "TMP" symbol and failing that "C:\TEMP\" is used in non-Unix systems and "/tmp/" is used in Unix systems.
…… ♦ If `temp_location` was supplied, *** If it is an existing file, that file's directory is used. *** If it is an existing directory, it is used.

Returns: A **sequence**, A generated file name.

Example 1:

```
temp_file("/usr/space", "myapp", "tmp") --> /usr/space/myapp736321.tmp
temp_file() --> /tmp/277382._T_
temp_file("/users/me/abc.exw") --> /users/me/992831._T_
```

walk_dir

is a generalized directory walker.

Signature:

```
walk_dir(sequence path_name, object your_function,
integer scan_subdirs = types :FALSE,
object dir_source = types :NO_ROUTINE_ID)
```

```
public function
include filesys.e
namespace filesys
```

Arguments: ≡ `path_name` : a sequence, the name of the directory to walk through
≡ `your_function` : the routine id of a function that will receive each path returned from the result of `dir_source`, one at a time.
≡ `scan_subdirs` : an optional integer, 1 to also walk though subfolders, 0 (the default) to skip them all.
≡ `dir_source` : an optional integer. A routine_id of a user-defined routine that returns the list of paths to pass to `your_function`. If omitted, the dir() function is used.

Returns: An **object**,
• 0 on success
• W_BAD_PATH : an error occurred
• anything else : the custom function returned something to stop walk_dir().

Comments: This routine will "walk" through a directory named `path_name`. For each entry in the directory, it will call a function, whose routine_id is `your_function`. If `scan_subdirs` is non-zero (TRUE), then the subdirectories in `path_name` will be walked through recursively in the very same way.

The routine that you supply should accept two sequences, the path name and dir() entry for each file and subdirectory. It should return 0 to keep going, W_SKIP_DIRECTORY to avoid scan the contents of the supplied path name (if a directory), or non-zero to stop `walk_dir`(). Returning `W_BAD_PATH` is taken as denoting some error.

This mechanism allows you to write a simple function that handles one file at a time, while `walk_dir()` handles the process of walking through all the files and subdirectories.

By default, the files and subdirectories will be visited in alphabetical order. To use a different order, use the `dir_source` to pass the routine_id of your own modified dir function that sorts the directory entries differently.

The path that you supply to `walk_dir()` must not contain wildcards (* or ?). Only a single directory (and its subdirectories) can be searched at one time.

For non-unix systems, any '/' characters in `path_name` are replaced with '\'.

All trailing slash and whitespace characters are removed from `path_name`.

See Also: dir, sort, sort_columns

Example 1:

```
function look_at(sequence path_name, sequence item)
-- this function accepts two sequences as arguments
-- it displays all C/C++ source files and their sizes
    if find('d', item[D_ATTRIBUTES]) then
        -- Ignore directories
        if find('s', item[D_ATTRIBUTES]) then
           return W_SKIP_DIRECTORY -- Don't recurse a system directory
        else
           return 0 -- Keep processing as normal
        end if
    end if
    if not find(fileext(item[D_NAME]), {"c","h","cpp","hpp","cp"}) then
        return 0 -- ignore non-C/C++ files
    end if
    printf(STDOUT, "%s%s%s: %d\n",
           {path_name, {SLASH}, item[D_NAME], item[D_SIZE]})
    return 0 -- keep going
end function

function mysort(sequence path)
 object d

 d = dir(path)
 if atom(d) then
  return d
 end if
 -- Sort in descending file size.
 return sort_columns(d, {-D_SIZE})
end function

exit_code = walk_dir("C:\\MYFILES\\", routine_id("look_at"), TRUE,
                      routine_id("mysort") )
```

flags

Routines

which_bit
flags_to_string

flags API

flags_to_string

returns a list of strings that represent the human-readable identities of the supplied flag or flags.

Signature:

```
flags_to_string(object flag_bits, sequence flag_names,
integer expand_flags = 0)
```

```
public function
include flags.e
namespace flags
```

Arguments: ≡ `flag_bits` : Either a single 32-bit set of flags (a flag value), or a list of such flag values. The function returns the names for these flag values.
≡ `flag_names` : A sequence of two-element sub-sequences. Each sub-sequence is contains {flag value, flag name}, where *flag name* is a string and *flag value* is the set of bits that set the flag on.
≡ `expand_flags`: An integer. 0 (the default) means that the flag values in `flag_bits` are not broken down to their single-bit values. eg. #0c returns the name of #0c and not the names for #08 and #04. When `expand_flags` is non-zero then each bit in the `flag_bits` parameter is scanned for a matching name.

Returns: A sequence. This contains the name(s) for each supplied flag value(s).

Comments:

• The number of strings in the returned value depends on `expand_flags` is non-zero and whether `flags_bits` is an atom or sequence.
• When `flag_bits` is an atom, you get returned a sequence of strings, one for each matching name (according to `expand_flags` option).
• When `flag_bits` is a sequence, it is assumed to represent a list of atomic flags. That is, {#1, #4} is a set of two flags for which you want their names. In this case, you get returned a sequence that contains one sequence for each element in `flag_bits`, which in turn contain the matching name(s).
• When a flag's name can not be found in `flag_names`, this function returns the *name* of "`?`".

Example 1:

```
include std/console.e
sequence s
s = {
 {#00000000, "WS_OVERLAPPED"},
 {#80000000, "WS_POPUP"},
 {#40000000, "WS_CHILD"},
 {#20000000, "WS_MINIMIZE"},
 {#10000000, "WS_VISIBLE"},
 {#08000000, "WS_DISABLED"},
 {#44000000, "WS_CLIPPINGCHILD"},
 {#04000000, "WS_CLIPSIBLINGS"},
 {#02000000, "WS_CLIPCHILDREN"},
 {#01000000, "WS_MAXIMIZE"},
 {#00C00000, "WS_CAPTION"},
 {#00800000, "WS_BORDER"},
 {#00400000, "WS_DLGFRAME"},
 {#00100000, "WS_HSCROLL"},
 {#00200000, "WS_VSCROLL"},
 {#00080000, "WS_SYSMENU"},
 {#00040000, "WS_THICKFRAME"},
 {#00020000, "WS_MINIMIZEBOX"},
 {#00010000, "WS_MAXIMIZEBOX"},
 {#00300000, "WS_SCROLLBARS"},
 {#00CF0000, "WS_OVERLAPPEDWINDOW"},
 $
}
display( flags_to_string( {#0C20000,2,9,0}, s,1))
--> {
-->     "WS_BORDER",
-->     "WS_DLGFRAME",
-->     "WS_MINIMIZEBOX"
-->   },
-->   {
-->     "?"
-->   },
-->   {
-->     "?"
-->   },
-->   {
-->     "WS_OVERLAPPED"
-->   }
--> }
display( flags_to_string( #80000000, s))
--> {
-->   "WS_POPUP"
--> }
display( flags_to_string( #00C00000, s))
--> {
-->   "WS_CAPTION"
--> }
display( flags_to_string( #44000000, s))
--> {
-->   "WS_CLIPPINGCHILD"
--> }
display( flags_to_string( #44000000, s, 1))
--> {
-->   "WS_CHILD",
-->   "WS_CLIPSIBLINGS"
--> }
display( flags_to_string( #00000000, s))
--> {
-->   "WS_OVERLAPPED"
--> }
display( flags_to_string( #00CF0000, s))
--> {
-->   "WS_OVERLAPPEDWINDOW"
--> }
display( flags_to_string( #00CF0000, s, 1))
--> {
-->   "WS_BORDER",
-->   "WS_DLGFRAME",
-->   "WS_SYSMENU",
-->   "WS_THICKFRAME",
-->   "WS_MINIMIZEBOX",
-->   "WS_MAXIMIZEBOX"
--> }
```

which_bit

tests if the supplied value has only a single bit on in its representation.

Signature:

```
which_bit(object theValue)
```

```
public function
include flags.e
namespace flags
```

Arguments: ≡ `theValue` : an object to test.

Returns: An **integer**, either 0 if it contains multiple bits, zero bits or is an invalid value, otherwise the bit number set. The right-most bit is position 1 and the leftmost bit is position 32.

Example 1:

```
? which_bit(2) --> 2
? which_bit(0) --> 0
? which_bit(3) --> 0
? which_bit(4)          --> 3
? which_bit(17)         --> 0
? which_bit(1.7)        --> 0
? which_bit(-2)         --> 0
? which_bit("one")      --> 0
? which_bit(0x80000000) --> 32
```

---

get

Error Status Constants
- GET_SUCCESS
- GET_EOF
- GET_FAIL
- GET_NOTHING

Answer Types
- GET_SHORT_ANSWER
- GET_LONG_ANSWER

Routines
- get
- value
- defaulted_value

Error Status Constants These are returned from get and value.

---

get API

GET_EOF

Signature:

```
GET_EOF
```

```
public constant
include get.e
namespace stdget
```

GET_FAIL

Signature:

```
GET_FAIL
```

```
public constant
include get.e
namespace stdget
```

GET_LONG_ANSWER

Signature:

```
GET_LONG_ANSWER
```

```
public constant
include get.e
namespace stdget
```

GET_NOTHING

Signature:

```
GET_NOTHING
```

```
public constant
include get.e
namespace stdget
```

GET_SHORT_ANSWER

Signature:

```
GET_SHORT_ANSWER
```

```
public constant
include get.e
namespace stdget
```

GET_SUCCESS

Signature:

```
GET_SUCCESS
```

```
public constant
include get.e
namespace stdget
```

defaulted_value

calls `value` and returns the input object on success or the default object on failure.

Signature:

```
defaulted_value(object st, object def, integer start_point = 1)
```

```
public function
include get.e
namespace stdget
```

Arguments:
≡ `st` : object to retrieve value from.
≡ `def` : the value returned if `st` is an atom or `value(st)` fails.
≡ `start_point` : an integer, the position in `st` at which to start getting the value from. Defaults to 1

Returns:

- If `st`, is an atom then `def` is returned.
- If `value(st)`, call is a success, then `value()[2]`, otherwise it will return the parameter #def#.

See Also: value

Example 1:

```
object i
i = defaulted_value("10", 0)
-- i is 10
```

```
i = defaulted_value("abc", 39)
-- i is 39

i = defaulted_value(12, 42)
-- i is 42

i = defaulted_value("{1,2}", 42)
-- i is {1,2}
```

get

Input, from an open file, a human-readable string of characters representing a Euphoria object.

Signature:

```
get(integer file, integer offset = 0, integer answer = GET_SHORT_ANSWER)
```

```
public function
include get.e
namespace stdget
```

Arguments:
≡ `file` : an integer, the handle to an open file from which to read
≡ `offset` : an integer, an offset to apply to file position before reading. Defaults to 0.
≡ `answer` : an integer, either `GET_SHORT_ANSWER` (the default) or `GET_LONG_ANSWER`.

Returns: A **sequence**, of length 2 (`GET_SHORT_ANSWER`) or 4 (`GET_LONG_ANSWER`), made of

• an integer, the return status. This is any of:
…… ♦ `GET_SUCCESS` -- object was read successfully
…… ♦ `GET_EOF` -- end of file before object was read completely
…… ♦ `GET_FAIL` -- object is not syntactically correct
…… ♦ `GET_NOTHING` -- nothing was read, even a partial object string, before end of input
• an object, the value that was read. This is valid only if return status is `GET_SUCCESS`.
• an integer, the number of characters read. On an error, this is the point at which the error was detected.
• an integer, the amount of initial whitespace read before the first active character was found

Comments: When `answer` is not specified, or explicitly `GET_SHORT_ANSWER`, only the first two elements in the returned sequence are actually returned.

The `GET_NOTHING` return status will not be returned if `answer` is `GET_SHORT_ANSWER`.

`get` can read arbitrarily complicated Euphoria objects. You could have a long sequence of values in braces and separated by commas and comments (for example `{23, {49, 57}, 0.5, -1, 99, 'A', "john"}`). A single call to `get` will read in this entire sequence and return its value as a result, as well as complementary information.

If a nonzero offset is supplied, it is interpreted as an offset to the current file position, and the file will be seek()ed there first.

`get()` returns a 2 or 4 element sequence, like `value()` does:

• a status code (success/error/end of file/no value at all)
• the value just read (meaningful only when the status code is `GET_SUCCESS`) (optionally)
• the total number of characters read
• the amount of initial whitespace read.

Using the default value for answer, or setting it to `GET_SHORT_ANSWER`, returns 2 elements. Setting it to `GET_LONG_ANSWER` causes 4 elements to be returned.

Each call to `get()` picks up where the previous call left off. For instance, a series of 5 calls to `get()` would be needed to read in

"99 5.2 {1, 2, 3} "Hello" -1"

On the sixth and any subsequent call to `get()` you would see a `GET_EOF` status. If you had something like

{1, 2, xxx}

in the input stream you would see a `GET_FAIL` error status because xxx is not a Euphoria object. And seeing

-- something\nBut no value

and the input stream stops right there, you'll receive a status code of `GET_NOTHING`, because nothing but whitespace or comments was read. If you had opted for a short answer, you would get `GET_EOF` instead.

Multiple "top-level" objects in the input stream must be separated from each other with one or more "whitespace" characters (blank, tab, \r or \n). At the very least, a top level number must be followed by a white space from the following object. Whitespace is not necessary *within* a top-level object. Comments, terminated by either '\n' or '\r', are allowed anywhere inside sequences, and ignored if at the top level. A call to `get` will read one entire top-level object, plus possibly one additional (whitespace) character, after a top level number, even though the next object may have an identifiable starting point.

The combination of `print()` and `get()` can be used to save a Euphoria object to disk and later read it back. This technique could be used to implement a database as one or more large Euphoria sequences stored in disk files. The sequences could be read into memory, updated and then written back to disk after each series of transactions is complete. Remember to write out a whitespace character (using `puts()`) after each call to `print()`, at least when a top level number was just printed.

The value returned is not meaningful unless you have a `GET_SUCCESS` status.

See Also: value

Example 1:

```
-- If he types 77.5, get(0) would return:
{GET_SUCCESS, 77.5}

-- whereas gets(0) would return:
"77.5\n"
```

Example 2:

value

reads from a string, a human-readable string of characters representing a Euphoria object.

Signature:

```
value(sequence st, integer start_point = 1,
integer answer = GET_SHORT_ANSWER)
```

```
public function
include get.e
namespace stdget
```

Arguments: ≡ `st` : a sequence, from which to read text
≡ `offset` : an integer, the position at which to start reading. Defaults to 1.
≡ `answer` : an integer, either GET_SHORT_ANSWER (the default) or GET_LONG_ANSWER.

Returns: A **sequence**, of length 2 (GET_SHORT_ANSWER) or 4 (GET_LONG_ANSWER), made of

• an integer, the return status. This is any of
…… ♦ `GET_SUCCESS` -- object was read successfully
…… ♦ `GET_EOF` -- end of file before object was read completely
…… ♦ `GET_FAIL` -- object is not syntactically correct
…… ♦ `GET_NOTHING` -- nothing was read, even a partial object string, before end of input
• an object, the value that was read. This is valid only if return status is `GET_SUCCESS`.
• an integer, the number of characters read. On an error, this is the point at which the error was detected.
• an integer, the amount of initial whitespace read before the first active character was found

Comments: When `answer` is not specified, or explicitly `GET_SHORT_ANSWER`, only the first two elements in the returned sequence are actually returned.

This works the same as get(), but it reads from a string that you supply, rather than from a file or device.

After reading one valid representation of a Euphoria object, `value()` will stop reading and ignore any additional characters in the string. For example, "36" and "36P" will both give you `{GET_SUCCESS, 36}`.

The function returns `{return_status, value}` if the answer type is not passed or set to `GET_SHORT_ANSWER`. If set to `GET_LONG_ANSWER`, the number of characters read and the amount of leading whitespace are returned in 3rd and 4th position. The `GET_NOTHING` return status can occur only on a long answer.

See Also: get

Example 1:

```
s = value("12345"}
s is {GET_SUCCESS, 12345}
```

Example 2:

```
s = value("{0, 1, -99.9}")
-- s is {GET_SUCCESS, {0, 1, -99.9}}
```

Example 3:

```
s = value("+++")
-- s is {GET_FAIL, 0}
```

---

graphcst

Error Code Constants

BMP_SUCCESS
BMP_OPEN_FAILED
BMP_UNEXPECTED_EOF
BMP_UNSUPPORTED_FORMAT
BMP_INVALID_MODE

video_config sequence accessors	
	VC_COLOR
	VC_MODE
	VC_LINES
	VC_COLUMNS
	VC_XPIXELS
	VC_YPIXELS
	VC_NCOLORS
	VC_PAGES
	VC_SCRNLINES
	VC_SCRNCOLS
Colors	
	BLACK
	BLUE
	GREEN
	CYAN
	RED
	MAGENTA
	BROWN
	WHITE
	GRAY
	BRIGHT_BLUE
	BRIGHT_GREEN
	BRIGHT_CYAN
	BRIGHT_RED
	BRIGHT_MAGENTA
	YELLOW
	BRIGHT_WHITE
	true_fgcolor
	true_bgcolor
	BLINKING
	BYTES_PER_CHAR
	color
Routines	
	mixture
	video_config
	FGSET
	BGSET

graphcst API

BGSET

Signature: Background set of colors

```
BGSET
```

```
public enum
include graphcst.e
namespace graphcst
```

BLACK

Signature:

```
BLACK
```

```
public constant
include graphcst.e
namespace graphcst
```

BLINKING

provides blinking text.

Signature:

```
BLINKING
```

```
public constant
include graphcst.e
namespace graphcst
```

BLUE

Signature:

```
BLUE
```

```
public constant
include graphcst.e
namespace graphcst
```

BMP_INVALID_MODE

Signature:

```
BMP_INVALID_MODE
```

```
public enum
include graphcst.e
namespace graphcst
```

BMP_OPEN_FAILED

Signature:

```
BMP_OPEN_FAILED
```

```
public enum
include graphcst.e
namespace graphcst
```

BMP_SUCCESS

Signature:

```
BMP_SUCCESS
```

```
public enum
include graphcst.e
namespace graphcst
```

BMP_UNEXPECTED_EOF

Signature:

```
BMP_UNEXPECTED_EOF
```

```
public enum
include graphcst.e
namespace graphcst
```

BMP_UNSUPPORTED_FORMAT

Signature:

```
BMP_UNSUPPORTED_FORMAT
```

```
public enum
include graphcst.e
namespace graphcst
```

BRIGHT_BLUE

Signature:

```
BRIGHT_BLUE
```

```
public constant
include graphcst.e
namespace graphcst
```

BRIGHT_CYAN

Signature:

```
BRIGHT_CYAN
```

```
public constant
include graphcst.e
namespace graphcst
```

BRIGHT_GREEN

Signature:

```
BRIGHT_GREEN
```

```
public constant
include graphcst.e
namespace graphcst
```

BRIGHT_MAGENTA

Signature:

```
BRIGHT_MAGENTA
```

```
public constant
include graphcst.e
namespace graphcst
```

BRIGHT_RED

Signature:

```
BRIGHT_RED
```

```
public constant
include graphcst.e
namespace graphcst
```

BRIGHT_WHITE

Signature:

```
BRIGHT_WHITE
```

```
public constant
include graphcst.e
namespace graphcst
```

BROWN

Signature:

```
BROWN
```

```
public constant
include graphcst.e
namespace graphcst
```

BYTES_PER_CHAR

Signature:

```
BYTES_PER_CHAR
```

```
public constant
include graphcst.e
namespace graphcst
```

CYAN

Signature:

```
CYAN
```

```
public constant
include graphcst.e
namespace graphcst
```

FGSET

Foreground (text) set of colors

Signature:

```
FGSET
```

```
public enum
include graphcst.e
namespace graphcst
```

GRAY

Signature:

```
GRAY
```

```
public constant
include graphcst.e
namespace graphcst
```

GREEN

Signature:

```
GREEN
```

```
public constant
include graphcst.e
namespace graphcst
```

MAGENTA

Signature:

```
MAGENTA
```

```
public constant
include graphcst.e
namespace graphcst
```

RED

Signature:

```
RED
```

```
public constant
include graphcst.e
namespace graphcst
```

VC_COLOR

Signature:

```
VC_COLOR
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_COLUMNS

Signature:

```
VC_COLUMNS
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_LINES

Signature:

```
VC_LINES
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_MODE

Signature:

```
VC_MODE
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_NCOLORS

Signature:

```
VC_NCOLORS
```

```
public enum
include graphcst.e
```

```
namespace graphcst
```

VC_PAGES

Signature:

```
VC_PAGES
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_SCRNCOLS

Signature:

```
VC_SCRNCOLS
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_SCRNLINES

Signature:

```
VC_SCRNLINES
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_XPIXELS

Signature:

```
VC_XPIXELS
```

```
public enum
include graphcst.e
namespace graphcst
```

VC_YPIXELS

Signature:

```
VC_YPIXELS
```

```
public enum
include graphcst.e
namespace graphcst
```

WHITE

Signature:

```
WHITE
```

```
public constant
include graphcst.e
namespace graphcst
```

YELLOW

Signature:

```
YELLOW
```

```
public constant
include graphcst.e
namespace graphcst
```

color

Signature:

```
color(object x)
```

```
public type
include graphcst.e
namespace graphcst
```

mixture

is a type for mixtures.

Signature:

```
mixture(object s)
```

```
public type
include graphcst.e
namespace graphcst
```

Comments: A mixture is a `{red, green, blue}` triple of intensities, which enables you to define custom colors. Intensities must be from 0 (weakest) to 63 (strongest). Thus, the brightest white is {63, 63, 63}.

true_bgcolor

Signature:

```
true_bgcolor
```

```
export sequence
include graphcst.e
namespace graphcst
```

true_fgcolor

Signature:

```
true_fgcolor
```

```
export sequence
include graphcst.e
namespace graphcst
```

video_config

returns a description of the current video configuration.

Signature:

```
video_config()
```

```
public function
include graphcst.e
namespace graphcst
```

Returns: A **sequence**, of 10 non-negative integers, laid out as follows: # color monitor? -- 1 0 if monochrome, 1 otherwise # current video mode # number of text rows in console buffer # number of text columns in console buffer # screen width in pixels # screen height in pixels # number of colors # number of display pages # number of text rows for current screen size # number of text columns for current screen size

Comments: A public enum is available for convenient access to the returned configuration data:

- `VC_COLOR`
- `VC_MODE`
- `VC_LINES`
- `VC_COLUMNS`
- `VC_XPIXELS`
- `VC_YPIXELS`
- `VC_NCOLORS`
- `VC_PAGES`
- `VC_SCRNLINES`
- `VC_SCRNCOLS`

This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

See Also: graphics_mode

Color Set Selection

Example 1:

```
vc = video_config()
-- vc could be {1, 3, 300, 132, 0, 0, 32, 8, 37, 90}
```

graphics

Routines

- position
- get_position
- text_color
- bk_color

---

graphics API

---

bk_color

sets the background color to one of the 16 standard colors.

Signature:

```
bk_color(color c)
```

```
public procedure
include graphics.e
namespace graphics
```

Arguments: ≡ `c` : the new text color. Add `BLINKING` to get blinking text in some modes.

Comments: To restore the original background color when your program finishes (for example: `0 -- BLACK`) you must call `bk_color(0)`. If the cursor is at the bottom line of the screen, you may have to actually print something before terminating your program. Printing `'\n'` may be enough.

See Also: text_color

Example 1:

```
 bk_color(BLACK)
```

console_colors

sets the codes for the colors used in text_color and bk_color.

Signature:

```
console_colors(sequence colorset = {})
```

```
public function
include graphics.e
namespace graphics
```

Arguments: ≡ `colorset` : A sequence in one of two formats. `Containing two sets of exactly 16 color numbers in which the first set are foreground (text) colors and the other set are background colors.` Containing a set of exactly sixteen color numbers. These are to be applied to both foreground and background.

Returns: A sequence: This contains two sets of 16 color values currently in use for FG and BG respectively.

Comments:

- If the `colorset` is omitted then this just returns the current values without changing anything.
- A color set contains 16 values. You can access the color value for a specific color by using [X + 1] where 'X' is one of the Euphoria color constants such as `RED`, `BLUE`, etc ...
- This can be used to change the meaning of the standard color codes for some consoles that are not using standard values. For example, the Unix default color value for RED is 1 and BLUE is 4, but you might need this to swapped. See code example 1. Another use might be to suppress highlighted (bold) colors. See code

example 2.

See Also: text_color bk_color

Example 1:

```
sequence cs
cs = console_colors() -- Get the current FG and BG color values.
cs[FGSET][RED + 1] = 4 -- set RED to 4
cs[FGSET][BLUE + 1] = 1 -- set BLUE to 1
cs[BGSET][RED + 1] = 4 -- set RED to 4
cs[BGSET][BLUE + 1] = 1 -- set BLUE to 1
console_colors(cs)
```

Example 2:

```
-- Prevent highlighted background colors
sequence cs
cs = console_colors()
for i = GRAY + 1 to BRIGHT_WHITE + 1 do
   cs[BGSET][i] = cs[BGSET][i - 8]
end for
console_colors(cs)
```

get_position

returns the current line and column position of the cursor .

Signature:

```
get_position()
```

```
public function
include graphics.e
namespace graphics
```

Returns: A **sequence**, `{line, column}`, the current position of the text mode cursor.

Comments: The coordinate system for displaying text is different from the one for displaying pixels. Pixels are displayed such that the top-left is (x=0,y=0) and the first coordinate controls the horizontal, left-right location. In pixel-graphics modes you can display both text and pixels. `get_position`() returns the current line and column for the text that you are displaying, not the pixels that you may be plotting. There is no corresponding routine for getting the current pixel position, because there is not such a thing.

See Also: position

graphics_mode

attempts to set up a new graphics mode.

Signature:

```
graphics_mode(object m = - 1)
```

```
public function
include graphics.e
namespace graphics
```

Arguments: ≡ x : an object, but it will be ignored.

Returns: An **integer**, always returns zero.

Comments:

- This has no effect on *unix* platforms.
- On *windows* it causes a console to be shown if one has not already been created.

See Also: video_config

position

Signature:

```
position(integer row, integer column)
```

```
<built-in> procedure
```

Arguments: ≡ `row` : an integer, the index of the row to position the cursor on.
≡ `column` : an integer, the index of the column to position the cursor on.

Comments: Set the cursor to line `row`, column `column`, where the top left corner of the screen is line 1, column 1. The next character displayed on the screen will be printed at this location. `position`() will report an error if the location is off the screen. The *windows* console does not check for rows, as the physical height of the console may be vastly less than its logical height.

See Also: get_position

Example 1:

```
position(2,1)
-- the cursor moves to the beginning of the second line from the top
```

scroll

scrolls a region of text on the screen.

Signature:

```
scroll(integer amount, console :positive_int top_line,
console :positive_int bottom_line)
```

```
public procedure
include graphics.e
namespace graphics
```

Arguments: ≡ `amount` : an integer, the number of lines by which to scroll. This is >0 to scroll up and <0 to scroll down.
≡ `top_line` : the 1-based number of the topmost line to scroll.
≡ `bottom_line` : the 1-based number of the bottom-most line to scroll.

Comments:

• New blank lines will appear at the vacated lines.
• You could perform the scrolling operation using a series of calls to `[:puts]]()`, but `scroll`() is much faster.
• The position of the cursor after scrolling is not defined.

See Also: clear_screen, text_rows

Example 1:

text_color

sets the foreground text color.

Signature:

```
text_color(color c)
```

```
public procedure
include graphics.e
namespace graphics
```

Arguments: ≡ `c` : the new text color. Add `BLINKING` to get blinking text in some modes.

Comments: Text that you print after calling `text_color`() will have the desired color.

When your program terminates, the last color that you selected and actually printed on the screen will remain in effect. Thus you may have to print something, maybe just `'\n'`, in `WHITE` to restore white text, especially if you are at the bottom line of the screen, ready to scroll up.

See Also: bk_color , clear_screen

Example 1:

```
text_color(BRIGHT_BLUE)
```

wrap

determines whether text will wrap when hitting the rightmost column.

Signature:

```
wrap(object on = 1)


public procedure
include graphics.e
namespace graphics
```

Arguments: ≡ `on` : an object, 0 to truncate text, anything else to wrap.

Comments: By default text will wrap.

Use `wrap`() in text modes or pixel-graphics modes when you are displaying long lines of text.

See Also: puts, position

Example 1:

```
puts(1, repeat('x', 100) & "\n\n")
-- now have a line of 80 'x' followed a line of 20 more 'x'
wrap(0)
puts(1, repeat('x', 100) & "\n\n")
-- creates just one line of 80 'x'
```

hash

Type Constants

- HSIEH30
- HSIEH32
- ADLER32
- FLETCHER32
- MD5
- SHA256

Routines

- hash

hash API

ADLER32

Signature:

```
ADLER32
```

```
public enum
include hash.e
namespace stdhash
```

FLETCHER32

Signature:

```
FLETCHER32
```

```
public enum
include hash.e
namespace stdhash
```

HSIEH30

Signature:

```
HSIEH30
```

```
public enum
include hash.e
namespace stdhash
```

HSIEH32

Signature:

```
HSIEH32
```

```
public enum
include hash.e
namespace stdhash
```

MD5

Signature:

```
MD5
```

```
public enum
include hash.e
namespace stdhash
```

SHA256

Signature:

```
SHA256
```

```
public enum
include hash.e
namespace stdhash
```

hash

calculates a hash value from a *key* using the algorithm *algo*.

Signature:

```
hash(object source, atom algo)
```

```
<built-in> function
```

Arguments: ≡ `source` : Any Euphoria object
≡ `algo` : A code indicating which algorithm to use.
…… ♦ HSIEH30 uses Hsieh. Returns a 30-bit (a Euphoria integer). Fast and good dispersion
…… ♦ HSIEH32 uses Hsieh. Returns a 32-bit value. Fast and very good dispersion
…… ♦ ADLER32 uses Adler. Very fast and reasonable dispersion, especially for small strings
…… ♦ FLETCHER32 uses Fletcher. Very fast and good dispersion
…… ♦ MD5 uses MD5 (not implemented yet) Slower but very good dispersion. Suitable for signatures.
…… ♦ SHA256 uses SHA256 (not implemented yet) Slow but excellent dispersion. Suitable for signatures. More secure than MD5.
…… ♦ 0 and above (integers and decimals) and non-integers less than zero use the cyclic variant (hash = hash * algo + c). This is a fast and good to excellent dispersion depending on the value of *algo*. Decimals give better dispersion but are slightly slower.

Returns: An **atom**, Except for the HSIEH30, MD5 and SHA256 algorithms, this is a 32-bit integer.
An **integer**, Except for the HSIEH30 algorithms, this is a 30-bit integer.
A **sequence**, MD5 returns a 4-element sequence of integers
SHA256 returns a 8-element sequence of integers.

Comments:

• For *algo* values from zero to less than 1, that actual value used is (algo + 69096).

Example 1:

```
 ? hash("The quick brown fox jumps over the lazy dog", 0         )
       --> 3071488335
 ? hash("The quick brown fox jumps over the lazy dog", 99        )
       --> 4122557553
 ? hash("The quick brown fox jumps over the lazy dog", 99.94     )
       -->   95918096
 ? hash("The quick brown fox jumps over the lazy dog", -99.94    )
       --> 4175585990
 ? hash("The quick brown fox jumps over the lazy dog", HSIEH30   )
       -->   96435427
 ? hash("The quick brown fox jumps over the lazy dog", HSIEH32   )
        -->   96435427
 ? hash("The quick brown fox jumps over the lazy dog", ADLER32   )
       --> 1541148634
 ? hash("The quick brown fox jumps over the lazy dog", FLETCHER32)
       --> 1730140417
 ? hash(123,                                           99        )
       --> 1188623852
 ? hash(1.23,                                          99        )
       --> 3808916725
 ? hash({1, {2,3, {4,5,6}, 7}, 8.9},                   99        )
       -->  526266621
```

image

image API

graphics_point

Signature:

```
graphics_point(object p)
```

```
public type
include image.e
namespace image
```

read_bitmap

reads a bitmap (.bmp) file into a 2-d sequence of sequences (image)

Signature:

```
read_bitmap(sequence file_name)
```

```
public function
include image.e
namespace image
```

Arguments: ≡ `file_name` : a sequence, the path to a .bmp file to read from. The extension is not assumed if missing.

Returns: An **object**, on success, a sequence of the form `{palette,image}`. On failure, an error code is returned.

Comments: In the returned value, the first element is a list of mixtures, each of which defines a color, and the second, a list of point rows. Each pixel in a row is represented by its color index.

The file should be in the bitmap format. The most common variations of the format are supported.

Bitmaps of 2, 4, 16 or 256 colors are supported.

If the file is not in a good format, an error code (atom) is returned instead:
BMP_OPEN_FAILED = 1,
BMP_UNEXPECTED_EOF = 2,
BMP_UNSUPPORTED_FORMAT = 3

You can create your own bitmap picture files using Windows Paintbrush and many other graphics programs. You can then incorporate these pictures into your Euphoria programs.

See Also: save_bitmap

Example 1:

```
 x = read_bitmap("c:\\windows\\arcade.bmp")
```

```
   -- Write escaped backslash \\ to get a single backslash \ in a string
```

save_bitmap

creates a .BMP bitmap file, given a palette and a 2-d sequence of sequences of colors.

Signature:

```
save_bitmap(two_seq palette_n_image, sequence file_name)
```

```
public function
include image.e
namespace image
```

Arguments: ≡ `palette_n_image` : a {palette, image} pair, like read_bitmap() returns
≡ `file_name` : a sequence, the name of the file to save to.

Returns: An **integer**, 0 on success.

Comments: This routine does the opposite of read_bitmap(). The first element of `palette_n_image` is a sequence of mixtures defining each color in the bitmap. The second element is a sequence of sequences of colors. The inner sequences must have the same length.

The result will be one of the following codes:
BMP_SUCCESS = 0,
BMP_OPEN_FAILED = 1,
BMP_INVALID_MODE = 4

`save_bitmap`() produces bitmaps of 2, 4, 16, or 256 colors and these can all be read with `read_bitmap`(). Windows Paintbrush and some other tools do not support 4-color bitmaps.

See Also: read_bitmap

Example 1:

```
 code = save_bitmap({paletteData, imageData},
                   "c:\\example\\a1.bmp")
```

io

Constants

STDIN
STDOUT
STDERR
SCREEN
EOF

Read and Write Routines

?
print
printf
puts
getc
gets
get_bytes
get_integer32
get_integer16

Read and Write Routines

---

io API

?

"quick print" is shorthand for: `pretty_print(STDOUT, x, {})`.

Signature:

```
?
```

```
<built-in> procedure
```

Comments: This procedure prints the value of an expression to the standard output, using braces and indentation to show the structure.

No parentheses, (), are used to surround the unique parameter.

See Also: print

Example 1:

```
 ? {1, 2} + {3, 4}  -- will display {4, 6}
```

BINARY_MODE

Signature:

```
BINARY_MODE
```

```
public enum
include io.e
namespace io
```

DOS_TEXT

Signature:

```
DOS_TEXT
```

```
public enum
include io.e
namespace io
```

EOF

End of file

Signature:

```
EOF
```

```
public constant
include io.e
namespace io
```

LOCK_EXCLUSIVE

Signature:

```
LOCK_EXCLUSIVE
```

```
public enum
include io.e
namespace io
```

LOCK_SHARED

Signature:

```
LOCK_SHARED
```

```
public enum
include io.e
namespace io
```

SCREEN

Screen (Standard Out)

Signature:

```
SCREEN
```

```
public constant
include io.e
namespace io
```

STDERR

Standard Error

Signature:

```
STDERR
```

```
public constant
include io.e
namespace io
```

STDIN

Standard Input

Signature:

```
STDIN
```

```
public constant
include io.e
namespace io
```

STDOUT

Standard Output

Signature:

```
STDOUT
```

```
public constant
include io.e
namespace io
```

TEXT_MODE

Signature:

```
TEXT_MODE
```

```
public enum
include io.e
namespace io
```

UNIX_TEXT

Signature:

```
UNIX_TEXT
```

```
public enum
include io.e
namespace io
```

append_lines

appends a sequence of lines to a file.

Signature:

```
append_lines(sequence file, sequence lines)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `file` : an object, either a file path or the handle to an open file.
≡ `lines` : the sequence of lines to write

Returns: An **integer**, 1 on success, -1 on failure.

Comments: `file` is opened, written to and then closed.

See Also: write_lines, puts

Example 1:

```
if append_lines("data.txt",{"This is important data","Goodbye"})!=-1
   then puts(STDERR, "Failed to append data\n")
end if
```

byte_range

Byte Range Type

Signature:

```
byte_range(object r)
```

```
public type
include io.e
namespace io
```

close

closes a file (or device) and flushs out any still-buffered characters.

Signature:

```
close(atom fn)
```

```
<built-in> procedure
```

Arguments: ≡ `fn` : an integer, the handle to the file or device to query.

Comments: All open files are closed automatically when your program terminates.

file_number

File number type

Signature:

```
file_number(object f)
```

```
public type
include io.e
namespace io
```

file_position

File position type

Signature:

```
file_position(object p)
```

```
public type
include io.e
namespace io
```

flush

Force writing any buffered data to an open file or device.

Signature:

```
flush(file_number fn)
```

```
public procedure
include io.e
namespace io
```

Arguments: ≡ `fn` : an integer, the handle to the file or device to close.

Returns: An **integer**, 0 on failure, 1 on success.

Comments: When you write data to a file, Euphoria normally stores the data in a memory buffer until a large enough chunk of data has accumulated. This large chunk can then be written to disk very efficiently. Sometimes you may want to force, or flush, all data out immediately, even if the memory buffer is not full. To do this you must call `flush(fn)`, where `fn` is the file number of a file open for writing or appending.

When a file is closed, (see close()), all buffered data is flushed out. When a program terminates, all open files are flushed and closed automatically. Use flush() when another process may need to see all of the data written so far, but you are not ready to close the file yet. flush() is also used in crash routines, where files may not be closed in the cleanest possible way.

See Also: close, crash_routine

When multiple processes can simultaneously access a file, some kind of locking mechanism may be needed to avoid mangling the contents of the file, or causing erroneous data to be read from the file.

Example 1:

```
f = open("file.log", "w")
puts(f, "Record#1\n")
puts(STDOUT, "Press Enter when ready\n")

flush(f)  -- This forces "Record #1" into "file.log" on disk.
          -- Without this, "file.log" will appear to have
          -- 0 characters when we stop for keyboard input.

s = gets(0) -- wait for keyboard input
```

get_bytes

reads the next bytes from a file.

Signature:

```
get_bytes(integer fn, integer n)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fn` : an integer, the handle to an open file to read from.
≡ `n` : a positive integer, the number of bytes to read.

Returns: A **sequence**, of length at most `n`, made of the bytes that could be read from the file.

Comments: When `n` > 0 and the function returns a sequence of length less than `n` you know you have reached the end of file. Eventually, an empty sequence will be returned.

This function is normally used with files opened in binary mode, "rb". This avoids the confusing situation in text mode where *windows* will convert CR LF pairs to LF.

See Also: getc, gets, get_integer32, get_dstring

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

sequence whole_file
whole_file = {}

sequence chunk

while 1 do
    chunk = get_bytes(fn, 100) -- read 100 bytes at a time
    whole_file &= chunk        -- chunk might be empty, that's ok
    if length(chunk) < 100 then
        exit
    end if
end while

close(fn)
? length(whole_file)  -- should match DIR size of "temp"
```

get_dstring

reads a delimited byte string from an opened file .

Signature:

```
get_dstring(integer fh, integer delim = 0)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fh` : an integer, the handle to an open file to read from.
≡ `delim` : an integer, the delimiter that marks the end of a byte string. If omitted, a zero is assumed.

Returns: An **sequence**, made of the bytes that could be read from the file.

Comments:

• If the end-of-file is found before the delimiter, the delimiter is appended to the returned string.

See Also: getc, gets, get_bytes, get_integer32

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

sequence text
text = get_dstring(fn) -- Get a zero-delimited string
text = get_dstring(fn, '$') -- Get a '$'-delimited string
```

get_integer16

read the next two bytes from a file and returns them as a single integer.

Signature:

```
get_integer16(integer fh)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fh` : an integer, the handle to an open file to read from.

Returns: An **integer**, made of the bytes that could be read from the file. When an end of file is encountered, it returns -1.

Comments:

• This function is normally used with files opened in binary mode, "rb".

See Also: getc, gets, get_bytes, get_dstring

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

atom file_type_code
file_type_code = get_integer16(fn)
```

get_integer32

reads the next four bytes from a file and returns them as a single integer.

Signature:

```
get_integer32(integer fh)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fh` : an integer, the handle to an open file to read from.

Returns: An **atom**, between -1 and power(2,32)-1, made of the bytes that could be read from the file. When an end of file is encountered, it returns -1.

Comments:

• This function is normally used with files opened in binary mode, "rb".

See Also: getc, gets, get_bytes, get_dstring

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

atom file_type_code
```

```
 file_type_code = get_integer32(fn)
```

getc

gets the next character (byte) from a file or device fn.

Signature:

```
getc(integer fn)
```

```
<built-in> function
```

Arguments: ≡ `fn` : an integer, the handle of the file or device to read from.

Returns: An **integer**, the character read from the file, in the 0..255 range. If no character is left to read, EOF is returned instead.

Comments: File input using `getc`() is buffered. The function `getc`() does not access the disk for each individual character, but instead reads a large block of characters; individual characters are returned to you, one by one, from a memory buffer.

Keyboard input using `getc`() is also buffered. The function `getc()` will not receive any characters until the user presses Enter. Note that the user can type Control+Z, which the operating system treats as "end of file". EOF will be returned.

See Also: gets, get_key

gets

gets the next sequence (one line, including '\n') of characters from a file or device.

Signature:

```
gets(integer fn)
```

```
<built-in> function
```

Arguments: ≡ `fn` : an integer, the handle of the file or device to read from.

Returns: An **object**, either EOF on end of file, or the next line of text from the file.

Comments: The characters will have values from 0 to 255.

If the line has an end of line marker then a newline character ('\n') terminates the line. The last line of a file does not require an end of line marker and could be missing.

After reading a line of text from the keyboard, you should normally output a `\n` character (for example `puts(1, '\n')`) before printing something. Only on the last line of the screen does the operating system automatically scroll the screen and advance to the next line.

When your program reads from the keyboard, the user can type Control-Z, which the operating system treats as "end of file". EOF will be returned.

See Also: getc, read_lines

Example 1:

```
 sequence buffer
 object line
 integer fn

 -- read a text file into a sequence
 fn = open("my_file.txt", "r")
 if fn = -1 then
     puts(1, "Couldn't open my_file.txt\n")
```

```
        abort(1)
    end if

    buffer = {}
    while 1 do
        line = gets(fn)
        if atom(line) then
            exit   -- EOF is returned at end of file
        end if
        buffer = append(buffer, line)
    end while
```

Example 2:

```
    object line

    puts(1, "What is your name?\n")
    line = gets(0)  -- read standard input (keyboard)
    line = line[1..$-1] -- get rid of \n character at end
    puts(1, '\n')   -- necessary
    puts(1, line & " is a nice name.\n")
```

lock_file

Signature:

```
lock_file(file_number fn, lock_type t, byte_range r = {})
```

```
public function
include io.e
namespace io
```

lock_type

Lock Type

Signature:

```
lock_type(object t)
```

```
public type
include io.e
namespace io
```

open

opens a file (or device) and provides a file number.

Signature:

```
open(sequence path, sequence mode, integer cleanup = 0)
```

```
<built-in> function
```

Arguments:
≡ `path` : a string, the path to the file or device to open.
≡ `mode` : a string, the mode being used o open the file.
≡ `cleanup` : an integer, if 0, then the file must be manually closed by the coder. If 1, then the file will be closed when either the file handle references go to 0, or if called as a parameter to `delete`().

Returns: A small **integer**, -1 on failure, else 0 or more.

Comments: Possible modes are:

- `"r"` -- open text file for reading
- `"rb"` -- open binary file for reading
- `"w"` -- create text file for writing
- `"wb"` -- create binary file for writing
- `"u"` -- open text file for update (reading and writing)
- `"ub"` -- open binary file for update
- `"a"` -- open text file for appending
- `"ab"` -- open binary file for appending

Files opened for read or update must already exist. Files opened for write or append will be created if necessary. A file opened for write will be set to 0 bytes. Output to a file opened for append will start at the end of file.

On *windows*, output to text files will have carriage-return characters automatically added before linefeed characters. On input, these carriage-return characters are removed. A Control-Z character (ASCII 26) will signal an immediate end of file.

I/O to binary files is not modified in any way. Any byte values from 0 to 255 can be read or written. On *unix*, all files are binary files, so "r" mode and "rb" mode are equivalent, as are "w" and "wb", "u" and "ub", and "a" and "ab".

Some typical devices that you can open on *windows* are:

- `"CON"` -- the console (screen)
- `"AUX"` -- the serial auxiliary port
- `"COM1"` -- serial port 1
- `"COM2"` -- serial port 2
- `"PRN"` -- the printer on the parallel port
- `"NUL"` -- a non-existent device that accepts and discards output

Close a file or device when done with it, flushing out any still-buffered characters prior.

On *windows* and *unix* long filenames are fully supported for reading, writing, and creating.

On *windows* be careful not to use the special device identifiers in a file name. Even if you add an extension, they will still refer to the device; for example `CON.TXT`, `CON.DAT`, or `CON.JPG` all refer to the `CON` device and **not** to separate files.

Example 1:

```
integer file_num, file_num95
sequence first_line
constant ERROR = 2

file_num = open("my_file", "r")
if file_num = -1 then
    puts(ERROR, "couldn't open my_file\n")
else
    first_line = gets(file_num)
end if

file_num = open("PRN", "w") -- open printer for output

-- on Windows 95:
file_num95 = open("big_directory_name\\very_long_file_name.abcdefg",
                  "r")
if file_num95 != -1 then
    puts(STDOUT, "it worked!\n")
end if
```

print

writes a *text* representation of an object to a file or device.

Signature:

```
print(integer fn, object x)
```

```
<built-in> procedure
```

Arguments:
≡ `fn` : an integer, the handle of a file or device
≡ `x` : the object to print

Comments: This is not used to write to "binary" files as it only outputs text.

See Also: [[:q_print ?]], puts

Example 1:

```
include std/io.e
print(STDOUT, "ABC")   -- output is:  "{65,66,67}"
puts (STDOUT, "ABC")    -- output is:  "ABC"
print(STDOUT, "65")    -- output is:  "65"
puts (STDOUT, 65)       -- output is:  "A"  (ASCII-65 ==> 'A')
print(STDOUT, 65.1234) -- output is:  "65.1234"
puts (STDOUT, 65.1234)  -- output is:  "A" (Converts to integer first)
```

Example 2:

```
include std/io.e
print(STDOUT, repeat({10,20}, 3)) -- output is: {{10,20},{10,20},{10,20}}
```

printf

prints a value (or several values) embedded in a string sequence containing format specifiers.

Signature:

```
printf(integer fn, sequence format, object values)
```

```
<built-in> procedure
```

Arguments:
≡ `fn` : an integer, the handle to a file or device to output to
≡ `format` : a sequence, the text to print. This text may contain format specifiers.
≡ `values` : usually, a sequence of values. It should have as many elements as format specifiers in `format`, as these values will be substituted to the specifiers.

Comments: A *format specifier* is a string of characters starting with a percent sign (%) and ending in a letter. Some extra information may come in between those.

This procedure writes out the `format` text to the output file `fn`, replacing format specifiers with the corresponding data from the `values` parameter. Whenever a format specifiers is found in `formatt`, the *N-th* item in `values` will be turned into a string according to the format specifier. The resulting string will the format specifier. This means that the first format specifier uses the first item in `values`, the second format specifier the second item, and so on.

You must have at least as many items in `values` as there are format specifiers in `format`. This means that if there is only one format specifier then `values` can be either an atom, integer or a non-empty sequence. And when there are more than one format specifier in `format` then `values` must be a sequence with a length that is greater than or equal to the number of format specifiers present.

This way, `printf`() always takes exactly three arguments, no matter how many values are to be printed.

The basic format specifiers are:

- `%d` -- print an atom as a decimal integer
- `%x` -- print an atom as a hexadecimal integer. Negative numbers are printed in two's complement, so -1 will print as FFFFFFFF
- `%o` -- print an atom as an octal integer
- `%s` -- print a sequence as a string of characters, or print an atom as a single character
- `%e` -- print an atom as a floating-point number with exponential notation
- `%f` -- print an atom as a floating-point number with a decimal point but no exponent
- `%g` -- print an atom as a floating-point number using whichever format seems appropriate, given the magnitude of the number
- `%%` -- print a literal '%' character; this is not an actual format specifier.

Field widths can be added to the basic formats (for example %5d, %8.2f, %10.4s). The number before the decimal point is the minimum field width to be used. The number after the decimal point is the precision to be used for numeric values.

If the field width is negative (for example %-5d) then the value will be left-justified within the field. Normally it will be right-justified, even for strings. If the field width starts with a leading zero (for example %08d) then leading zeros will be supplied to fill up the field. If the field width starts with a plus symbol, '+', (for example %+7d) then a plus symbol will be printed for positive values.

Example 1:

```
include std/io.e
sequence name="John Smith"
printf(STDOUT, "My name is %s", name)
```

The output of this will be *`My name is J`* because each format specifier uses exactly **one** item from the `values` parameter. In this case we have only one specifier so it uses the first item in the `values` parameter, which is the character 'J'. To fix this situation, you must ensure that the first item in the `values` parameter is the entire text string and not just a character, so you need code this instead:

```
include std/io.e
name="John Smith"
printf(STDOUT, "My name is %s", {name})
```

Now, the third argument of `printf`() is a one-element sequence containing all the text to be formatted.

Also note that if there is only one format specifier then `values` can simply be an atom or integer.

Example 2:

```
include std/io.e
atom rate = 7.875
printf(STDOUT, "The interest rate is: %8.2f\n", rate)

--      The interest rate is:     7.88
```

Example 3:

```
include std/io.e
sequence name="John Smith"
integer score=97
printf(STDOUT, "%15s, %5d\n", {name, score})

-- "     John Smith,    97"
```

Example 4:

```
include std/io.e
printf(STDOUT, "%-10.4s $ %s", {"ABCDEFGHIJKLMNOP", "XXX"})
--      ABCD       $ XXX
```

Example 4:

```
include std/io.e
printf(STDOUT, "%d  %e  %f  %g", repeat(7.75, 4))
                  -- same value in different formats

--      7  7.750000e+000  7.750000  7.75
```

Example 5:

```
include std/io.e
sequence name = {"John", "Smith"}
printf(STDOUT, "%s", {name} )
```

process_lines

processes the contents of a file, one line at a time.

Signature:

```
process_lines(object file, integer proc, object user_data = 0)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `file` : an object. Either a file path or the handle to an open file. An empty string signifies STDIN - the console keyboard.
≡ `proc` : an integer. The routine_id of a function that will process the line.
≡ `user_data` : on object. This is passed untouched to `proc` for each line.

Returns: An object. If 0 then all the file was processed successfully. Anything else means that something went wrong and this is whatever value was returned by `proc`.

Comments:
• The function `proc` must accept three parameters ...
…… ♦ A sequence: The line to process. It will **not** contain an end-of-line character.
…… ♦ An integer: The line number.
…… ♦ An object : This is the `user_data` that was passed to `process_lines`.
• If `file` was a sequence, the file will be closed on completion. Otherwise, it will remain open, and be positioned where ever reading stopped.

See Also: gets, read_lines, read_file

Example 1:

```
-- Format each supplied line according to the format pattern supplied as well.
function show(sequence aLine, integer line_no, object data)
  writefln( data[1], {line_no, aLine})
  if data[2] > 0 and line_no = data[2] then
   return 1
  else
   return 0
  end if
end function
-- Show the first 20 lines.
process_lines("sample.txt", routine_id("show"), {"[1z:4] : [2]", 20})
```

put_integer16

writes the supplied integer as two bytes to a file.

Signature:

```
put_integer16(integer fh, atom val)
```

```
public procedure
include io.e
```

```
namespace io
```

Arguments: ≡ `fh` : an integer, the handle to an open file to write to.
≡ `val` : an integer

Comments:

• This function is normally used with files opened in binary mode, "wb".

See Also: getc, gets, get_bytes, get_dstring

Example 1:

```
 integer fn
 fn = open("temp", "wb")

 put_integer16(fn, 1234)
```

put_integer32

writes the supplied integer as four bytes to a file.

Signature:

```
put_integer32(integer fh, atom val)
```

```
public procedure
include io.e
namespace io
```

Arguments: ≡ `fh` : an integer, the handle to an open file to write to.
≡ `val` : an integer

Comments:

• This function is normally used with files opened in binary mode, "wb".

See Also: getc, gets, get_bytes, get_dstring

Example 1:

```
 integer fn
 fn = open("temp", "wb")

 put_integer32(fn, 1234)
```

puts

outputs to a file or device, a single byte (atom) or sequence of bytes. The low order

Signature:

```
puts(integer fn, object text)
```

```
<built-in> procedure
```

Arguments: ≡ `fn` : an integer, the handle to an opened file or device
≡ `text` : an object, either a single character or a sequence of characters.

Comments: When you output a sequence of bytes it must not have any (sub)sequences within it. It must be a sequence of atoms only. (Typically a string of ASCII codes).

Avoid outputting 0's to the screen or to standard output. Your output might get truncated.

Remember that if the output file was opened in text mode, *windows* will change `\n` (10) to `\r\n` (13 10). Open the file in binary mode if this is not what you want.

See Also: print

Example 1:

```
include std/io.e
puts(SCREEN, "Enter your first name: ")
```

Example 2:

```
puts(output, 'A')  -- the single byte 65 will be sent to output
```

read_file

reads the contents of a file as a single sequence of bytes.

Signature:

```
read_file(object file, integer as_text = BINARY_MODE)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `file` : an object, either a file path or the handle to an open file.
≡ `as_text` : integer, **BINARY_MODE** (the default) assumes *binary mode* that causes every byte to be read in, and **TEXT_MODE** assumes *text mode* that ensures that lines end with just a Control-J (new line) character, and the first byte value of 26 (Control-Z) is interpreted as End-Of-File.

Returns: A **sequence**, holding the entire file.

Comments
• When using BINARY_MODE, each byte in the file is returned as an element in the return sequence.
• When not using BINARY_MODE, the file will be interpreted as a text file. This means that all line endings will be transformed to a single 0x0A character and the first 0x1A character (Control-Z) will indicate the end of file (all data after this will not be returned to the caller.)

See Also: write_file, read_lines

Example 1:

```
data = read_file("my_file.txt")
-- data contains the entire contents of ##my_file.txt##
```

Example 2:

```
fh = open("my_file.txt", "r")
data = read_file(fh)
close(fh)

-- data contains the entire contents of ##my_file.txt##
```

read_lines

reads the contents of a file as a sequence of lines.

Signature:

```
read_lines(object file)
```

```
public function
include io.e
namespace io
```

Arguments: `file` : an object, either a file path or the handle to an open file. If this is an empty string, STDIN (the console) is used.

Returns: -1 on error or a **sequence**, made of lines from the file, as gets could read them.

Comments: If `file` was a sequence, the file will be closed on completion. Otherwise, it will remain open, but at end of file.

See Also: gets, write_lines, read_file

Example 1:

```
data = read_lines("my_file.txt")
-- data contains the entire contents of ##my_file.txt##, 1 sequence per line:
-- {"Line 1", "Line 2", "Line 3"}
```

Example 2:

```
fh = open("my_file.txt", "r")
data = read_lines(fh)
close(fh)

-- data contains the entire contents of ##my_file.txt##, 1 sequence per line:
-- {"Line 1", "Line 2", "Line 3"}
```

seek

seeks (moves) to any byte position in a file.

Signature:

```
seek(file_number fn, file_position pos)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fn` : an integer, the handle to the file or device to seek()
≡ `pos` : an atom, either an absolute 0-based position or -1 to seek to end of file.

Returns: An **integer**, 0 on success, 1 on failure.

Comments: For each open file, there is a current byte position that is updated as a result of I/O operations on the file. The initial file position is 0 for files opened for read, write or update. The initial position is the end of file for files opened for append. It is possible to seek past the end of a file. If you seek past the end of the file, and write some data, undefined bytes will be inserted into the gap between the original end of file and your new data.

After seeking and reading (writing) a series of bytes, you may need to call seek() explicitly before you switch to writing (reading) bytes, even though the file position should already be what you want.

This function is normally used with files opened in binary mode. In text mode, Windows converts CR LF to LF on input, and LF to CR LF on output, which can cause great confusion when you are trying to count bytes because seek() counts the *windows* end of line sequences as two bytes, even if the file has been opened in text mode.

See Also: get_bytes, puts, where

Example 1:

```
include std/io.e

integer fn
fn = open("my.data", "rb")
-- read and display first line of file 3 times:
for i = 1 to 3 do
    puts(STDOUT, gets(fn))
    if seek(fn, 0) then
        puts(STDOUT, "rewind failed!\n")
    end if
end for
```

unlock_file

unlocks (a portion of) an open file.

Signature:

```
unlock_file(file_number fn, byte_range r = {})
```

```
public procedure
include io.e
namespace io
```

Arguments: ≡ `fn` : an integer, the handle to the file or device to (partially) lock.
≡ `r` : a sequence, defining a section of the file to be locked, or {} for the whole file (the default).

Comments: You must have previously locked the file using `lock_file`(). On *windows* you can unlock a range of bytes within a file by specifying the `r` as {first_byte, last_byte}. The same range of bytes must have been locked by a previous call to lock_file(). On *unix* you can currently only lock or unlock an entire file. `r` should be `{}` when you want to unlock an entire file. On *unix*, `r` must always be `{}`, which is the default.

You should unlock a file as soon as possible so other processes can use it.

Any files that you have locked, will automatically be unlocked when your program terminates.

See Also: lock_file

where

retrieves the current file position for an opened file or device.

Signature:

```
where(file_number fn)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `fn` : an integer, the handle to the file or device to query.

Returns: An **atom**, the current byte position in the file.

Comments: The file position is is the place in the file where the next byte will be read from, or written to. It is updated by reads, writes and seeks on the file. This procedure always counts *windows* end of line sequences (CR LF) as two bytes even when the file number has been opened in text mode.

write_file

writes a sequence of bytes to a file.

Signature:

```
write_file(object file, sequence data, integer as_text = BINARY_MODE)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `file` : an object, either a file path or the handle to an open file.
≡ `data` : the sequence of bytes to write

≡ `as_text` : integer
…… ♦ **BINARY_MODE** (the default) assumes *binary mode* that causes every byte to be written out as is,
…… ♦ **TEXT_MODE** assumes *text mode* that causes a new line to be written out according to the operating system end of line convention. On *unix* this is Control-J and on *windows* this is the pair {Control-L, Control-J}.
…… ♦ **UNIX_TEXT** ensures that lines are written out with unix-style line endings (Control-J).
…… ♦ **DOS_TEXT** ensures that lines are written out with windows-style line endings {Control-L, Control-J}.

Returns: An **integer**, 1 on success, -1 on failure.

Comments:

• When `file` is a file handle, the file is not closed after writing is finished. When `file` is a file name, it is opened, written to and then closed.
• Note that when writing the file in ony of the text modes, the file is truncated at the first Control-Z character in the input data.

See Also: read_file, write_lines

Example 1:

```
if write_file("data.txt", "This is important data\nGoodbye") = -1 then
    puts(STDERR, "Failed to write data\n")
end if
```

write_lines

writes a sequence of lines to a file.

Signature:

```
write_lines(object file, sequence lines)
```

```
public function
include io.e
namespace io
```

Arguments: ≡ `file` : an object, either a file path or the handle to an open file.
≡ `lines` : the sequence of lines to write

Returns: An **integer**, 1 on success, -1 on failure.

Comments: If `file` was a sequence, the file will be closed on completion. Otherwise, it will remain open, but at end of file.

Whatever integer the lines in `lines` holds will be truncated to its 8 lowest bits so as to fall in the 0.255 range.

See Also: read_lines, write_file, puts

Example 1:

```
if write_lines("data.txt",{"This is important data","Goodbye"}) !=-1 then
    puts(STDERR, "Failed to write data\n")
end if
```

writef

writes formatted text to a file.

Signature:

```
writef(object fm, object data = {}, object fn = 1,
object data_not_string = 0)
```

```
public procedure
include io.e
```

```
namespace io
```

Arguments: There are two ways to pass arguments to this function, # Traditional way with first arg being a file handle. `: integer, The file handle.` : sequence, The format pattern. `: object, The data that will be formatted. #`
≡ data_not_string`: object, If not 0 then the` data `is not a string. By default this is 0 meaning that` data `could be a single string. # Alternative way with first argument being the format pattern.` : sequence, Format pattern. `: sequence, The data that will be formatted,` : object, The file to receive the formatted output. Default is to the STDOUT device (console). #
≡ `data_not_string`: object, If not 0 then the `data` is not a string. By default this is 0 meaning that `data` could be a single string.

Comments:

• With the traditional arguments, the first argument must be an integer file handle.
• With the alternative arguments, the thrid argument can be a file name string, in which case it is opened for output, written to and then closed.
• With the alternative arguments, the third argument can be a two-element sequence containing a file name string and an output type ("a" for append, "w" for write), in which case it is opened accordingly, written to and then closed.
• With the alternative arguments, the third argument can a file handle, in which case it is written to only
• The format pattern uses the formatting codes defined in text:format.
• When the data to be formatted is a single text string, it does not have to be enclosed in braces,

See Also: text:format, writefln, write_lines

Example 1:

```
 -- To console
 writef("Today is [4], [u2:3] [3:02], [1:4].",
        {Year, MonthName, Day, DayName})
 -- To "sample.txt"
 writef("Today is [4], [u2:3] [3:02], [1:4].",
        {Year, MonthName, Day, DayName}, "sample.txt")
 -- To "sample.dat"
 integer dat = open("sample.dat", "w")
 writef("Today is [4], [u2:3] [3:02], [1:4].",
        {Year, MonthName, Day, DayName}, dat)
 -- Appended to "sample.log"
 writef("Today is [4], [u2:3] [3:02], [1:4].",
        {Year, MonthName, Day, DayName}, {"sample.log", "a"})
 -- Simple message to console
 writef("A message")
 -- Another console message
 writef(STDERR, "This is a []", "message")
 -- Outputs two numbers
 writef(STDERR, "First [], second []", {65, 100},, 1)
      -- Note that {65, 100} is also "Ad"
```

writefln

writes formatted text to a file, ensuring that a new line is also output.

Signature:

```
writefln(object fm, object data = {}, object fn = 1,
object data_not_string = 0)


public procedure
include io.e
namespace io
```

Arguments: ≡ `fm` : sequence, Format pattern.
≡ `data` : sequence, The data that will be formatted,
≡ `fn` : object, The file to receive the formatted output. Default is to the STDOUT device (console).

≡ `data_not_string`: object, If not 0 then the `data` is not a string. By default this is 0 meaning that `data` could be a single string.

Comments:

- This is the same as writef, except that it always adds a New Line to the output.
- When `fn` is a file name string, it is opened for output, written to and then closed.
- When `fn` is a two-element sequence containing a file name string and an output type ("a" for append, "w" for write), it is opened accordingly, written to and then closed.
- When `fn` is a file handle, it is written to only
- The `fm` uses the formatting codes defined in text:format.

See Also: text:format, writef, write_lines

Example 1:

```
-- To console
writefln("Today is [4], [u2:3] [3:02], [1:4].",
         {Year, MonthName, Day, DayName})
-- To "sample.txt"
writefln("Today is [4], [u2:3] [3:02], [1:4].",
         {Year, MonthName, Day, DayName}, "sample.txt")
-- Appended to "sample.log"
writefln("Today is [4], [u2:3] [3:02], [1:4].",
         {Year, MonthName, Day, DayName}, {"sample.log", "a"})
```

lcid

lcid API

get_lcid

returns the current locale context identifier.

Signature:

```
get_lcid(sequence name)
```

```
public function
include lcid.e
namespace lcid
```

Returns: # The `lcid` which is of type LCID in C/C++.

lcid

is the type for a locale context.

Signature:

```
lcid(object x)
```

```
public type
include lcid.e
namespace lcid
```

locale

locale API

datetime

formats a date according to current locale.

Signature:

```
datetime(sequence fmt, datetime :datetime dtm)


public function
include locale.e
namespace locale
```

Arguments:
≡ `fmt` : A format string, as described in datetime:format
≡ `dtm` : the datetime to write out.

Returns: A **sequence**, representing the formatted date.

See Also: datetime:format

Example 1:

```
include std/datetime.e

datetime("Today is a %A", datetime:now())
```

get

gets the current locale string.

Signature:

```
get()


public function
include locale.e
namespace locale
```

Returns: A **sequence**, a locale string.

See Also: set

get_def_lang

Signature:

gets the default language (translation) map.

```
get_def_lang()
```

```
public function
include locale.e
namespace locale
```

Returns: An **object**, a language map, or zero if there is no default language map yet.

Example 1:

```
 object langmap = get_def_lang()
```

get_lang_path

Signature:

gets the language path.

```
get_lang_path()
```

```
public function
include locale.e
namespace locale
```

Returns: An **object**, the current language path.

See Also: get_lang_path

get_text

Signature:

gets the text associated with the message number in the requested locale.

```
get_text(integer MsgNum, sequence LocalQuals = {},
sequence DBBase = "teksto")
```

```
public function
include locale.e
namespace locale
```

Arguments:
≡ `msg_num` : An integer. The message number whose text you are trying to get.
≡ `local_quals` : A sequence. Zero or more locale codes. Default is {}.
≡ `DBBase`: A sequence. The base name for the database files containing the locale text strings. The default is "teksto".

Returns: A string **sequence**, the text associated with the message number and locale.
The **integer** zero, if associated text can not be found for any reason.

Comments:

• This first scans the database(s) linked to the locale codes supplied.
• The database name for each locale takes the format of "<DBBase>_<Locale>.edb" so if the default DBBase is used, and the locales supplied are {"enus", "enau"} the databases scanned are "teksto_enus.edb" and "teksto_enau.edb". The database table name searched is "1" with the key being the message number, and the text is

the record data.
• If the message is not found in these databases (or the databases don't exist) a database called "<DBBase>.edb" is searched. Again the table name is "1" but it first looks for keys with the format {<locale>,msgnum} and failing that it looks for keys in the format {"", msgnum}, and if that fails it looks for a key of just the msgnum.

lang_load

loads a language file.

Signature:

```
lang_load(sequence filename)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `filename` : a sequence, the name of the file to load. If no file extension is supplied, then ".lng" is used.

Returns: A language **map**, if successful. This is to be used when calling translate().

If the load fails it returns a zero.

Comments: The language file must be made of lines which are either comments, empty lines or translations. Note that leading whitespace is ignored on all lines except continuation lines.

• **Comments** are lines that begin with a # character and extend to the end of the line.
• **Empty Lines** are ignored.
• **Translations** have two forms ...

keyword translation_text In which the 'keyword' is a word that must not have any spaces in it. keyphrase = translation_text In which the 'keyphrase' is anything up to the first '=' symbol.

It is possible to have the translation text span multiple lines. You do this by having '&' as the last character of the line. These are placed by newline characters when loading.

See Also: translate

Example 1:

```
# Example translation file
#


hello Hola
world Mundo
greeting %s, %s!

help text = &
This is an example of some &
translation text that spans &
multiple lines.

# End of example PO #2
```

money

converts an amount of currency into a string representing that amount.

Signature:

```
money(object amount)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `amount` : an atom, the value to write out.

Returns: A **sequence**, a string that writes out `amount` of current currency.

See Also: set, number

Example 1:

```
-- Assuming an en_US locale
money(1020.5) -- returns"$1,020.50"
```

number

converts a number into a string representing that number.

Signature:

```
number(object num)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `num` : an atom, the value to write out.

Returns: A **sequence**, a string that writes out `num`.

See Also: set, money

Example 1:

```
-- Assuming an en_US locale
number(1020.5) -- returns "1,020.50"
```

set

sets the computer locale, and possibly load appropriate translation file.

Signature:

```
set(sequence new_locale)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `new_locale` : a sequence representing a new locale.

Returns: An **integer**, either 0 on failure or 1 on success.

Comments: Locale strings have the following format: xx_YY or xx_YY.xyz . The xx part refers to a culture, or main language/script. For instance, "en" refers to English, "de" refers to German, and so on. For some language, a script may be specified, like in "mn_Cyrl_MN" (mongolian in cyrillic transcription).

The YY part refers to a subculture, or variant, of the main language. For instance, "fr_FR" refers to metropolitan France, while "fr_BE" refers to the variant spoken in Wallonie, the French speaking region of Belgium.

The optional .xyz part specifies an encoding (like .utf8 or .1252) which is required in some cases.

set_def_lang

sets the default language (translation) map.

Signature:

```
set_def_lang(object langmap)
```

```
public procedure
include locale.e
namespace locale
```

Arguments: ≡ `langmap` : A value returned by lang_load(), or zero to remove any default map.

Example 1:

```
set_def_lang( lang_load("appmsgs") )
```

set_lang_path

sets the language path.

Signature:

```
set_lang_path(object pp)
```

```
public procedure
include locale.e
namespace locale
```

Arguments: ≡ `pp` : an object, either an actual path or an atom.

Comments: When the language path is not set, and it is unset by default, set() does not load any language file.

See Also: set

translate

translates a word, using the current language file.

Signature:

```
translate(sequence word, object langmap = 0, object defval = "",
integer mode = 0)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `word` : a sequence, the word to translate.
≡ `langmap` : Either a value returned by lang_load() or zero to use the default language map
≡ `defval` : a object. The value to return if the word cannot be translated. Default is "". If `defval` is `PINF` then the `word` is returned if it can't be translated.
≡ `mode` : an integer. If zero (the default) it uses `word` as the keyword and returns the translation text. If not zero it uses `word` as the translation and returns the keyword.

Returns: A **sequence**, the value associated with `word`, or `defval` if there is no association.

See Also: set, lang_load

Example 1:

```
sequence newword
newword = translate(msgtext)
if length(msgtext) = 0 then
```

```
       error_message(msgtext)
    else
       error_message(newword)
    end if
```

Example 2:

```
    error_message(translate(msgtext, , PINF))
```

trsprintf

returns a formatted string with automatic translation performed on the parameters.

Signature:

```
trsprintf(sequence fmt, sequence data, object langmap = 0)
```

```
public function
include locale.e
namespace locale
```

Arguments: ≡ `fmt` : A sequence. Contains the formatting string. see printf() for details.
≡ `data` : A sequence. Contains the data that goes into the formatted result. see printf for details.
≡ `langmap` : An object. Either 0 (the default) to use the default language maps, or the result returned from lang_load() to specify a particular language map.

Returns: A **sequence**, the formatted result.

Comments: This works very much like the `sprintf` function. The difference is that the `fmt` sequence and sequences contained in the `data` parameter are translated before passing them to `sprintf`. If an item has no translation, it remains unchanged.

Further more, after the translation pass, if the result text begins with "__", the "__" is removed. This method can be used when you do not want an item to be translated.

See Also: sprinf , translate

Example 1:

```
    -- Assuming a language has been loaded and
    --   "greeting" translates as '%s %s, %s'
    --   "hello"  translates as "G'day"
    --   "how are you today" translates as "How's the family?"
    sequence UserName = "Bob"
    sequence result = trsprintf(
                         "greeting",
                        {"hello", "__" & UserName, "how are you today"})
       --> "G'day Bob, How's the family?"
```

localeconv

Constants

w32_names
w32_name_canonical
posix_names
locale_canonical
platform_locale

Locale Name Translation

canonical
decanonical
canon2win

***Constants** Windows* locale names:

af-ZA	sq-AL	gsw-FR	am-ET	ar-DZ	ar-BH	ar-EG	ar-IQ
ar-JO	ar-KW	ar-LB	ar-LY	ar-MA	ar-OM	ar-QA	ar-SA
ar-SY	ar-TN	ar-AE	ar-YE	hy-AM	as-IN	az-Cyrl-AZ	az-Latn-AZ
ba-RU	eu-ES	be-BY	bn-IN	bs-Cyrl-BA	bs-Latn-BA	br-FR	bg-BG
ca-ES	zh-HK	zh-MO	zh-CN	zh-SG	zh-TW	co-FR	hr-BA
hr-HR	cs-CZ	da-DK	prs-AF	dv-MV	nl-BE	nl-NL	en-AU
en-BZ	en-CA	en-029	en-IN	en-IE	en-JM	en-MY	en-NZ
en-PH	en-SG	en-ZA	en-TT	en-GB	en-US	en-ZW	et-EE
fo-FO	fil-PH	fi-FI	fr-BE	fr-CA	fr-FR	fr-LU	fr-MC
fr-CH	fy-NL	gl-ES	ka-GE	de-AT	de-DE	de-LI	de-LU
de-CH	el-GR	kl-GL	gu-IN	ha-Latn-NG	he-IL	hi-IN	hu-HU
is-IS	ig-NG	id-ID	iu-Latn-CA	iu-Cans-CA	ga-IE	it-IT	it-CH
ja-JP	kn-IN	kk-KZ	kh-KH	qut-GT	rw-RW	kok-IN	ko-KR
ky-KG	lo-LA	lv-LV	lt-LT	dsb-DE	lb-LU	mk-MK	ms-BN
ms-MY	ml-IN	mt-MT	mi-NZ	arn-CL	mr-IN	moh-CA	mn-Cyrl-MN
mn-Mong-CN	ne-IN	ne-NP	nb-NO	nn-NO	oc-FR	or-IN	ps-AF
fa-IR	pl-PL	pt-BR	pt-PT	pa-IN	quz-BO	quz-EC	quz-PE
ro-RO	rm-CH	ru-RU	smn-FI	smj-NO	smj-SE	se-FI	se-NO
se-SE	sms-FI	sma-NO	sma-SE	sa-IN	sr-Cyrl-BA	sr-Latn-BA	sr-Cyrl-CS
sr-Latn-CS	ns-ZA	tn-ZA	si-LK	sk-SK	sl-SI	es-AR	es-BO
es-CL	es-CO	es-CR	es-DO	es-EC	es-SV	es-GT	es-HN
es-MX	es-NI	es-PA	es-PY	es-PE	es-PR	es-ES	es-ES_tradnl
es-US	es-UY	es-VE	sw-KE	sv-FI	sv-SE	syr-SY	tg-Cyrl-TJ
tmz-Latn-DZ	ta-IN	tt-RU	te-IN	th-TH	bo-BT	bo-CN	tr-TR
tk-TM	ug-CN	uk-UA	wen-DE	tr-IN	ur-PK	uz-Cyrl-UZ	uz-Latn-UZ
vi-VN	cy-GB	wo-SN	xh-ZA	sah-RU	ii-CN	yo-NG	zu-ZA

localeconv API

canon2win

gets the translation of a canoncial locale string for the *windows* platform.

Signature:

```
canon2win(sequence new_locale)
```

```
public function
include localeconv.e
namespace localconv
```

Arguments: ≡ `new_locale`: a sequence, the string for the locale.

Returns: A **sequence**, either the *windows* native locale name on success or "C" on failure.

Platform: *windows*

See Also: get, set, canonical, decanonical

canonical

Get canonical name for a locale.

Signature:

```
canonical(sequence new_locale)
```

```
public function
include localeconv.e
namespace localconv
```

Arguments: ≡ `new_locale` : a sequence, the string for the locale.

Returns: A **sequence**, either the translated locale on success or `new_locale` on failure.

See Also: get, set, decanonical

decanonical

gets the translation of a locale string for current platform.

Signature:

```
decanonical(sequence new_locale)
```

```
public function
include localeconv.e
namespace localconv
```

Arguments: ≡ `new_locale`: a sequence, the string for the locale.

Returns: A **sequence**, either the translated locale on success or `new_locale` on failure.

See Also: get, set, canonical

locale_canonical

Signature:

```
locale_canonical
```

```
public constant
include localeconv.e
namespace localconv
```

platform_locale

Signature:

```
platform_locale
```

```
public constant
include localeconv.e
namespace localconv
```

posix_names

Signature:

```
posix_names
```

```
public constant
include localeconv.e
namespace localconv
```

w32_name_canonical

Canonical locale names for *windows*:

Signature:

```
w32_name_canonical
```

```
public constant
include localeconv.e
namespace localconv
```

w32_names

Signature:

```
w32_names
```

```
public constant
include localeconv.e
namespace localconv
```

---

map

Operation codes for put

- PUT
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE
- APPEND
- CONCAT
- LEAVE

Types of Maps

- SMALLMAP
- LARGEMAP

Types

- map

Routines

- calc_hash
- threshold
- type_of
- rehash
- new
- new_extra
- compare
- has
- get
- nested_get
- put
- nested_put

---

map API

ADD

Signature:

```
ADD
```

```
public enum
include map.e
namespace map
```

APPEND

Signature:

```
APPEND
```

```
public enum
include map.e
namespace map
```

AVERAGE_BUCKET

Signature:

```
AVERAGE_BUCKET
```

```
public enum
include map.e
namespace map
```

CONCAT

Signature:

```
CONCAT
```

```
public enum
include map.e
namespace map
```

DIVIDE

Signature:

```
DIVIDE
```

```
public enum
include map.e
namespace map
```

LARGEMAP

Signature:

```
LARGEMAP
```

```
public constant
include map.e
namespace map
```

LARGEST_BUCKET

Signature:

```
LARGEST_BUCKET
```

```
public enum
include map.e
namespace map
```

LEAVE

Signature:

```
LEAVE
```

```
public enum
include map.e
namespace map
```

MULTIPLY

Signature:

```
MULTIPLY
```

```
public enum
include map.e
namespace map
```

NUM_BUCKETS

Signature:

```
NUM_BUCKETS
```

```
public enum
include map.e
namespace map
```

NUM_ENTRIES

Signature:

```
NUM_ENTRIES
```

```
public enum
include map.e
namespace map
```

NUM_IN_USE

Signature:

```
NUM_IN_USE
```

```
public enum
include map.e
namespace map
```

PUT

Signature:

```
PUT
```

```
public enum
include map.e
namespace map
```

SMALLEST_BUCKET

Signature:

```
SMALLEST_BUCKET
```

```
public enum
include map.e
namespace map
```

SMALLMAP

Signature:

```
SMALLMAP
```

```
public constant
include map.e
namespace map
```

SM_RAW

Signature:

```
SM_RAW
```

```
public enum
include map.e
namespace map
```

SM_TEXT

Signature:

```
SM_TEXT
```

```
public enum
include map.e
namespace map
```

STDEV_BUCKET

Signature:

```
STDEV_BUCKET
```

```
public enum
include map.e
namespace map
```

SUBTRACT

Signature:

```
SUBTRACT
```

```
public enum
include map.e
namespace map
```

calc_hash

calculate a hashing value from the supplied data.

Signature:

```
calc_hash(object key_p, integer max_hash_p)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `key_p` : The data for which you want a hash value calculated.
≡ `max_hash_p` : The returned value will be no larger than this value.

Returns: An **integer**, the value of which depends only on the supplied data.

Comments: This is used whenever you need a single number to represent the data you supply. It can calculate the number based on all the data you give it, which can be an atom or sequence of any value.

Example 1:

```
integer h1
-- calculate a hash value and ensure it will be a value from 1 to 4097.
h1 = calc_hash( symbol_name, 4097 )
```

clear

removes all entries in a map.

Signature:

```
clear(map the_map_p)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map to operate on

Comments:

- This is much faster than removing each entry individually.
- If you need to remove just one entry, see remove

See Also: remove, has

Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, "Amy", 66.9)
put(the_map_p, "Betty", 67.8)
put(the_map_p, "Claire", 64.1)
..
clear(the_map_p)
-- the_map_p is now an empty map again
```

compare

tests if two maps are equal.

Signature:

```
compare(map map_1_p, map map_2_p, integer scope_p = 'd')
```

```
public function
```

```
include map.e
namespace map
```

Arguments: ≡ `map_1_p` : A map
≡ `map_2_p` : A map
≡ `scope_p` : An integer that specifies what to compare.
…… ♦ 'k' or 'K' to only compare keys.
…… ♦ 'v' or 'V' to only compare values.
…… ♦ 'd' or 'D' to compare both keys and values. This is the default.

Returns: An **integer**,
• -1 if they are not equal.
• 0 if they are literally the same map.
• 1 if they contain the same keys and/or values.

Example 1:

```
map map_1_p = foo()
map map_2_p = bar()
if compare(map_1_p, map_2_p, 'k') >= 0 then
     ..
```

copy

duplicates a map.

Signature:

```
copy(map source_map, object dest_map = 0, integer put_operation = PUT)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `source_map` : map to copy from
≡ `dest_map` : optional, map to copy to
≡ `put_operation` : optional, operation to use when `dest_map` is used. The default is PUT.

Returns: If `dest_map` was not provided, an exact duplicate of `source_map` otherwise `dest_map`, which does not have to be empty, is returned with the new values copied from `source_map`, according to the `put_operation` value.

See Also: put

Example 1:

```
map m1 = new()
put(m1, 1, "one")
put(m1, 2, "two")

map m2 = copy(m1)
printf(1, "%s, %s\n", { get(m2, 1), get(m2, 2) })
-- one, two

put(m1, 1, "one hundred")
printf(1, "%s, %s\n", { get(m1, 1), get(m1, 2) })
-- one hundred, two

printf(1, "%s, %s\n", { get(m2, 1), get(m2, 2) })
-- one, two
```

Example 2:

```
map m1 = new()
map m2 = new()

put(m1, 1, "one")
put(m1, 2, "two")
put(m2, 3, "three")
```

```
copy(m1, m2)

? keys(m2)
-- { 1, 2, 3 }
```

Example 3:

```
map m1 = new()
map m2 = new()

put(m1, "XY", 1)
put(m1, "AB", 2)
put(m2, "XY", 3)

pairs(m1) --> { {"AB", 2}, {"XY", 1} }
pairs(m2) --> { {"XY", 3} }

-- Add same keys' values.
copy(m1, m2, ADD)

pairs(m2) --> { {"AB", 2}, {"XY", 4} }
```

for_each

calls a user-defined routine for each of the items in a map.

Signature:

```
for_each(map source_map, integer user_rid, object user_data = 0,
integer in_sorted_order = 0, integer signal_boundary = 0)


public function
include map.e
namespace map
```

Arguments: ≡ `source_map` : The map containing the data to process
≡ `user_rid`: The routine_id of a user defined processing function
≡ `user_data`: An object. Optional. This is passed, unchanged to each call of the user defined routine. By default, zero (0) is used.
≡ `in_sorted_order`: An integer. Optional. If non-zero the items in the map are processed in ascending key sequence otherwise the order is undefined. By default they are not sorted.
≡ `signal_boundary`: A integer; 0 (the default) means that the user routine is not called if the map is empty and when the last item is passed to the user routine, the Progress Code is not negative.

Returns: An integer: 0 means that all the items were processed, and anything else is whatever was returned by the user routine to abort the `for_each()` process.

Comments:

• The user defined routine is a function that must accept four parameters. `Object: an Item Key` Object: an Item Value `Object: The` user_data `value. This is never used by` for_each() `itself, merely passed to the user routine.` Integer: Progress code. * **The `abs()` value of the progress code is the ordinal call number. That is 1 means the first call, 2 means the second call, etc ... *** If the progress code is negative, it is also the last call to the routine. * **If the progress code is zero, it means that the map is empty and thus the item key and value cannot be used. * note** that if `signal_boundary` is zero, the Progress Code is never less than 1.
• The user routine must return 0 to get the next map item. Anything else will cause `for_each()` to stop running, and is returned to whatever called `for_each()`.
• Note that any changes that the user routine makes to the map do not affect the order or number of times the routine is called. `for_each()` takes a copy of the map keys and data before the first call to the user routine and uses the copied data to call the user routine.

Example 1:

```
include std/map.e
include std/math.e
include std/io.e

function Process_A(object k, object v, object d, integer pc)
    writefln("[] = []", {k, v})
    return 0
end function

function Process_B(object k, object v, object d, integer pc)
    if pc = 0 then
      writefln("The map is empty")
    else
      integer c
      c = abs(pc)
      if c = 1 then
          writefln("---[]---", {d}) -- Write the report title.
      end if
      writefln("[]: [:15] = []", {c, k, v})
      if pc < 0 then
          writefln(repeat('-', length(d) + 6), {}) -- Write the report end.
      end if
    end if
    return 0
end function

map m1 = new()
map:put(m1, "application", "Euphoria")
map:put(m1, "version", "4.0")
map:put(m1, "genre", "programming language")
map:put(m1, "crc", "4F71AE10")

-- Unsorted
map:for_each(m1, routine_id("Process_A"))
-- Sorted
map:for_each(m1, routine_id("Process_B"), "List of Items", 1)
```

The output from the first call could be...

```
application = Euphoria
version = 4.0
genre = programming language
crc = 4F71AE10
```

The output from the second call should be...

```
---List of Items---
1: application     = Euphoria
2: crc             = 4F71AE10
3: genre           = programming language
4: version         = 4.0
-------------------
```

get

retrieves the value associated to a key in a map.

Signature:

```
get(integer the_map_p, object the_key_p, object default_value_p = 0)
```

```
public function
include map.e
namespace map
```

Arguments:
≡ `the_map_p` : the map to inspect
≡ `the_key_p` : an object, the the_key_p being looked tp
≡ `default_value_p` : an object, a default value returned if `the_key_p` not found. The default is 0.

Returns: An **object**, the value that corresponds to `the_key_p` in `the_map_p`. If `the_key_p` is not in `the_map_p`, `default_value_p` is returned instead.

See Also: has

Example 1:

```
map ages
ages = new()
put(ages, "Andy", 12)
put(ages, "Budi", 13)

integer age
age = get(ages, "Budi", -1)
if age = -1 then
    puts(1, "Age unknown")
else
    printf(1, "The age is %d", age)
end if
```

has

tests if a key exists in a map.

Signature:

```
has(integer the_map_p, object the_key_p)
```

```
public function
include map.e
namespace map
```

Arguments:
≡ `the_map_p` : the map to inspect
≡ `the_key_p` : an object to be looked up

Returns: An **integer**, 0 if not present, 1 if present.

See Also: get

Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, "name", "John")
? has(the_map_p, "name") -- 1
? has(the_map_p, "age")  -- 0
```

keys

returns all keys in a map.

Signature:

```
keys(map the_map_p, integer sorted_result = 0)
```

```
public function
include map.e
namespace map
```

Arguments:
≡ `the_map_p`: the map being queried
≡ `sorted_result`: optional integer. 0 [default] means do not sort the output and 1 means to sort the output before returning.

Returns: A **sequence** made of all the keys in the map.

Comments: If `sorted_result` is not used, the order of the keys returned is not predicable.

See Also: has, values, pairs

Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence keys
keys = keys(the_map_p) -- keys might be {20,40,10,30} or some other order
keys = keys(the_map_p, 1) -- keys will be {10,20,30,40}
```

load_map

loads a map from a file.

Signature:

```
load_map(object input_file_name)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `file_name_p` : The file to load from. This file may have been created by the save_map function. This can either be a name of a file or an already opened file handle.

Returns: Either a **map**, with all the entries found in `file_name_p`, or **-1** if the file failed to open, or **-2** if the file is incorrectly formatted.

Comments: If `file_name_p` is an already opened file handle, this routine will write to that file and not close it. Otherwise, the named file will be created and closed by this routine.

The input file can be either one created by the save_map function or a manually created/edited text file. See save_map for details about the required layout of the text file.

See Also: new, save_map

Example 1:

```
include std/error.e

object loaded
map AppOptions
sequence SavedMap = "c:\\myapp\\options.txt"

loaded = load_map(SavedMap)
if equal(loaded, -1) then
    crash("Map '%s' failed to open", SavedMap)
end if

-- By now we know that it was loaded and a new map created,
-- so we can assign it to a 'map' variable.
AppOptions = loaded
if get(AppOptions, "verbose", 1) = 3 then
    ShowIntructions()
end if
```

map

defines the datatype 'map'.

Signature:

```
map(object obj_p)
```

```
public type
include map.e
namespace map
```

Comments: Used when declaring a map variable.

Example 1:

```
map SymbolTable = new() -- Create a new map to hold the symbol table.
```

nested_get

returns the value that corresponds to the object `the_keys_p` in the nested map

Signature:

```
nested_get(map the_map_p, sequence the_keys_p, object default_value_p = 0)
```

```
public function
include map.e
namespace map
```

Comments: `the_keys_p` is a sequence of keys. If any key is not in the map, the object default_value_p is returned instead.

nested_put

adds or updates an entry on a map.

Signature:

```
nested_put(map the_map_p, sequence the_keys_p, object the_value_p,
integer operation_p = PUT, integer trigger_p = threshold_size)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map where an entry is being added or updated
≡ `the_keys_p` : a sequence of keys for the nested maps
≡ `the_value_p` : an object, the value to add, or to use for updating.
≡ `operation_p` : an integer, indicating what is to be done with `value`. Defaults to PUT.
≡ `trigger_p` : an integer. Default is the current threshold size. See Comments for details.

Comments:

• If existing entry with the same key is already in the map, the value of the entry is updated.
• The *trigger* parameter is used when you need to keep the average number of keys in a hash bucket to a specific maximum. The *trigger* value is the maximum allowed. Each time a *put* operation increases the hash table's average bucket size to be more than the *trigger* value the table is expanded by a factor 3.5 and the keys are rehashed into the enlarged table. This can be a time intensive action so set the value to one that is appropriate to your application.
…… ♦ By keeping the average bucket size to a certain maximum, it can speed up lookup times.
…… ♦ If you set the *trigger* to zero, it will not check to see if the table needs reorganizing. You might do this if you created the original bucket size to an optimal value. See new on how to do this.

Example 1:

```
map city_population
city_population = new()
nested_put(city_population, {"United States", "California", "Los Angeles"},
    3819951 )
nested_put(city_population, {"Canada",        "Ontario",    "Toronto"},
    2503281 )
```

new

creates a new map data structure.

Signature:

```
new(integer initial_size_p = 690)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `initial_size_p` : An estimate of how many initial elements will be stored in the map. If this value is less than the threshold value, the map will initially be a *small* map otherwise it will be a *large* map.

Returns: An empty **map**.

Comments: A new object of type map is created. The resources allocated for the map will be automatically cleaned up if the reference count of the returned value drops to zero, or if passed in a call to delete.

Example 1:

```
map m = new()  -- m is now an empty map
map x = new(threshold()) -- Forces a small map to be initialized
x = new()    -- the resources for the map previously stored in x
             -- are released automatically
delete( m )  -- the resources for the map are released
```

new_extra

returns either the supplied map or a new map.

Signature:

```
new_extra(object the_map_p, integer initial_size_p = 690)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : An object, that could be an existing map
≡ `initial_size_p` : An estimate of how many initial elements will be stored in a new map.

Returns: A **map**, If `m` is an existing map then it is returned otherwise this returns a new empty **map**.

Comments: This is used to return a new map if the supplied variable isn't already a map.

Example 1:

```
map m = new_extra( foo() ) -- If foo() returns a map it is used, otherwise
                           --  a new map is created.
```

new_from_kvpairs

converts a set of key/value pairs to a map.

Signature:

```
new_from_kvpairs(sequence kv_pairs)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `kv_pairs` : A seqeuence containing any number of subsequences that have the format {KEY, VALUE}. These are loaded into a new map which is then returned by this function.

Returns: A **map**, containing the data from `kv_pairs`

Example 1:

```
map m1 = new_from_kvpairs( {
    { "application", "Euphoria" },
    { "version", "4.0" },
    { "genre", "programming language" },
    { "crc", 0x4F71AE10 }
})

v = map:get(m1, "application") --> "Euphoria"
```

new_from_string

converts a set of key/value pairs contained in a string to a map.

Signature:

```
new_from_string(sequence kv_string)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `kv_string` : A string containing any number of lines that have the format KEY=VALUE. These are loaded into a new map which is then returned by this function.

Returns: A **map**, containing the data from `kv_string`

Example 1:

```
 application = Euphoria,
 version     = 4.0,
 genre       = "programming language",
 crc         = 4F71AE10

map m1 = new_from_string( read_file("xyz.config", TEXT_MODE))

printf(1, "%s\n", {map:get(m1, "application")}) --> "Euphoria"
printf(1, "%s\n", {map:get(m1, "genre")})       --> "programming language"
printf(1, "%s\n", {map:get(m1, "version")})     --> "4.0"
printf(1, "%s\n", {map:get(m1, "crc")})         --> "4F71AE10"
```

optimize

widens a map to increase performance.

Signature:

```
optimize(map the_map_p, integer max_p = threshold_size,
atom grow_p = 1.333)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map being optimized
≡ `max_p` : an integer, the maximum desired size of a bucket. Default is the current threshold size. This must be 3 or higher.
≡ `grow_p` : an atom, the factor to grow the number of buckets for each iteration of rehashing. Default is 1.333. This must be greater than 1.

Comments: This rehashes the map until either the maximum bucket size is less than the desired maximum or the maximum bucket size is less than the largest size statistically expected (mean + 3 standard deviations).

See Also: statistics, rehash

pairs

Signature:

```
pairs(map the_map_p, integer sorted_result = 0)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map to get the data from
≡ `sorted_result` : optional integer. 0 [default] means do not sort the output and 1 means to sort the output before returning.

Returns: A **sequence**, of all key/value pairs stored in `the_map_p`. Each pair is a sub-sequence in the form {key, value}

Comments: If `sorted_result` is not used, the order of the values returned is not predicable.

See Also: get, keys, values

Example 1:

```
map the_map_p

the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence keyvals
keyvals = pairs(the_map_p)
-- might be {{20,"twenty"},{40,"forty"},{10,"ten"},{30,"thirty"}}

keyvals = pairs(the_map_p, 1)
-- will be {{10,"ten"},{20,"twenty"},{30,"thirty"},{40,"forty"}}
```

put

adds or updates an entry in a map.

Signature:

```
put(integer the_map_p, object the_key_p, object the_value_p,
integer operation_p = map :PUT, integer trigger_p = threshold_size)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map where an entry is being added or updated
≡ `the_key_p` : an object, the the_key_p to look up
≡ `the_value_p` : an object, the value to add, or to use for updating.
≡ `operation` : an integer, indicating what is to be done with `the_value_p`. Defaults to PUT.
≡ `trigger_p` : an integer. Default is the current threshold size. See Comments for details.

Comments:

• The operation parameter can be used to modify the existing value. Valid operations are:

...... ♦ PUT -- This is the default, and it replaces any value in there already
...... ♦ ADD -- Equivalent to using the += operator
...... ♦ SUBTRACT -- Equivalent to using the -= operator
...... ♦ MULTIPLY -- Equivalent to using the *= operator
...... ♦ DIVIDE -- Equivalent to using the /= operator
...... ♦ APPEND -- Appends the value to the existing data
...... ♦ CONCAT -- Equivalent to using the &= operator
...... ♦ LEAVE -- If it already exists, the current value is left unchanged otherwise the new value is added to the map.

• The *trigger* parameter is used when you need to keep the average number of keys in a hash bucket to a specific maximum. The *trigger* value is the maximum allowed. Each time a *put* operation increases the hash table's average bucket size to be more than the *trigger* value the table is expanded by a factor of 3.5 and the keys are rehashed into the enlarged table. This can be a time intensive action so set the value to one that is appropriate to your application.
...... ♦ By keeping the average bucket size to a certain maximum, it can speed up lookup times.
...... ♦ If you set the *trigger* to zero, it will not check to see if the table needs reorganizing. You might do this if you created the original bucket size to an optimal value. See new on how to do this.

See Also: remove, has, nested_put

Example 1:

```
map ages
ages = new()
put(ages, "Andy", 12)
put(ages, "Budi", 13)
put(ages, "Budi", 14)

-- ages now contains 2 entries: "Andy" => 12, "Budi" => 14
```

rehash

changes the width (the number of buckets) of a map.

Signature:

```
rehash(integer the_map_p, integer requested_bucket_size_p = 0)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `m` : the map to resize
≡ `requested_bucket_size_p` : a lower limit for the new size.

Comments: Only affects *large* maps.

If the `requested_bucket_size_p` is not greater than zero, a new width is automatically derived from the current one.

See Also: statistics, optimize

remove

remove an entry with given key from a map.

Signature:

```
remove(map the_map_p, object the_key_p)
```

```
public procedure
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map to operate on
≡ `key` : an object, the key to remove.

Comments:

- If `key` is not on `the_map_p`, the `the_map_p` is returned unchanged.
- If you need to remove all entries, see clear

See Also: clear, has

Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, "Amy", 66.9)
remove(the_map_p, "Amy")
-- the_map_p is now an empty map again
```

save_map

saves a map to a file.

Signature:

```
save_map(map the_map_, object file_name_p, integer type_ = SM_TEXT)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `m` : a map.
≡ `file_name_p` : Either a sequence, the name of the file to save to, or an open file handle as returned by open().
≡ `type` : an integer. SM_TEXT for a human-readable format (default), SM_RAW for a smaller and faster format, but not human-readable.

Returns: An **integer**, the number of keys saved to the file, or -1 if the save failed.

Comments: If `file_name_p` is an already opened file handle, this routine will write to that file and not close it. Otherwise, the named file will be created and closed by this routine.

The SM_TEXT type saves the map keys and values in a text format which can be read and edited by standard text editor. Each entry in the map is saved as a KEY/VALUE pair in the form
key = value Note that if the 'key' value is a normal string value, it can be enclosed in double quotes. If it is not thus quoted, the first character of the key determines its Euphoria value type. A dash or digit implies an atom, an left-brace implies a sequence, an alphabetic character implies a text string that extends to the next equal '=' symbol, and anything else is ignored.

Note that if a line contains a double-dash, then all text from the double-dash to the end of the line will be ignored. This is so you can optionally add comments to the saved map. Also, any blank lines are ignored too.

All text after the '=' symbol is assumed to be the map item's value data.

Because some map data can be rather long, it is possible to split the text into multiple lines, which will be considered by load_map as a single *logical* line. If an line ends with a comma (,) or a dollar sign ($), then the next actual line is appended to the end of it. After all these physical lines have been joined into one logical line, all combinations of `",$"` and `,$` are removed.

See Also: load_map

Example 1:

```
include std/error.e

map AppOptions
if save_map(AppOptions,
    crash("Failed to save application options")
end if

if save_map(AppOptions,
    crash("Failed to save application options")
end if
```

size

returns the number of entries in a map.

Signature:

```
size(map the_map_p)
```

```
public function
include map.e
namespace map
```

Arguments: `the_map_p` : the map being queried

Returns: An **integer**, the number of entries it has.

Comments: For an empty map, size will be zero

See Also: statistics

Example 1:

```
map the_map_p
put(the_map_p, 1, "a")
put(the_map_p, 2, "b")
? size(the_map_p) -- outputs 2
```

statistics

retrieves the characteristics of a map.

Signature:

```
statistics(map the_map_p)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `the_map_p` : the map being queried

Returns: A **sequence**, of 7 integers:
- `NUM_ENTRIES` -- number of entries
- `NUM_IN_USE` -- number of buckets in use
- `NUM_BUCKETS` -- number of buckets
- `LARGEST_BUCKET` -- size of largest bucket
- `SMALLEST_BUCKET` -- size of smallest bucket
- `AVERAGE_BUCKET` -- average size for a bucket
- `STDEV_BUCKET` -- standard deviation for the bucket length series

Example 1:

```
sequence s = statistics(mymap)
printf(1, "The average size of the buckets is %d", s[AVERAGE_BUCKET])
```

threshold

gets or sets the threshold value that determines at what point a small map

Signature:

```
threshold(integer new_value_p = 0)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `new_value_p` : If this is greater than zero then it **sets** the threshold value.

Returns: An **integer**, the current value (when `new_value_p` is less than 1) or the old value prior to setting it to `new_value_p`.

Comments: The initial threshold is set to twenty-three, meaning that maps up to 23 elements use the *small map* structure.

type_of

determines the type of the map.

Signature:

```
type_of(map the_map_p)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `m` : A map

Returns: An **integer**, Either *SMALLMAP* or *LARGEMAP*

values

returns values, without their keys, from a map.

Signature:

```
values(map the_map, object keys = 0, object default_values = 0)
```

```
public function
include map.e
namespace map
```

Arguments: ≡ `the_map` : the map being queried
≡ `keys` : optional, key list of values to return.
≡ `default_values` : optional default values for keys list

Returns: A **sequence**, of all values stored in `the_map`.

Comments:

- The order of the values returned may not be the same as the putting order.
- Duplicate values are not removed.
- You use the `keys` parameter to return a specific set of values from the map. They are returned in the same order as the `keys` parameter. If no `default_values` is given and one is needed, 0 will be used.
- If `default_values` is an atom, it represents the default value for all values in `keys`.
- If `default_values` is a sequence, and its length is less than `keys`, then the last item in `default_values` is used for the rest of the `keys`.

See Also: get, keys, pairs

Example 1:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence values
values = values(the_map_p)
-- values might be {"twenty","forty","ten","thirty"}
-- or some other order
```

Example 2:

```
map the_map_p
the_map_p = new()
put(the_map_p, 10, "ten")
put(the_map_p, 20, "twenty")
put(the_map_p, 30, "thirty")
put(the_map_p, 40, "forty")

sequence values
values = values(the_map_p, { 10, 50, 30, 9000 })
-- values WILL be { "ten", 0, "thirty", 0 }
values = values(the_map_p, { 10, 50, 30, 9000 }, {-1,-2,-3,-4})
-- values WILL be { "ten", -2, "thirty", -4 }
```

math

Sign and comparisons

- abs
- sign
- larger_of
- smaller_of
- max
- min
- ensure_in_range
- ensure_in_list

Roundings and remainders

- remainder
- mod
- trunc
- frac
- intdiv
- floor
- ceil
- round

Trigonometry

- arctan
- tan
- cos
- sin
- arccos
- arcsin
- atan2
- rad2deg

---

math API

---

abs

Returns the absolute value of numbers.

Signature:

```
abs(object a)


public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom is processed, no matter how deeply nested.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is the same if not less than zero, and the opposite value otherwise.

Comments: This function may be applied to an atom or to all elements of a sequence

See Also: sign

Example 1:

```
x = abs({10.5, -12, 3})
-- x is {10.5, 12, 3}

i = abs(-4)
-- i is 4
```

and_bits

Perform the bitwise AND operation on corresponding bits in two objects. A bit in the

Signature:

```
and_bits(object a, object b)
```

```
<built-in> function
```

Arguments:
- ≡ `a` : one of the objects involved
- ≡ `b` : the second object

Returns: An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwise AND between atoms on both objects.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply. The atoms in the arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as atom, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

To understand the binary representation of a number you should display it in hexadecimal notation. Use the %x format of printf(). Using int_to_bits() is an even more direct approach.

See Also: or_bits, xor_bits, not_bits, int_to_bits

Example 1:

```
a = and_bits(#0F0F0000, #12345678)
-- a is #02040000
```

Example 2:

```
a = and_bits(#FF, {#123456, #876543, #2211})
-- a is {#56, #43, #11}
```

Example 3:

```
a = and_bits(#FFFFFFFF, #FFFFFFFF)
-- a is -1
-- Note that #FFFFFFFF is a positive number,
-- but the result of a bitwise operation is interpreted
-- as a signed 32-bit number, so it's negative.
```

approx

Compares two (sets of) numbers based on approximate equality.

Signature:

```
approx(object p, object q, atom epsilon = 0.005)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `p` : an object, one of the sets to consider
≡ `q` : an object, the other set.
≡ `epsilon` : an atom used to define the amount of inequality allowed. This must be a positive value. Default is 0.005

Returns: An **integer**,
• 1 when p > (q + epsilon) : P is definitely greater than q.
• -1 when p < (q - epsilon) : P is definitely less than q.
• 0 when p >= (q - epsilon) and p <= (q + epsilon) : p and q are approximately equal.

Comments: This can be used to see if two numbers are near enough to each other.

Also, because of the way floating point numbers are stored, it not always possible express every real number exactly, especially after a series of arithmetic operations. You can use `approx()` to see if two floating point numbers are almost the same value.

If `p` and `q` are both sequences, they must be the same length as each other.

If `p` or `q` is a sequence, but the other is not, then the result is a sequence of results whose length is the same as the sequence argument.

Example 1:

```
? approx(10, 33.33 * 30.01 / 100)
          --> 0 because 10 and 10.002333 are within 0.005 of each other
? approx(10, 10.001)
          --> 0 because 10 and 10.001 are within 0.005 of each other
? approx(10, {10.001,9.999, 9.98, 10.04})
          --> {0,0,1,-1}
? approx({10.001,9.999, 9.98, 10.04}, 10)
          --> {0,0,-1,1}
? approx({10.001,{9.999, 10.01}, 9.98, 10.04}, {10.01,9.99, 9.8, 10.4})
          --> {-1,{1,1},1,-1}
? approx(23,32, 10)
          --> 0 because 23 and 32 are within 10 of each other.
```

arccos

Return an angle given its cosine.

Signature:

```
arccos(trig_range x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom in which will be acted upon.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is an atom, an angle whose cosine is `value`.

Comments: A value between 0 and PI radians will be returned.

This function may be applied to an atom or to all elements of a sequence.

`arccos`() is not as fast as arctan().

See Also: cos, PI, arctan

Example 1:

```
s = arccos({-1,0,1})
-- s is {3.141592654, 1.570796327, 0}
```

arccosh

Computes the reverse hyperbolic cosine of an object.

Signature:

```
arccosh(not_below_1 a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic cosine grows like the logarithm function.

See Also: arccos, arcsinh, cosh

Example 1:

```
 ? arccosh(1) -- prints out 0
```

arcsin

Return an angle given its sine.

Signature:

```
arcsin(trig_range x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom in which will be acted upon.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is an atom, an angle whose sine is `value`.

Comments: A value between -PI/2 and +PI/2 (radians) inclusive will be returned.

This function may be applied to an atom or to all elements of a sequence.

`arcsin`() is not as fast as arctan().

See Also: arccos, arccos, sin

Example 1:

```
 s = arcsin({-1,0,1})
 s is {-1.570796327, 0, 1.570796327}
```

arcsinh

Computes the reverse hyperbolic sine of an object.

Signature:

```
arcsinh(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic sine grows like the logarithm function.

See Also: arccosh, arcsin, sinh

Example 1:

```
? arcsinh(1) -- prints out 0,4812118250596034
```

arctan

Return an angle with given tangent.

Signature:

```
arctan(object tangent)
```

```
<built-in> function
```

Arguments: ≡ `tangent` : an object, each atom of which will be converted, no matter how deeply nested.

Returns: An **object**, of the same shape as `tangent`. For each atom in `flatten(tangent)`, the angle with smallest magnitude that has this atom as tangent is computed.

Comments: All atoms in the returned value lie between -PI/2 and PI/2, exclusive.

This function may be applied to an atom or to all elements of a sequence (of sequence (...)).

`arctan`() is faster than `arcsin()` or `arccos()`.

See Also: arcsin, arccos, tan, flatten

Example 1:

```
s = arctan({1,2,3})
-- s is {0.785398, 1.10715, 1.24905}
```

arctanh

Computes the reverse hyperbolic tangent of an object.

Signature:

```
arctanh(abs_below_1 a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic cosine grows like the logarithm function.

See Also: arccos, arcsinh, cosh

Example 1:

```
? arctanh(1/2) -- prints out 0,5493061443340548456976
```

atan2

Calculate the arctangent of a ratio.

Signature:

```
atan2(atom y, atom x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `y` : an atom, the numerator of the ratio
≡ `x` : an atom, the denominator of the ratio

Returns: An **atom**, which is equal to arctan(`y`/`x`), except that it can handle zero denominator and is more accurate.

See Also: arctan

Example 1:

```
a = atan2(10.5, 3.1)
-- a is 1.283713958
```

ceil

Computes the next integer equal or greater than the argument.

Signature:

```
ceil(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom of which processed, no matter how deeply nested.

Returns: An **object**, the same shape as `value`. Each atom in `value` is returned as an integer that is the smallest integer equal to or greater than the corresponding atom in `value`.

Comments: This function may be applied to an atom or to all elements of a sequence.

`ceil(X)` is 1 more than `floor(X)` for non-integers. For integers, `X = floor(X) = ceil(X)`.

See Also: floor, round

Example 1:

```
sequence nums
nums = {8, -5, 3.14, 4.89, -7.62, -4.3}
nums = ceil(nums) -- {8, -5, 4, 5, -7, -4}
```

cos

Return the cosine of an angle expressed in radians

Signature:

```
cos(object angle)
```

```
<built-in> function
```

Arguments: ≡ `angle` : an object, each atom of which will be converted, no matter how deeply nested.

Returns: An **object**, the same shape as `angle`. Each atom in `angle` is turned into its cosine.

Comments: This function may be applied to an atom or to all elements of a sequence.

The cosine of an angle is an atom between -1 and 1 inclusive. 0.0 is hit by odd multiples of PI/2 only.

See Also: sin, tan, arccos, PI, deg2rad

Example 1:

```
x = cos({0.5, 0.6, 0.7})
-- x is {0.8775826, 0.8253356, 0.7648422}
```

cosh

Computes the hyperbolic cosine of an object.

Signature:

```
cosh(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic cosine grows like the exponential function.

For all reals, `power(cosh(x), 2) - power(sinh(x), 2) = 1`. Compare with ordinary trigonometry.

See Also: cos, sinh, arccosh

Example 1:

```
? cosh(LN2) -- prints out 1.25
```

deg2rad

Convert an angle measured in degrees to an angle measured in radians

Signature:

```
deg2rad(object x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `angle` : an object, all atoms of which will be converted, no matter how deeply nested.

Returns: An **object**, the same shape as `angle`, all atoms of which were multiplied by PI/180.

Comments: This function may be applied to an atom or sequence. A flat angle is PI radians and 180 degrees. sin(), cos() and tan() expect angles in radians.

See Also: rad2deg

Example 1:

```
x = deg2rad(194)
-- x is 3.385938749
```

ensure_in_list

Ensures that the `item` is in a list of values supplied by `list`

Signature:

```
ensure_in_list(object item, sequence list, integer default = 1)
```

```
public function
include math.e
```

```
namespace math
```

Arguments: ≡ `item` : The object to test for.
≡ `list` : A sequence of elements that `item` should be a member of.
≡ `default` : an integer, the index of the list item to return if `item` is not found. Defaults to 1.

Returns: An **object**, if `item` is not in the list, it returns the list item of index `default`, otherwise it returns `item`.

Comments: If `default` is set to an invalid index, the first item on the list is returned instead when `item` is not on the list.

Example 1:

```
object valid_data = ensure_in_list(user_data, {100, 45, 2, 75, 121})
if not equal(valid_data, user_data) then
    errmsg("Invalid input supplied. Using %d instead.", valid_data)
end if
procA(valid_data)
```

ensure_in_range

Ensures that the `item` is in a range of values supplied by inclusive `range_limits`

Signature:

```
ensure_in_range(object item, sequence range_limits)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `item` : The object to test for.
≡ `range_limits` : A sequence of two or more elements. The first is assumed to be the smallest value and the last is assumed to be the highest value.

Returns: A **object**, If `item` is lower than the first item in the `range_limits` it returns the first item. If `item` is higher than the last element in the `range_limits` it returns the last item. Otherwise it returns `item`.

Example 1:

```
object valid_data = ensure_in_range(user_data, {2, 75})
if not equal(valid_data, user_data) then
    errmsg("Invalid input supplied. Using %d instead.", valid_data)
end if
procA(valid_data)
```

exp

Computes some power of E.

Signature:

```
exp(atom x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, all atoms of which will be acted upon, no matter how deeply nested.

Returns: An **object**, the same shape as `value`. When `value` is an atom, its exponential is being returned.

Comments: This function can be applied to a single atom or to a sequence of any shape.

Due to its rapid growth, the returned values start losing accuracy as soon as values are greater than 10. Values above 710 will cause an overflow in hardware.

See Also: log

Example 1:

```
 x = exp(5.4)
 -- x is 221.4064162
```

fib

Computes the Nth Fibonacci Number

Signature:

```
fib(integer i)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an integer. The starting value to compute a Fibonacci Number from.

Returns: An **atom**,
• The Fibonacci Number specified by value.

Comments:

• Note that due to the limitations of the floating point implementation, only 'i' values less than 76 are accurate on Windows platforms, and 69 on other platforms (due to rounding differences in the native C runtime libraries).

Example 1:

```
 ? fib(6)
 -- output ...
 -- 8
```

floor

Rounds `value` down to the next integer less than or equal to `value`. It

Signature:

```
floor(object value)
```

```
<built-in> function
```

Arguments: ≡ `value` : any Euphoria object; each atom in `value` will be acted upon.

Returns: An **object**, the same shape as `value` but with each item guarenteed to be an integer less than or equal to the corresponding item in `value`.

See Also: ceil, round

Example 1:

```
 y = floor({0.5, -1.6, 9.99, 100})
 -- y is {0, -2, 9, 100}
```

frac

Return the fractional portion of a number.

Signature:

```
frac(object x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : any Euphoria object.

Returns: An **object**, the shape of which depends on `values`'s. Each item in the returned object will be the same corresponding items in `value` except with the integer portion removed.

Comments: Note that `trunc(x) + frac(x) = x`

See Also: trunc

Example 1:

```
 a = frac(9.4)
 -- a is 0.4
```

Example 2:

```
 s = frac({81, -3.5, -9.999, 5.5})
 -- s is {0, -0.5, -0.999, 0.5}
```

gcd

Returns the greater common divisor of to atoms

Signature:

```
gcd(atom p, atom q)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `p` : one of the atoms to consider
≡ `q` : the other atom.

Returns: A positive **atom**, without a fractional part, evenly dividing both parameters, and is the greatest value with those properties.

Comments: Signs are ignored. Atoms are rounded down to integers.

Any zero parameter causes 0 to be returned.

Parameters and return value are atoms so as to take mathematical integers up to `power(2,53)`.

Example 1:

```
 ? gcd(76.3, -114) -- prints out gcd(76,114), which is 38
```

Floating Point

intdiv

Return an integral division of two objects.

Signature:

```
intdiv(object a, object b)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `divided` : any Euphoria object.
≡ `divisor` : any Euphoria object.

Returns: An **object**, which will be a sequence if either `dividend` or `divisor` is a sequence.

Comments:

• This calculates how many non-empty sets when `dividend` is divided by `divisor`.
• The result's sign is the same as the `dividend`'s sign.

Example 1:

```
object Tokens = 101
object MaxPerEnvelope = 5
integer Envelopes = intdiv( Tokens, MaxPerEnvelope) --> 21
```

is_even

Test if the supplied integer is a even or odd number.

Signature:

```
is_even(integer test_integer)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `test_integer` : an integer. The item to test.

Returns: An **integer**,
• 1 if its even.
• 0 if its odd.

Example 1:

```
for i = 1 to 10 do
  ? {i, is_even(i)}
end for
-- output ...
-- {1,0}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}
-- {6,1}
-- {7,0}
-- {8,1}
-- {9,0}
-- {10,1}
```

is_even_obj

Test if the supplied Euphoria object is even or odd.

Signature:

```
is_even_obj(object test_object)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `test_object` : any Euphoria object. The item to test.

Returns: An **object**,
• If `test_object` is an integer...
…… ♦ 1 if its even.
…… ♦ 0 if its odd.
• Otherwise if `test_object` is an atom this always returns 0

• otherwise if `test_object` is an sequence it tests each element recursively, returning a sequence of the same structure containing ones and zeros for each element. A 1 means that the element at this position was even otherwise it was odd.

Example 1:

```
for i = 1 to 5 do
  ? {i, is_even_obj(i)}
end for
-- output ...
-- {1,0}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}
```

Example 2:

```
? is_even_obj(3.4) --> 0
```

Example 3:

```
? is_even_obj({{1,2,3}, {{4,5},6,{7,8}},9})
--> {{0,1,0},{{1,0},1,{0,1}},0}
```

larger_of

Returns the larger of two objects.

Signature:

```
larger_of(object objA, object objB)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `objA` : an object.
≡ `objB` : an object.

Returns: Whichever of `objA` and `objB` is the larger one.

Comments: Introduced in v4.0.3

See Also: max, compare, smaller_of

Examples:

```
? larger_of(10, 15.4) -- returns 15.4
? larger_of("cat", "dog") -- returns "dog"
? larger_of("apple", "apes") -- returns "apple"
? larger_of(10, 10) -- returns 10
```

log

Return the natural logarithm of a positive number.

Signature:

```
log(object value)
```

```
<built-in> function
```

Arguments: ≡ `value` : an object, any atom of which `log`() acts upon.

Returns: An **object**, the same shape as `value`. For an atom, the returned atom is its logarithm of base E.

Comments: This function may be applied to an atom or to all elements of a sequence.

To compute the inverse, you can use power(E, x) where E is 2.7182818284590452, or equivalently exp(x). Beware that the logarithm grows very slowly with x, so that exp() grows very fast.

See Also: E, exp, log10

Example 1:

```
 a = log(100)
 -- a is 4.60517
```

log10

Return the base 10 logarithm of a number.

Signature:

```
log10(object x1)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom of which will be converted, no matter how deeply nested.

Returns: An **object**, the same shape as `value`. When `value` is an atom, raising 10 to the returned atom yields `value` back.

Comments: This function may be applied to an atom or to all elements of a sequence.

`log10`() is proportional to `log`() by a factor of `1/log(10)`, which is about `0.435` .

See Also: log

Example 1:

```
 a = log10(12)
 -- a is 2.48490665
```

max

Computes the maximum value among all the argument's elements

Signature:

```
max(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `values` : an object, all atoms of which will be inspected, no matter how deeply nested.

Returns: An **atom**, the maximum of all atoms in flatten(`values`).

Comments: This function may be applied to an atom or to a sequence of any shape.

See Also: min, compare, flatten

Example 1:

```
 a = max({10,15.4,3})
 -- a is 15.4
```

min

Computes the minimum value among all the argument's elements

Signature:

```
min(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `values` : an object, all atoms of which will be inspected, no matter how deeply nested.

Returns: An **atom**, the minimum of all atoms in flatten(`values`).

Comments: This function may be applied to an atom or to a sequence of any shape.

Example 1:

```
a = min({10,15.4,3})
-- a is 3
```

mod

Compute the remainder of the division of two objects using floored division.

Signature:

```
mod(object x, object y)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `dividend` : any Euphoria object.
≡ `divisor` : any Euphoria object.

Returns: An **object**, the shape of which depends on `dividend`'s and `divisor`'s. For two atoms, this is the remainder of dividing `dividend` by `divisor`, with `divisor`'s sign.

Comments:

• There is a integer `N` such that `dividend` = N * `divisor` + result.
• The result is non-negative and has lesser magnitude than `divisor`. n needs not fit in an Euphoria integer.
• The result has the same sign as the dividend.
• The arguments to this function may be atoms or sequences. The rules for operations on sequences apply, and determine the shape of the returned object.
• When both arguments have the same sign, mod() and remainder() return the same result.
• This differs from remainder() in that when the operands' signs are different this function rounds `dividend/divisior` away from zero whereas remainder() rounds towards zero.

See Also: remainder, Relational operators, Operations on sequences

Example 1:

```
a = mod(9, 4)
-- a is 1
```

Example 2:

```
s = mod({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1,-0.1,1,-2.5}
```

Example 3:

```
s = mod({17, 12, 34}, 16)
```

```
-- s is {1, 12, 2}
```

Example 4:

```
s = mod(16, {2, 3, 5})
-- s is {0, 1, 1}
```

not_bits

Perform the bitwise NOT operation on each bit in an object. A bit in the result will be 1

Signature:

```
not_bits(object a)
```

```
<built-in> function
```

Arguments: ≡ `a` : the object to invert the bits of.

Returns: An **object**, the same shape as `a`. Each bit in an atom of the result is the reverse of the corresponding bit inside `a`.

Comments: The argument to this function may be an atom or a sequence.

The argument must be representable as a 32-bit number, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as atom, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

A simple equality holds for an atom `a`: `a + not_bits(a) = -1`.

See Also: and_bits, or_bits, xor_bits, int_to_bits

Example 1:

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFF08 interpreted as a negative number)
```

or_all

Or's together all atoms in the argument, no matter how deeply nested.

Signature:

```
or_all(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `values` : an object, all atoms of which will be added up, no matter how nested.

Returns: An **atom**, the result of or'ing all atoms in flatten(`values`).

Comments: This function may be applied to an atom or to all elements of a sequence. It performs or_bits() operations repeatedly.

See Also: sum, product, or_bits

Example 1:

```
a = sum({10, 7, 35})
-- a is 47
```

or_bits

Perform the bitwise OR operation on corresponding bits in two objects. A bit in the

Signature:

```
or_bits(object a, object b)
```

```
<built-in> function
```

Arguments:
- ≡ `a` : one of the objects involved
- ≡ `b` : the second object

Returns: An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwise OR between atoms on both objects.

Comments: The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as atom, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

See Also: and_bits, xor_bits, not_bits, int_to_bits

Example 1:

```
a = or_bits(#0F0F0000, #12345678)
-- a is #1F3F5678
```

Example 2:

```
a = or_bits(#FF, {#123456, #876543, #2211})
-- a is {#1234FF, #8765FF, #22FF}
```

power

Raise a base value to some power.

Signature:

```
power(object base, object exponent)
```

```
<built-in> function
```

Arguments:
- ≡ `base` : an object, the value(s) to raise to some power.
- ≡ `exponent` : an object, the exponent(s) to apply to `base`.

Returns: An **object**, the shape of which depends on `base`'s and `exponent`'s. For two atoms, this will be `base` raised to the power `exponent`.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply.

Powers of 2 are calculated very efficiently.

Other languages have a ** or ^ operator to perform the same action. But they don't have sequences.

See Also: log, Operations on sequences

Example 1:

```
? power(5, 2)
-- 25 is printed
```

Example 2:

```
? power({5, 4, 3.5}, {2, 1, -0.5})
-- {25, 4, 0.534522} is printed
```

Example 3:

```
? power(2, {1, 2, 3, 4})
-- {2, 4, 8, 16}
```

Example 4:

```
? power({1, 2, 3, 4}, 2)
-- {1, 4, 9, 16}
```

powof2

Tests for power of 2

Signature:

```
powof2(object p)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `p` : an object. The item to test. This can be an integer, atom or sequence.

Returns: An **integer**,
• 1 for each item in `p` that is a power of two, eg. 2,4,8,16,32, ...
• 0 for each item in `p` that is **not** a power of two, eg. 3, 54.322, -2

Example 1:

```
for i = 1 to 10 do
  ? {i, powof2(i)}
end for
-- output ...
-- {1,1}
-- {2,1}
-- {3,0}
-- {4,1}
-- {5,0}
-- {6,0}
-- {7,0}
-- {8,1}
-- {9,0}
-- {10,0}
```

product

Compute the product of all the atom in the argument, no matter how deeply nested.

Signature:

```
product(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `values` : an object, all atoms of which will be multiplied up, no matter how nested.

Returns: An **atom**, the product of all atoms in flatten(`values`).

Comments: This function may be applied to an atom or to all elements of a sequence

See Also: sum, or_all

Example 1:

```
a = product({10, 20, 30})
-- a is 6000

a = product({10.5, {11.2} , 8.1})
-- a is 952.56
```

rad2deg

Convert an angle measured in radians to an angle measured in degrees

Signature:

```
rad2deg(object x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `angle` : an object, all atoms of which will be converted, no matter how deeply nested.

Returns: An **object**, the same shape as `angle`, all atoms of which were multiplied by 180/PI.

Comments: This function may be applied to an atom or sequence. A flat angle is PI radians and 180 degrees.

arcsin(), arccos() and arctan() return angles in radians.

See Also: deg2rad

Example 1:

```
x = rad2deg(3.385938749)
-- x is 194
```

remainder

Compute the remainder of the division of two objects using truncated division.

Signature:

```
remainder(object dividend, object divisor)
```

```
<built-in> function
```

Arguments: ≡ `dividend` : any Euphoria object.
≡ `divisor` : any Euphoria object.

Returns: An **object**, the shape of which depends on `dividend`'s and `divisor`'s. For two atoms, this is the remainder of dividing `dividend` by `divisor`, with `dividend`'s sign.

Comments:

- There is a integer `N` such that `dividend` = `N` * `divisor` + result.
- The result has the sign of `dividend` and lesser magnitude than `divisor`.
- The result has the same sign as the dividend.
- This differs from mod() in that when the operands' signs are different this function rounds `dividend/divisior` towards zero whereas mod() rounds away from zero.

The arguments to this function may be atoms or sequences. The rules for operations on sequences apply, and determine the shape of the returned object.

See Also: mod, Relational operators, Operations on sequences

Example 1:

```
a = remainder(9, 4)
-- a is 1
```

Example 2:

```
s = remainder({81, -3.5, -9, 5.5}, {8, -1.7, 2, -4})
-- s is {1, -0.1, -1, 1.5}
```

Example 3:

```
s = remainder({17, 12, 34}, 16)
-- s is {1, 12, 2}
```

Example 4:

```
s = remainder(16, {2, 3, 5})
-- s is {0, 1, 1}
```

rotate_bits

Rotates the bits in the input value by the specified distance.

Signature:

```
rotate_bits(object source_number, integer shift_distance)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `source_number` : object: value(s) whose bits will be be rotated.
≡ `shift_distance` : integer: number of bits to be moved by.

Returns: Atom(s) containing a 32-bit integer. A single atom in `source_number` is an atom, or a sequence in the same form as `source_number` containing 32-bit integers.

Comments:

- If `source_number` is a sequence, each element is rotated.
- The value(s) in `source_number` are first truncated to a 32-bit integer.
- The output is truncated to a 32-bit integer.
- If `shift_distance` is negative, the bits in `source_number` are rotated left.
- If `shift_distance` is positive, the bits in `source_number` are rotated right.
- If `shift_distance` is zero, the bits in `source_number` are not rotated.

See Also: shift_bits

Arithmetics

Example 1:

```
? rotate_bits(7, -3) --> 56
? rotate_bits(0, -9) --> 0
? rotate_bits(4, -7) --> 512
? rotate_bits(8, -4) --> 128
? rotate_bits(0xFE427AAC, -7) --> 0x213D567F
? rotate_bits(-7, -3) --> -49  which is 0xFFFFFFCF
? rotate_bits(131, 0) --> 131
? rotate_bits(184.464, 0) --> 184
? rotate_bits(999_999_999_999_999, 0) --> -1530494977 which is 0xA4C67FFF
? rotate_bits(184, 3) -- 23
? rotate_bits(48, 2) --> 12
? rotate_bits(121, 3) --> 536870927
? rotate_bits(0xFE427AAC, 7) -->  0x59FC84F5
? rotate_bits(-7, 3) --> 0x3FFFFFFF
? rotate_bits({48, 121}, 2) --> {12, 1073741854}
```

round

Return the argument's elements rounded to some precision

Signature:

```
round(object a, object precision = 1)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom of which will be acted upon, no matter how deeply nested.
≡ `precision` : an object, the rounding precision(s). If not passed, this defaults to 1.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is that atom rounded to the nearest integer multiple of 1/`precision`.

Comments: This function may be applied to an atom or to all elements of a sequence.

See Also: floor, ceil

Example 1:

```
 round(5.2) -- 5
 round({4.12, 4.67, -5.8, -5.21}, 10) -- {4.1, 4.7, -5.8, -5.2}
 round(12.2512, 100) -- 12.25
```

shift_bits

Moves the bits in the input value by the specified distance.

Signature:

```
shift_bits(object source_number, integer shift_distance)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `source_number` : object: The value(s) whose bits will be be moved.
≡ `shift_distance` : integer: number of bits to be moved by.

Returns: Atom(s) containing a 32-bit integer. A single atom in `source_number` is an atom, or a sequence in the same form as `source_number` containing 32-bit integers.

Comments:

- If `source_number` is a sequence, each element is shifted.
- The value(s) in `source_number` are first truncated to a 32-bit integer.
- The output is truncated to a 32-bit integer.
- Vacated bits are replaced with zero.
- If `shift_distance` is negative, the bits in `source_number` are moved left.
- If `shift_distance` is positive, the bits in `source_number` are moved right.
- If `shift_distance` is zero, the bits in `source_number` are not moved.

See Also: rotate_bits

Example 1:

```
 ? shift_bits((7, -3) --> 56
 ? shift_bits((0, -9) --> 0
 ? shift_bits((4, -7) --> 512
 ? shift_bits((8, -4) --> 128
 ? shift_bits(0xFE427AAC, -7) --> 0x213D5600
 ? shift_bits((-7, -3) --> -56  which is 0xFFFFFFC8
 ? shift_bits((131, 0) --> 131
 ? shift_bits((184.464, 0) --> 184
 ? shift_bits((999_999_999_999_999, 0) --> -1530494977 which is 0xA4C67FFF
 ? shift_bits((184, 3) -- 23
 ? shift_bits((48, 2) --> 12
 ? shift_bits((121, 3) --> 15
 ? shift_bits((0xFE427AAC, 7) -->  0x01FC84F5
 ? shift_bits((-7, 3) --> 0x1FFFFFFF
 ? shift_bits({48, 121}, 2) --> {12, 30}
```

sign

Return -1, 0 or 1 for each element according to it being negative, zero or positive

Signature:

```
sign(object a)


public function
include math.e
namespace math
```

Arguments: ≡ `value` : an object, each atom of which will be acted upon, no matter how deeply nested.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is -1 if `value` is less than zero, 1 if greater and 0 if equal.

Comments: This function may be applied to an atom or to all elements of a sequence.

For an atom, `sign(x)` is the same as compare(x,0).

See Also: compare

Example 1:

```
 i = sign(5)
 i is 1

 i = sign(0)
 -- i is 0

 i = sign(-2)
 -- i is -1
```

sin

Return the sine of an angle expressed in radians

Signature:

```
sin(object angle)


<built-in> function
```

Arguments: ≡ `angle` : an object, each atom in which will be acted upon.

Returns: An **object**, the same shape as `angle`. When `angle` is an atom, the result is the sine of `angle`.

Comments: This function may be applied to an atom or to all elements of a sequence.

The sine of an angle is an atom between -1 and 1 inclusive. 0.0 is hit by integer multiples of PI only.

See Also: cos, arcsin, PI, deg2rad

Example 1:

```
 sin_x = sin({0.5, 0.9, 0.11})
 -- sin_x is {.479, .783, .110}
```

sinh

Computes the hyperbolic sine of an object.

Signature:

```
sinh(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic sine grows like the exponential function.

For all reals, `power(cosh(x), 2) - power(sinh(x), 2) = 1`. Compare with ordinary trigonometry.

See Also: cosh, sin, arcsinh

Example 1:

```
 ? sinh(LN2) -- prints out 0.75
```

smaller_of

Returns the smaller of two objects.

Signature:

```
smaller_of(object objA, object objB)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `objA` : an object.
≡ `objB` : an object.

Returns: Whichever of `objA` and `objB` is the smaller one.

Comments: Introduced in v4.0.3

See Also: min, compare, larger_of

Examples:

```
 ? smaller_of(10, 15.4) -- returns 10
 ? smaller_of("cat", "dog") -- returns "cat"
 ? smaller_of("apple", "apes") -- returns "apes"
 ? smaller_of(10, 10) -- returns 10
```

sqrt

Calculate the square root of a number.

Signature:

```
sqrt(object value)
```

```
<built-in> function
```

Arguments: ≡ `value` : an object, each atom in which will be acted upon.

Returns: An **object**, the same shape as `value`. When `value` is an atom, the result is the positive atom whose square is `value`.

Comments: This function may be applied to an atom or to all elements of a sequence.

See Also: power, Operations on sequences

Example 1:

```
r = sqrt(16)
-- r is 4
```

sum

Compute the sum of all atoms in the argument, no matter how deeply nested

Signature:

```
sum(object a)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `values` : an object, all atoms of which will be added up, no matter how nested.

Returns: An **atom**, the sum of all atoms in flatten(`values`).

Comments: This function may be applied to an atom or to all elements of a sequence

See Also: product, or_all

Example 1:

```
a = sum({10, 20, 30})
-- a is 60

a = sum({10.5, {11.2} , 8.1})
-- a is 29.8
```

tan

Return the tangent of an angle, or a sequence of angles.

Signature:

```
tan(object angle)
```

```
<built-in> function
```

Arguments: ≡ `angle` : an object, each atom of which will be converted, no matter how deeply nested.

Returns: An **object**, of the same shape as `angle`. Each atom in the flattened `angle` is replaced by its tangent.

Comments: This function may be applied to an atom or to all elements of a sequence of arbitrary shape, recursively.

See Also: sin, cos, arctan

Example 1:

```
t = tan(1.0)
-- t is 1.55741
```

tanh

Computes the hyperbolic tangent of an object.

Signature:

```
tanh(object a)
```

```
public function
```

```
include math.e
namespace math
```

Arguments: ≡ `x` : the object to process.

Returns: An **object**, the same shape as `x`, each atom of which was acted upon.

Comments: The hyperbolic tangent takes values from -1 to +1.

`tanh`() is the ratio `sinh() / cosh()`. Compare with ordinary trigonometry.

See Also: cosh, sinh, tan, arctanh

Example 1:

```
 ? tanh(LN2) -- prints out 0.6
```

trunc

Return the integer portion of a number.

Signature:

```
trunc(object x)
```

```
public function
include math.e
namespace math
```

Arguments: ≡ `value` : any Euphoria object.

Returns: An **object**, the shape of which depends on `values`'s. Each item in the returned object will be an integer. These are the same corresponding items in `value` except with any fractional portion removed.

Comments:

• This is essentially done by always rounding towards zero. The floor() function rounds towards negative infinity, which means it rounds towards zero for positive values and away from zero for negative values.
• Note that `trunc(x) + frac(x) = x`

See Also: floor frac

Example 1:

```
 a = trunc(9.4)
 -- a is 9
```

Example 2:

```
 s = trunc({81, -3.5, -9.999, 5.5})
 -- s is {81,-3, -9, 5}
```

xor_bits

Perform the bitwise XOR operation on corresponding bits in two objects. A bit in the

Signature:

```
xor_bits(object a, object b)
```

```
<built-in> function
```

Arguments: ≡ `a` : one of the objects involved
≡ `b` : the second object

Returns: An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by bitwisel XOR between atoms on both objects.

Comments: The arguments must be representable as 32-bit numbers, either signed or unsigned.

If you intend to manipulate full 32-bit values, you should declare your variables as atom, rather than integer. Euphoria's integer type is limited to 31-bits.

Results are treated as signed numbers. They will be negative when the highest-order bit is 1.

See Also: and_bits, or_bits, not_bits, int_to_bits

Example 1:

```
a = xor_bits(#0110, #1010)
-- a is #1100
```

---

mathcons

Constants
- PI
- QUARTPI
- HALFPI
- TWOPI
- PISQR
- INVSQ2PI
- PHI
- E
- LN2
- INVLN2
- LN10
- INVLN10
- SQRT2
- HALFSQRT2
- SQRT3
- DEGREES_TO_RADIANS
- RADIANS_TO_DEGREES
- EULER_GAMMA
- SQRTE
- PINF
- MINF
- SQRT5

---

mathcons API

DEGREES_TO_RADIANS

Conversion factor: Degrees to Radians = PI / 180

Signature:

```
DEGREES_TO_RADIANS
```

```
public constant
include mathcons.e
namespace mathcons
```

E

Euler (e)The base of the natural logarithm.

Signature:

```
E
```

```
public constant
include mathcons.e
namespace mathcons
```

EULER_GAMMA

Gamma (Euler Gamma)

Signature:

```
EULER_GAMMA
```

```
public constant
include mathcons.e
namespace mathcons
```

HALFPI

Half of PI

Signature:

```
HALFPI
```

```
public constant
include mathcons.e
namespace mathcons
```

HALFSQRT2

sqrt(2)/ 2

Signature:

```
HALFSQRT2
```

```
public constant
include mathcons.e
namespace mathcons
```

INVLN10

1 / ln(10)

Signature:

```
INVLN10
```

```
public constant
include mathcons.e
namespace mathcons
```

INVLN2

Signature: 1 / (ln(2))

```
INVLN2
```

```
public constant
include mathcons.e
namespace mathcons
```

INVSQ2PI

Signature: 1 / (sqrt(2PI))

```
INVSQ2PI
```

```
public constant
include mathcons.e
namespace mathcons
```

LN10

Signature: ln(10) :: 10 = power(E, LN10)

```
LN10
```

```
public constant
include mathcons.e
namespace mathcons
```

LN2

Signature: ln(2) :: 2 = power(E, LN2)

```
LN2
```

```
public constant
include mathcons.e
namespace mathcons
```

MINF

Signature: Negative Infinity

```
MINF
```

```
public constant
include mathcons.e
```

```
namespace mathcons
```

PHI

phi => Golden Ratio = (1 + sqrt(5)) / 2

Signature:

```
PHI
```

```
public constant
include mathcons.e
namespace mathcons
```

PI

PI is the ratio of a circle's circumference to it's diameter.

Signature:

```
PI
```

```
public constant
include mathcons.e
namespace mathcons
```

PINF

Positive Infinity

Signature:

```
PINF
```

```
public constant
include mathcons.e
namespace mathcons
```

PISQR

PI ^ 2

Signature:

```
PISQR
```

```
public constant
include mathcons.e
namespace mathcons
```

QUARTPI

Quarter of PI

Signature:

```
QUARTPI
```

```
public constant
```

```
include mathcons.e
namespace mathcons
```

RADIANS_TO_DEGREES

Conversion factor: Radians to Degrees = 180 / PI

Signature:

```
RADIANS_TO_DEGREES
```

```
public constant
include mathcons.e
namespace mathcons
```

SQRT2

sqrt(2)

Signature:

```
SQRT2
```

```
public constant
include mathcons.e
namespace mathcons
```

SQRT3

Square root of 3

Signature:

```
SQRT3
```

```
public constant
include mathcons.e
namespace mathcons
```

SQRT5

sqrt(5)

Signature:

```
SQRT5
```

```
public constant
include mathcons.e
namespace mathcons
```

SQRTE

sqrt(e)

Signature:

```
SQRTE
```

```
public constant
include mathcons.e
namespace mathcons
```

TWOPI

Two times PI

Signature:

```
TWOPI
```

```
public constant
include mathcons.e
namespace mathcons
```

memconst

Microsoft Windows Memory Protection Constants	
	PAGE_EXECUTE PAGE_EXECUTE_READ PAGE_EXECUTE_READWRITE PAGE_EXECUTE_WRITECOPY PAGE_WRITECOPY PAGE_READWRITE PAGE_READONLY PAGE_NOACCESS
Standard Library Memory Protection Constants	
	PAGE_NONE PAGE_READ_EXECUTE PAGE_READ_WRITE PAGE_READ PAGE_READ_WRITE_EXECUTE PAGE_WRITE_EXECUTE_COPY PAGE_WRITE_COPY

Microsoft Windows Memory Protection Constants microsoftsmemoryprotectionconstants These Memory Protection constants are as provided by Microsoft.

Standard Library Memory Protection Constants stardardlibrarymemoryprotectionconstants

Memory Protection Constants are the same constants names and meaning across all platforms yet possibly of different numeric value. They are only necessary for allocate_protect

The constant names are created like this: You have four aspects of protection READ, WRITE, EXECUTE and COPY. You take the word PAGE and you concatonate an underscore and the aspect in the order above. For example: PAGE_WRITE_EXECUTE The sole exception to this nomenclature is when you will have no acesss to the page the constant is called PAGE_NONE.

memconst API

PAGE_EXECUTE

You may run the data in this page

Signature:

```
PAGE_EXECUTE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_EXECUTE_READ

You may read or run the data

Signature:

```
PAGE_EXECUTE_READ
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_EXECUTE_READWRITE

You may run, read or write in this page

Signature:

```
PAGE_EXECUTE_READWRITE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_EXECUTE_WRITECOPY

You may run or write in this page

Signature:

```
PAGE_EXECUTE_WRITECOPY
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_NOACCESS

You have no access to this page

Signature:

```
PAGE_NOACCESS
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_NONE

You have no access to this page.

Signature:

```
PAGE_NONE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READ

You may only read to this page

Signature:

```
PAGE_READ
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READONLY

You may only read data in this page

Signature:

```
PAGE_READONLY
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READWRITE

You may read or write in this page.

Signature:

```
PAGE_READWRITE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READ_EXECUTE

You may read or run the data

Signature:

```
PAGE_READ_EXECUTE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READ_WRITE

Signature: You may read or write to this page

```
PAGE_READ_WRITE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_READ_WRITE_EXECUTE

Signature: You may run, read or write in this page

```
PAGE_READ_WRITE_EXECUTE
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_WRITECOPY

Signature: You may write to this page.

```
PAGE_WRITECOPY
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_WRITE_COPY

Signature: You may write to this page. Data

```
PAGE_WRITE_COPY
```

```
public constant
include memconst.e
namespace memconst
```

PAGE_WRITE_EXECUTE_COPY

Signature: You may run or write to this page. Data

```
PAGE_WRITE_EXECUTE_COPY
```

```
public constant
include memconst.e
namespace memconst
```

memory

Usage Notes

Positive integer type

Machine address type

memory API

os

	CMD_SWITCHES
Operating System Constants	
	WIN32
	WINDOWS
	LINUX
	OSX
	OPENBSD
	NETBSD
	FREEBSD
Environment.	
	instance
	get_pid
	uname
	is_win_nt
	getenv
	setenv
	unsetenv
	platform
Interacting with the OS	
	system
	system_exec
Miscellaneous	
	sleep

os API

CMD_SWITCHES

Signature:

```
CMD_SWITCHES
```

```
public constant
include os.e
namespace os
```

FREEBSD

These constants are returned by the platform function.

Signature:

```
FREEBSD
```

```
public enum
include os.e
namespace os
```

LINUX

Signature:

```
LINUX
```

```
public enum
include os.e
namespace os
```

NETBSD

Signature:

```
NETBSD
```

```
public enum
include os.e
namespace os
```

OPENBSD

Signature:

```
OPENBSD
```

```
public enum
include os.e
namespace os
```

OSX

Signature:

```
OSX
```

```
public enum
include os.e
namespace os
```

WIN32

Signature:

```
WIN32
```

```
public enum
include os.e
namespace os
```

WINDOWS

Signature:

```
WINDOWS
```

```
public enum
include os.e
namespace os
```

get_pid

returns the ID of the current Process (pid).

Signature:

```
get_pid()
```

```
public function
include os.e
namespace os
```

Returns: An atom: The current process' id.

Example 1:

```
 mypid = get_pid()
```

getenv

returns the value of an environment variable.

Signature:

```
getenv(sequence var_name)
```

```
<built-in> function
```

Arguments: ≡ `var_name` : a string, the name of the variable being queried.

Returns: An **object**, -1 if the variable does not exist, else a sequence holding its value.

Comments: Both the argument and the return value, may, or may not be, case sensitive. You might need to test this on your own system.

See Also: setenv, command_line

Example: <

instance

returns `hInstance` on *windows* and Process ID (pid) on *unix*.

Signature:

```
instance()
```

```
public function
include os.e
namespace os
```

Comments: On *windows* the `hInstance` can be passed around to various *windows* routines.

is_win_nt

tests if the host system is a newer Windows version (NT/2K/XP/Vista).

Signature:

```
is_win_nt()
```

```
public function
include os.e
namespace os
```

Returns: An **integer**, 1 if host system is a newer Windows (NT/2K/XP/Vista), else 0.

platform

Indicates the platform that the program is being executed on.

Signature:

```
platform()
```

```
<built-in> function
```

Returns: An **integer**,

WIN32 = WINDOWS,
LINUX,
FREEBSD,
OSX,
OPENBSD,
NETBSD,
FREEBSD

Comments: The ifdef statement is much more versatile and in most cases supersedes `platform`().

`platform`() used to be the way to execute different code depending on which platform the program is running on. Additional platforms will be added as Euphoria is ported to new machines and operating environments.

See Also: Platform-Specific Issues, ifdef statement

Example 1:

```
ifdef WINDOWS then
    -- call system Beep routine
    err = c_func(Beep, {0,0})
elsedef
    -- do nothing (Linux/FreeBSD)
end if
```

setenv

sets an environment variable.

Signature:

```
setenv(sequence name, sequence val, integer overwrite = 1)
```

```
public function
include os.e
namespace os
```

Arguments: ≡ `name` : a string, the environment variable name
≡ `val` : a string, the value to set to
≡ `overwrite` : an integer, nonzero to overwrite an existing variable, 0 to disallow this.

See Also: getenv, unsetenv

Example 1:

```
 ? setenv("NAME", "John Doe")
 ? setenv("NAME", "Jane Doe")
 ? setenv("NAME", "Jim Doe", 0)
```

sleep

suspends thread execution for `t` seconds.

Signature:

```
sleep(atom t)
```

```
public procedure
include os.e
namespace os
```

Arguments: ≡ `t` : an atom, the number of seconds for which to sleep.

Comments: The operating system will suspend your process and schedule other processes.

With multiple tasks, the whole program sleeps, not just the current task. To make just the current task sleep, you can call `task_schedule(task_self(), {i, i})` and then execute task_yield(). Another option is to call task_delay().

See Also: task_schedule, task_yield, task_delay

Example 1:

```
 puts(1, "Waiting 15 seconds and a quarter...\n")
 sleep(15.25)
 puts(1, "Done.\n")
```

system

passes a command string to the operating system command interpreter.

Signature:

```
system(sequence command, integer mode=0)
```

```
<built-in> procedure
```

Arguments: ≡ `command` : a string to be passed to the shell
≡ `mode` : an integer, indicating the manner in which to return from the call.

Comments: Allowable values for `mode` are:
• 0: the previous graphics mode is restored and the screen is cleared.
• 1: a beep sound will be made and the program will wait for the user to press a key

before the previous graphics mode is restored.
• 2: the graphics mode is not restored and the screen is not cleared.

`mode` = 2 should only be used when it is known that the command executed by `system`() will not change the graphics mode.

You can use Euphoria as a sophisticated "batch" (.bat) language by making calls to `system`() and `system_exec`().

`system`() will start a new command shell.

`system`() allows you to use command-line redirection of standard input and output in `command`.

See Also: system_exec, command_line, current_dir, getenv

Example 1:

```
system("copy temp.txt a:\\temp.bak", 2)
-- note use of double backslash in literal string to get
-- single backslash
```

Example 2:

```
system("eui \\test\\myprog.ex < indata > outdata", 2)
-- executes myprog by redirecting standard input and
-- standard output
```

system_exec

tries to run the a shell executable command.

Signature:

```
system_exec(sequence command, integer mode=0)
```

```
<built-in> function
```

Arguments: ≡ `command` : a string to be passed to the shell, representing an executable command
≡ `mode` : an integer, indicating the manner in which to return from the call.

Returns: An **integer**, basically the exit/return code from the called process.

Comments: Allowable values for `mode` are:
• 0 -- the previous graphics mode is restored and the screen is cleared.
• 1 -- a beep sound will be made and the program will wait for the user to press a key before the previous graphics mode is restored.
• 2 -- the graphics mode is not restored and the screen is not cleared.

If it is not possible to run the program, `system_exec`() will return -1.

On *windows* `system_exec`() will only run .exe and .com programs. To run .bat files, or built-in shell commands, you need system(). Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

On *windows* `system_exec`() does not allow the use of command-line redirection in `command`. Nor does it allow you to quote strings that contain blanks, such as file names.

exit codes from *windows* programs are normally in the range 0 to 255, with 0 indicating "success".

You can run a Euphoria program using `system_exec`(). A Euphoria program can return an exit code using abort().

`system_exec`() does not start a new command shell.

See Also: system, abort

Example 1:

```
integer exit_code
exit_code = system_exec("xcopy temp1.dat temp2.dat", 2)

if exit_code = -1 then
    puts(2, "\n couldn't run xcopy.exe\n")
elsif exit_code = 0 then
    puts(2, "\n xcopy succeeded\n")
else
    printf(2, "\n xcopy failed with code %d\n", exit_code)
end if
```

Example 2:

```
-- executes myprog with two file names as arguments
if system_exec("eui \\test\\myprog.ex indata outdata", 2) then
    puts(2, "failure!\n")
end if
```

uname

retrieves the name of the host OS.

Signature:

```
uname()
```

```
public function
include os.e
namespace os
```

Returns: A **sequence**, starting with the OS name. If identification fails, returns an OS name of UNKNOWN. Extra information depends on the OS.

Comments: On *unix* returns the same information as the uname() syscall in the same order as the struct utsname. This information is:

OS Name/Kernel Name
Local Hostname
Kernel Version/Kernel Release
Kernel Specific Version information (This is usually the date that the
kernel was compiled on and the name of the host that performed the compiling.)
Architecture Name (Usually a string of i386 vs x86_64 vs ARM vs etc)

On *windows* returns the following in order:

Windows Platform (out of WinCE, Win9x, WinNT, Win32s, or Unknown Windows)
Name of Windows OS (Windows 3.1, Win95, WinXP, etc)
Platform Number
Build Number
Minor OS version number
Major OS version number

On UNKNOWN, returns an OS name of "UNKNOWN". No other information is returned.

Returns a string of "" if an internal error has occured.

On *unix* M_UNAME is defined as a machine_func() and this is passed to the C backend. If the M_UNAME call fails, the raw machine_func() returns -1. On non-*unix* platforms, calling the machine_func() directly returns 0.

unsetenv

unsets an environment variable.

Signature:

```
unsetenv(sequence env)
```

```
public function
include os.e
namespace os
```

Arguments: ≡ `name` : name of environment variable to unset

See Also: setenv, getenv

Example 1:

```
 ? unsetenv("NAME")
```

pipeio

Notes

Accessor Constants

- STDIN
- STDOUT
- STDERR
- PID
- PARENT
- CHILD

Opening and Closing

- process
- close
- kill

Read/Write Process

- read
- write
- error_no
- create
- exec

Notes Due to a bug, Euphoria does not handle STDERR properly STDERR cannot captured for Euphoria programs (other programs will work fully) The I/O functions currently work with file handles, a future version might wrap them in streams so that they can be used directly alongside other file/socket/other-streams with a stream_select() function.

pipeio API

CHILD

Set of pipes that are given to the child - should not be used by the parent

Signature:

```
CHILD
```

```
public enum
include pipeio.e
namespace pipeio
```

PARENT

Set of pipes that are for the use of the parent

Signature:

```
PARENT
```

```
public enum
include pipeio.e
namespace pipeio
```

PID

Process ID

Signature:

```
PID
```

```
public enum
include pipeio.e
namespace pipeio
```

STDERR

Child processes standard error

Signature:

```
STDERR
```

```
public enum
include pipeio.e
namespace pipeio
```

STDIN

Child processes standard input

Signature:

```
STDIN
```

```
public enum
include pipeio.e
namespace pipeio
```

STDOUT

Child processes standard output

Signature:

```
STDOUT
```

```
public enum
include pipeio.e
namespace pipeio
```

close

Close handle fd

Signature:

```
close(atom fd)
```

```
public function
include pipeio.e
namespace pipeio
```

Returns: An **integer**, 0 on success, -1 on failure

Example 1:

```
integer status = pipeio:close(p[STDIN])
```

create

creates pipes for inter-process communication.

Signature:

```
create()
```

```
public function
include pipeio.e
namespace pipeio
```

Returns: A **handle**, process handles { {parent side pipes},{child side pipes} }

Example 1:

```
object p = exec("dir", create())
```

error_no

gets the error no. from last call to a pipe function

Signature:

```
error_no()
```

```
public function
include pipeio.e
namespace pipeio
```

Comments: Value returned will be OS-specific, and is not always set on *windows* at least.

Example 1:

```
integer error = error_no()
```

exec

Open process with command line cmd

Signature:

```
exec(sequence cmd, sequence pipe)
```

```
public function
include pipeio.e
namespace pipeio
```

Returns: A **handle**, process handles { PID, STDIN, STDOUT, STDERR }

Example 1:

```
 object p = exec("dir", create())
```

kill

close pipes and kills the process p with signal signal (default 15)

Signature:

```
kill(process p, atom signal = 15)
```

```
public procedure
include pipeio.e
namespace pipeio
```

Platform: *unix*

Comments: Signal is ignored on *windows*.

Example 1:

```
 kill(p)
```

process

Process Type

Signature:

```
process(object o)
```

```
public type
include pipeio.e
namespace pipeio
```

read

reads `bytes` bytes from handle `fd`.

Signature:

```
read(atom fd, integer bytes)
```

```
public function
include pipeio.e
namespace pipeio
```

Returns: A **sequence**, containing data, an empty sequence on EOF or an error code. Similar to get_bytes.

Example 1:

```
 sequence data=read(p[STDOUT],256)
```

write

writes `bytes` to handle `fd`.

Signature:

```
write(atom fd, sequence str)
```

```
public function
include pipeio.e
namespace pipeio
```

Returns: An **integer**, number of bytes written, or -1 on error

Example 1:

```
 integer bytes_written = write(p[STDIN],"Hello World!")
```

---

pretty

- PRETTY_DEFAULT
- DISPLAY_ASCII
- INDENT
- START_COLUMN
- WRAP
- INT_FORMAT
- FP_FORMAT
- MIN_ASCII
- MAX_ASCII
- MAX_LINES
- LINE_BREAKS

Routines

- pretty_print
- pretty_sprint

---

pretty API

DISPLAY_ASCII

Signature:

```
DISPLAY_ASCII
```

```
public enum
include pretty.e
namespace pretty
```

FP_FORMAT

Signature:

```
FP_FORMAT
```

```
public enum
include pretty.e
namespace pretty
```

INDENT

Signature:

```
INDENT
```

```
public enum
include pretty.e
namespace pretty
```

INT_FORMAT

Signature:

```
INT_FORMAT
```

```
public enum
include pretty.e
namespace pretty
```

LINE_BREAKS

Signature:

```
LINE_BREAKS
```

```
public enum
include pretty.e
namespace pretty
```

MAX_ASCII

Signature:

```
MAX_ASCII
```

```
public enum
include pretty.e
namespace pretty
```

MAX_LINES

Signature:

```
MAX_LINES
```

```
public enum
include pretty.e
namespace pretty
```

MIN_ASCII

Signature:

```
MIN_ASCII
```

```
public enum
include pretty.e
namespace pretty
```

PRETTY_DEFAULT

Signature:

```
PRETTY_DEFAULT
```

```
public constant
include pretty.e
namespace pretty
```

START_COLUMN

Signature:

```
START_COLUMN
```

```
public enum
include pretty.e
namespace pretty
```

WRAP

Signature:

```
WRAP
```

```
public enum
include pretty.e
namespace pretty
```

pretty_print

prints an object to a file (or device) showing the object structure using: using braces { , , , }, indentation, and multiple lines.

Signature:

```
pretty_print(integer fn, object x, sequence options = PRETTY_DEFAULT)
```

```
public procedure
include pretty.e
namespace pretty
```

Arguments:
≡ `fn` : an integer, the file/device number to write to
≡ `x` : the object to display/convert to printable form

≡ `options` : is an (up to) 10-element options sequence.

Comments: Pass {} in `options` to select the defaults, or set options as below:

# display ASCII characters:

** 0 -- never ** 1 -- alongside any integers in printable ASCII range (default) ** 2 -- display as "string" when all integers of a sequence are in ASCII range ** 3 -- show strings, and quoted characters (only) for any integers in ASCII range as well as the characters: \t \r \n

# amount to indent for each level of sequence nesting -- default: 2 # column we are starting at -- default: 1 # approximate column to wrap at -- default: 78 # format to use for integers -- default: "%d" # format to use for floating-point numbers -- default: "%.10g" # minimum value for printable ASCII -- default 32 # maximum value for printable ASCII -- default 127 # maximum number of lines to output # line breaks between elements -- default 1 (0 = no line breaks, -1 = line breaks to wrap only)

If the length is less than 10, unspecified options at the end of the sequence will keep the default values. e.g. {0, 5} will choose "never display ASCII", plus 5-character indentation, with defaults for everything else.

The default options can be applied using the public constant `PRETTY_DEFAULT`, and the elements may be accessed using the following public enum:

```
DISPLAY_ASCII
INDENT
START_COLUMN
WRAP
INT_FORMAT
FP_FORMAT
MIN_ASCII
MAX_ASCII
MAX_LINES
LINE_BREAKS
```

The display will start at the current cursor position. Normally you will want to call `pretty_print`() when the cursor is in column 1 (after printing a <code>\n</code> character). If you want to start in a different column, you should call `position`() and specify a value for option [3]. This will ensure that the first and last braces in a sequence line up vertically.

When specifying the format to use for integers and floating-point numbers, you can add some decoration, e.g. "(%d)" or "$ %.2f"

See Also: print, sprint, printf, sprintf, pretty_sprint

Example 1:

```
pretty_print(1, "ABC", {})

{65'A',66'B',67'C'}
```

Example 2:

```
pretty_print(1, {{1,2,3}, {4,5,6}}, {})

{
  {1,2,3},
  {4,5,6}
}
```

Example 3:

```
pretty_print(1, {"Euphoria", "Programming", "Language"}, {2})

{
  "Euphoria",
  "Programming",
  "Language"
}
```

Example 4:

```
puts(1, "word_list = ") -- moves cursor to column 13
pretty_print(1,
    {{"Euphoria", 8, 5.3},
     {"Programming", 11, -2.9},
     {"Language", 8, 9.8}},
     {2, 4, 13, 78, "%03d", "%.3f"}) -- first 6 of 8 options

word_list = {
    {
        "Euphoria",
        008,
        5.300
    },
    {
        "Programming",
        011,
        -2.900
    },
    {
        "Language",
        008,
        9.800
    }
}
```

pretty_sprint

formats an object as a string with the structure shown using: using braces { , , , }, indentation, and multiple lines.

Signature:

```
pretty_sprint(object x, sequence options = PRETTY_DEFAULT)
```

```
public function
include pretty.e
namespace pretty
```

Arguments: ≡ `x` : the object to display
≡ `options` : is an (up to) 10-element options sequence: Pass {} to select the defaults, or set options

Returns: A **sequence**, of printable characters, representing `x` in an human-readable form.

Comments: This function formats objects the same as pretty_print(), but returns the sequence obtained instead of sending it to some file..

See Also: pretty_print, sprint

primes

Routines

- calc_primes
- next_prime
- prime_list

primes API

calc_primes

returns a list of all prime numbers--up to the ceiling value (if it is prime), or up to the next larger prime.

Signature:

```
calc_primes(integer approx_limit, atom time_limit_p = 10)
```

```
public function
include primes.e
namespace primes
```

Arguments:
≡ `approx_limit` : an integer, This is not the upper limit but the last prime returned is the **next** prime after or on this value.
≡ `time_out_p` : an atom, the maximum number of seconds that this function can run for. The default is 10 (ten) seconds.

Returns: A **sequence**, made of prime numbers in increasing order. The last value is the prime number that falls on or just **after** the value of `approx_limit`.

Comments: The returned sequence contains all the prime numbers less than its last element.

The expected list of primes will contain `approx_limit` if it happens to be prime, or will contain the next prime above this value.

If the largest prime value is less than `approx_limit` then the execution time limit was reached before all calculations could finish; one or more of the largest prime values will then be missing from the list.

If the list is incomplete you must increase the time cap limit, or disable the limit by specifying a negative value.

Use the `prime_list` function if the largest prime found must *not exceed* the limit value.

See Also: next_prime prime_list

Example 1:

```
? calc_primes(1000, 5)
-- Note that 1000 is an "approximate" limit.
-- On a very slow computer, you may only get all primes up to say 719.
-- On a faster computer, the last element printed out will be 1009.
-- This call will never take longer than 5 seconds.
```

next_prime

returns the test value (if it is prime) or the next larger prime.

Signature:

```
next_prime(integer n, object fail_signal_p = - 1, atom time_out_p = 1)
```

```
public function
include primes.e
namespace primes
```

Arguments:
≡ `n` : an integer, the starting point for the search
≡ `fail_signal_p` : an integer, used to signal error. Defaults to -1.

Returns: An **integer**, which is prime only if it took less than 1 second to determine the next prime greater or equal to `n`.

Comments: The default value of -1 will alert you about an invalid returned value, since a prime not less than `n` is expected. However, you can pass another value for this parameter.

See Also: calc_primes

Example 1:

```
? next_prime(997)
-- On a very slow computer, you might get -997, but 1009 is expected.
```

prime_list

returns a list of all prime numbers--up to the ceiling value (if it is prime), or up to the largest prime below the ceiling.

Signature:

```
prime_list(integer top_prime_p = 0)
```

```
public function
include primes.e
namespace primes
```

Arguments: ≡ `top_prime_p` : The list will end with the prime less than or equal to this value. If `top_prime_p` is zero, the current list of calculated primes is returned.

Returns: An **sequence**, a list of prime numbers from 2 to <= `top_prime_p`

See Also: calc_primes, next_prime

Example 1:

```
? prime_list( 19 )
--> {2,3,5,11,13,19}
-- the limit is included because it is also a prime
```

Example 2:

```
sequence pList = prime_list(1000)
--> {2,3,5,7,11, ... ,983,991,997}
-- this list contains 169 prime values
-- the largest prime up to or including 1000 is 997
```

rand

rand API

chance

simulates the probability of a desired outcome.

Signature:

```
chance(atom my_limit, atom top_limit = 100)
```

```
public function
include rand.e
namespace random
```

Arguments:
- ≡ `my_limit` : an atom. The desired chance of something happening.
- ≡ `top_limit`: an atom. The maximum chance of something happening. The default is 100.

Returns: an integer. 1 if the desired chance happened otherwise 0.

Comments: This simulates the chance of something happening. For example, if you wnat something to happen with a probablity of 25 times out of 100 times then you code `chance(25)` and if you want something to (most likely) occur 345 times out of 999 times, you code `chance(345, 999)`.

See Also: rnd, roll

Example 1:

```
-- 65% of the days are sunny, so ...
 if chance(65) then
     puts(1, "Today will be a sunny day")
 elsif chance(40) then
     -- And 40% of non-sunny days it will rain.
     puts(1, "It will rain today")
 else
     puts(1, "Today will be a overcast day")
 end if
```

get_rand

retrieves the current values of the random generator's seeds.

Signature:

```
get_rand()
```

```
public function
include rand.e
namespace random
```

Returns: a sequence. A 2-element sequence containing the values of the two internal seeds.

Comments: You can use this to save the current seed values so that you can later reset them back to a known state.

See Also: set_rand

Example 1:

```
sequence seeds
 seeds = get_rand()
 some_func() -- Which might set the seeds to anything.
 set_rand(seeds) -- reset them back to whatever they were
                 -- before calling 'some_func()'.
```

rand

returns a random integral value.

Signature:

```
rand(object maximum)
```

```
<built-in> function
```

Arguments: ≡ `maximum` : an atom, a cap on the value to return.

Returns: An **atom**, from 1 to `maximum`.

Comments:

• The minimum value of `maximum` is 1.
• The maximum value that can possibly be returned is #FFFFFFFF (4_294_967_295)
• This function may be applied to an atom or to all elements of a sequence.
• In order to get reproducible results from this function, you should call set_rand() with a reproducible value prior.

See Also: set_rand, ceil

Example 1:

```
s = rand({10, 20, 30})
-- s might be: {5, 17, 23} or {9, 3, 12} etc.
```

rand_range

returns a random integer from a specified inclusive integer range.

Signature:

```
rand_range(atom lo, atom hi)
```

```
public function
include rand.e
namespace random
```

Arguments: ≡ `lo` : an atom, the lower bound of the range
≡ `hi` : an atom, the upper bound of the range.

Returns: An **atom**, randomly drawn between `lo` and `hi` inclusive.

Comments: This function may be applied to an atom or to all elements of a sequence. In order to get reproducible results from this function, you should call `set_rand`() with a reproducible value prior.

See Also: rand, set_rand, rnd

Example 1:

```
s = rand_range(18, 24)
-- s could be any of: 18, 19, 20, 21, 22, 23 or 24
```

rnd

returns a random floating point number in the range 0 to 1.

Signature:

```
rnd()
```

```
public function
include rand.e
namespace random
```

Arguments: None.

Returns: An **atom**, randomly drawn between 0.0 and 1.0 inclusive.

Comments: In order to get reproducible results from this function, you should call `set_rand`() with a reproducible value prior to calling this.

See Also: rand, set_rand, rand_range

Example 1:

```
set_rand(1001)
s = rnd()
```

```
    -- s is 0.6277338201
```

rnd_1

returns a random floating point number in the range 0 to less than 1.

Signature:

```
rnd_1()
```

```
public function
include rand.e
namespace random
```

Arguments: None.

Returns: An **atom**, randomly drawn between 0.0 and a number less than 1.0

Comments: In order to get reproducible results from this function, you should call `set_rand`() with a reproducible value prior to calling this.

See Also: rand, set_rand, rand_range

Example 1:

```
 set_rand(1001)
 s = rnd_1()
   -- s is 0.6277338201
```

roll

simulates the probability of a dice throw.

Signature:

```
roll(object desired, integer sides = 6)
```

```
public function
include rand.e
namespace random
```

Arguments: ≡ `desired` : an object. One or more desired outcomes.
≡ `sides`: an integer. The number of sides on the dice. Default is 6.

Returns: an integer. 0 if none of the desired outcomes occured, otherwise the face number that was rolled.

Comments: The minimum number of sides is 2 and there is no maximum.

See Also: rnd, chance

Example 1:

```
 res = roll(1, 2)
       --> Simulate a coin toss.
 res = roll({1,6})
       --> Try for a 1 or a 6 from a standard die toss.
 res = roll({1,2,3,4}, 20)
       --> Looking for any number under 5 from a 20-sided die.
```

sample

returns a set of random samples selected from a population set.

Signature:

```
sample(sequence population, integer sample_size,
integer sampling_method = 0)
```

```
public function
include rand.e
namespace random
```

Arguments: ≡ `population` : a sequence. The set of items from which to take a sample.
≡ `sample_size`: an integer. The number of samples to take.
≡ `sampling_method`: an integer. `When < 0, "with-replacement" method used.`
When = 0, "without-replacement" method used and a single set of samples returned.
When > 0, "without-replacement" method used and a sequence containing the set of samples (chosen items) and the set unchosen items, is returned.

Returns: A sequence. When `sampling_method` less than or equal to 0 then this is the set of samples, otherwise it returns a two-element sequence; the first is the samples, and the second is the remainder of the population (in the original order).

Comments: The random sample can be selected using either the "with-replacement" or "without-replacement" methods. When using the "with-replacement" method, after each sample is taken it is returned to the population set so that it could possible be taken again. The "without-replacement" method does not return the sample so these items can only ever be chosen once.

• If `sample_size` is less than 1, an empty set is returned.
• When using "without-replacement" method, if `sample_size` is greater than or equal to the population count, the entire population set is returned, but in a random order.
• When using "with-replacement" method, if `sample_size` can be any positive integer, thus it is possible to return more samples than there are items in the population set as items can be chosen more than once.

Example 1:

```
set_rand("example")
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 1)})
     --> "t"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 5)})
     --> "flukq"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", -1)})
     --> ""
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 26)})
     --> "kghrsxmjoeubaywlzftcpivqnd"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 25)})
    --> "omntrqsbjguaikzywvxflpedc"
```

Example 2:

```
set_rand("example")
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 1, -1)})
     --> "t"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 5, -1)})
     --> "fzycn"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", -1, -1)})
     --> ""
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 26, -1)})
     --> "keeamenuvvfyelqapucerghgfa"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxyz", 45, -1)})
    --> "orwpsaxuwuyrbstqqwfkykujukuzkkuxvzvzniinnpnxm"
```

Example 3:

```
-- Deal 4 hands of 5 cards from a standard deck of cards.
sequence theDeck
sequence hands = {}
sequence rt
function new_deck(integer suits = 4, integer cards_per_suit = 13,
                  integer wilds = 0)
 sequence nd = {}
 for i = 1 to suits do
  for j = 1 to cards_per_suit do
   nd = append(nd, {i,j})
  end for
 end for
 for i = 1 to wilds do
     nd = append(nd, {suits+1 , i})
 end for
 return nd
end function

theDeck = new_deck(4, 13, 2) -- Build the initial deck of cards
for i = 1 to 4 do
 -- Pick out 5 cards and also return the remaining cards.
 rt = sample(theDeck, 5, 1)
 theDeck = rt[2] -- replace the 'deck' with the remaining cards.
 hands = append(hands, rt[1])
end for
```

set_rand

resets the random number generator.

Signature:

```
set_rand(object seed)
```

```
public procedure
include rand.e
namespace random
```

Arguments: ≡ `seed` : an object. The generator uses this initialize itself for the next random number generated. This can be a single integer or atom, or a sequence of two integers, or an empty sequence or any other sort of sequence.

Comments:

• Starting from a `seed`, the values returned by `rand`() are reproducible. This is useful for demos and stress tests based on random data. Normally the numbers returned by the `rand`() function are totally unpredictable, and will be different each time you run your program. Sometimes however you may wish to repeat the same series of numbers, perhaps because you are trying to debug your program, or maybe you want the ability to generate the same output (e.g. a random picture) for your user upon request.
• Internally there are actually two seed values.
…… ♦ When `set_rand()` is called with a single integer or atom, the two internal seeds are derived from the parameter.
…… ♦ When `set_rand()` is called with a sequence of exactly two integers/atoms the internal seeds are set to the parameter values.
…… ♦ When `set_rand()` is called with an empty sequence, the internal seeds are set to random values and are unpredictable. This is how to reset the generator.
…… ♦ When `set_rand()` is called with any other sequence, the internal seeds are set based on the length of the sequence and the hashed value of the sequence.
• Aside from an empty `seed` parameter, this sets the generator to a known state and the random numbers generated after come in a predicable order, though they still appear to be random.

See Also: rand

Example 1:

```
sequence s, t
s = repeat(0, 3)
t = s

set_rand(12345)
s[1] = rand(10)
s[2] = rand(100)
s[3] = rand(1000)

set_rand(12345)  -- same value for set_rand()
t[1] = rand(10)  -- same arguments to rand() as before
t[2] = rand(100)
t[3] = rand(1000)
```

```
        -- at this point s and t will be identical
        set_rand("") -- Reset the generator to an unknown seed.
        t[1] = rand(10)  -- Could be anything now, no way to predict it.
```

---

regex

Introduction

Regular expressions in Euphoria are based on the PCRE (Perl Compatible Regular Expressions) library created by Philip Hazel.

This document will detail the Euphoria interface to Regular Expressions, not really regular expression syntax. It is a very complex subject that many books have been written on. Here are a few good resources online that can help while learning regular expressions.

- EUForum Article
- Perl Regular Expressions Man Page
- Regular Expression Library (user supplied regular expressions for just about any task).
- WikiPedia Regular Expression Article

- Man page of PCRE in HTML

General Use

Many functions take an optional `options` parameter. This parameter can be either a single option constant (see Option Constants), multiple option constants or'ed together into a single atom or a sequence of options, in which the function will take care of ensuring the are or'ed together correctly. Options are like their C equivalents with the 'PCRE_' prefix stripped off. Name spaces disambiguate symbols so we don't need this prefix.

All strings passed into this library must be either 8-bit per character strings or UTF which uses multiple bytes to encode UNICODE characters. You can use UTF8 encoded UNICODE strings when you pass the UTF8 option.

Compile Time and Match Time

When a regular expression object is created via `new` we call also say it get's "compiled." The options you may use for this are called "compile time" option constants. Once the regular expression is created you can use the other functions that take this regular expression and a string. These routines' options are called "match time" option constants. To not set any options at all, do not supply the options argument or supply DEFAULT.

Compile Time Option Constants

The only options that may set at "compile time"; that is, to pass to `new`; are ANCHORED, AUTO_CALLOUT, BSR_ANYCRLF, BSR_UNICODE, CASELESS, DEFAULT, DOLLAR_ENDONLY, DOTALL, DUPNAMES, EXTENDED, EXTRA, FIRSTLINE, MULTILINE, NEWLINE_CR, NEWLINE_LF, NEWLINE_CRLF, NEWLINE_ANY, NEWLINE_ANYCRLF, NO_AUTO_CAPTURE, NO_UTF8_CHECK, UNGREEDY, and UTF8.

Match Time Option Constants

Options that may be set at "match time" are ANCHORED, NEWLINE_CR, NEWLINE_LF, NEWLINE_CRLF, NEWLINE_ANY NEWLINE_ANYCRLF NOTBOL, NOTEOL, NOTEMPTY, NO_UTF8_CHECK. Routines that take match time option constants match, split or replace a regular expression against some string.

---

regex API

ANCHORED

Forces matches to be only from the first place it is asked to try to make a search.

Signature:

```
ANCHORED
```

```
public constant
```

Comments: In C, this is called PCRE_ANCHORED. This is passed to all routines including new.

AUTO_CALLOUT

In C, this is called PCRE_AUTO_CALLOUT.

Signature:

```
AUTO_CALLOUT
```

```
public constant
```

BSR_ANYCRLF

With this option only ASCII new line sequences are recognized as newlines. Other UNICODE

Signature:

```
BSR_ANYCRLF
```

```
public constant
```

BSR_UNICODE

With this option any UNICODE new line sequence is recognized as a newline.

Signature:

```
BSR_UNICODE
```

```
public constant
```

CASELESS

This will make your regular expression matches case insensitive. With this

Signature:

```
CASELESS
```

```
public constant
```

DEFAULT

This is a value used for not setting any flags at all. This can be passed to

Signature:

```
DEFAULT
```

```
public constant
```

DFA_RESTART

This is NOT used by any standard library routine.

Signature:

```
DFA_RESTART
```

```
public constant
```

DFA_SHORTEST

This is NOT used by any standard library routine.

Signature:

```
DFA_SHORTEST
```

```
public constant
```

DOLLAR_ENDONLY

If this bit is set, a dollar sign metacharacter in the pattern matches only

Signature:

```
DOLLAR_ENDONLY
```

```
public constant
```

DOTALL

With this option the '.' character also matches a newline sequence.

Signature:

```
DOTALL
```

```
public constant
```

DUPNAMES

Allow duplicate names for named subpatterns.

Signature:

```
DUPNAMES
```

```
public constant
```

ERROR_BADCOUNT

Signature:

```
ERROR_BADCOUNT
```

```
public constant
include regex.e
namespace regex
```

ERROR_BADMAGIC

Signature:

```
ERROR_BADMAGIC
```

```
public constant
include regex.e
```

```
namespace regex
```

ERROR_BADNEWLINE

Signature:

```
ERROR_BADNEWLINE
```

```
public constant
include regex.e
namespace regex
```

ERROR_BADOPTION

Signature:

```
ERROR_BADOPTION
```

```
public constant
include regex.e
namespace regex
```

ERROR_BADPARTIAL

Signature:

```
ERROR_BADPARTIAL
```

```
public constant
include regex.e
namespace regex
```

ERROR_BADUTF8

Signature:

```
ERROR_BADUTF8
```

```
public constant
include regex.e
namespace regex
```

ERROR_BADUTF8_OFFSET

Signature:

```
ERROR_BADUTF8_OFFSET
```

```
public constant
include regex.e
namespace regex
```

ERROR_CALLOUT

Signature:

```
ERROR_CALLOUT
```

```
public constant
include regex.e
namespace regex
```

ERROR_DFA_RECURSE

Signature:

```
ERROR_DFA_RECURSE
```

```
public constant
include regex.e
namespace regex
```

ERROR_DFA_UCOND

Signature:

```
ERROR_DFA_UCOND
```

```
public constant
include regex.e
namespace regex
```

ERROR_DFA_UITEM

Signature:

```
ERROR_DFA_UITEM
```

```
public constant
include regex.e
namespace regex
```

ERROR_DFA_UMLIMIT

Signature:

```
ERROR_DFA_UMLIMIT
```

```
public constant
include regex.e
namespace regex
```

ERROR_DFA_WSSIZE

Signature:

```
ERROR_DFA_WSSIZE
```

```
public constant
include regex.e
namespace regex
```

ERROR_INTERNAL

Signature:

```
ERROR_INTERNAL
```

```
public constant
include regex.e
namespace regex
```

ERROR_MATCHLIMIT

Signature:

```
ERROR_MATCHLIMIT
```

```
public constant
include regex.e
namespace regex
```

ERROR_NOMATCH

Signature:

```
ERROR_NOMATCH
```

```
public constant
include regex.e
namespace regex
```

ERROR_NOMEMORY

Signature:

```
ERROR_NOMEMORY
```

```
public constant
include regex.e
namespace regex
```

ERROR_NOSUBSTRING

Signature:

```
ERROR_NOSUBSTRING
```

```
public constant
include regex.e
namespace regex
```

ERROR_NULL

Signature:

```
ERROR_NULL
```

```
public constant
include regex.e
namespace regex
```

ERROR_NULLWSLIMIT

Signature:

```
ERROR_NULLWSLIMIT
```

```
public constant
include regex.e
namespace regex
```

ERROR_PARTIAL

Signature:

```
ERROR_PARTIAL
```

```
public constant
include regex.e
namespace regex
```

ERROR_RECURSIONLIMIT

Signature:

```
ERROR_RECURSIONLIMIT
```

```
public constant
include regex.e
namespace regex
```

ERROR_UNKNOWN_NODE

Signature:

```
ERROR_UNKNOWN_NODE
```

```
public constant
include regex.e
namespace regex
```

ERROR_UNKNOWN_OPCODE

Signature:

```
ERROR_UNKNOWN_OPCODE
```

```
public constant
include regex.e
namespace regex
```

EXTENDED

Whitespace and characters beginning with a hash mark to the end of the line

Signature:

```
EXTENDED
```

```
public constant
```

EXTRA

When an alphanumeric follows a backslash(\) has no special meaning an

Signature:

```
EXTRA
```

```
public constant
```

FIRSTLINE

If PCRE_FIRSTLINE is set, the match must happen before or at the first

Signature:

```
FIRSTLINE
```

```
public constant
```

MULTILINE

When MULTILINE it is set, the "start of line" and "end of line"

Signature:

```
MULTILINE
```

```
public constant
```

NEWLINE_ANY

Sets ANY newline sequence as the NEWLINE sequence including

Signature:

```
NEWLINE_ANY
```

```
public constant
```

NEWLINE_ANYCRLF

Sets ANY newline sequence from ASCII.

Signature:

```
NEWLINE_ANYCRLF
```

```
public constant
```

NEWLINE_CR

Sets CR as the NEWLINE sequence.

Signature:

```
NEWLINE_CR
```

```
public constant
```

NEWLINE_CRLF

Sets CRLF as the NEWLINE sequence

Signature:

```
NEWLINE_CRLF
```

```
public constant
```

NEWLINE_LF

Sets LF as the NEWLINE sequence.

Signature:

```
NEWLINE_LF
```

```
public constant
```

NOTBOL

This indicates that beginning of the passed string does **NOT** start

Signature:

```
NOTBOL
```

```
public constant
```

NOTEMPTY

Here matches of empty strings will not be allowed. In C, this is PCRE_NOTEMPTY.

Signature:

```
NOTEMPTY
```

```
public constant
```

NOTEOL

This indicates that end of the passed string does **NOT** end

Signature:

```
NOTEOL
```

```
public constant
```

NO_AUTO_CAPTURE

Disables capturing subpatterns except when the subpatterns are

Signature:

```
NO_AUTO_CAPTURE
```

```
public constant
```

NO_UTF8_CHECK

Turn off checking for the validity of your UTF string. Use this

Signature:

```
NO_UTF8_CHECK
```

```
public constant
```

PARTIAL

This option has no effect on whether a match will occur or not.

Signature:

```
PARTIAL
```

```
public constant
```

STRING_OFFSETS

This is used by matches and all_matches.

Signature:

```
STRING_OFFSETS
```

```
public constant
```

UNGREEDY

This modifier sets the pattern such that quantifiers are

Signature:

```
UNGREEDY
```

```
public constant
```

UTF8

Makes strings passed in to be interpreted as a UTF8 encoded string.

Signature:

```
UTF8
```

```
public constant
```

all_matches

gets the text of all matches.

Signature:

```
all_matches(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : options, defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: Returns **ERROR_NOMATCH** if there are no matches, or a **sequence** of **sequences** of **strings** if there is at least one match. In each member sequence of the returned sequence, the first string is the entire match and subsequent items being each of the captured groups. The size of the sequence is the number of groups in the expression plus one (for the entire match). In other words, each member of the return value will be of the same structure of that is returned by matches.

If `options` contains the bit STRING_OFFSETS, then the result is different. In each member sequence, instead of each member being a string each member is itself a sequence containing the matched text, the starting index in `haystack` and the ending index in `haystack`.

See Also: matches

Example 1:

```
include std/regex.e as re
constant re_name = re:new("([A-Z][a-z]+) ([A-Z][a-z]+)")

object matches = re:all_matches(re_name, "John Doe and Jane Doe")
-- matches is:
-- {
--   {              -- first match
--     "John Doe", -- full match data
--     "John",      -- first group
--     "Doe"        -- second group
--   },
--   {              -- second match
--      "Jane Doe", -- full match data
--      "Jane",     -- first group
--      "Doe"       -- second group
--    }
-- }

matches = re:all_matches(re_name, "John Doe and Jane Doe", , re:STRING_OFFSETS)
-- matches is:
-- {
--    {                       -- first match
--      { "John Doe",  1,  8 }, -- full match data
--      { "John",      1,  4 }, -- first group
--      { "Doe",       6,  8 }  -- second group
--    },
--    {                       -- second match
--      { "Jane Doe", 14, 21 }, -- full match data
--      { "Jane",     14, 17 }, -- first group
--      { "Doe",      19, 21 }  -- second group
--    }
-- }
```

error_message

If new returns an atom, this function will return a text error message

Signature:

```
error_message(object re)
```

```
public function
include regex.e
namespace regex
```

Arguments: ≡ `re`: Regular expression to get the error message from

Returns: An atom (0) when no error message exists, otherwise a sequence describing the error.

Example 1:

```
include std/regex.e
object r = regex:new("[A-Z[a-z]*")
if atom(r) then
  printf(1, "Regex failed to compile: %s\n", { regex:error_message(r) })
end if
```

error_names

Signature:

```
error_names
```

```
public constant
include regex.e
namespace regex
```

error_to_string

converts an regex error number to a string.

Signature:

```
error_to_string(integer i)
```

```
public function
include regex.e
namespace regex
```

See Also: error_message

escape

escapes special regular expression characters that may be entered into a search

Signature:

```
escape(string s)
```

```
public function
include regex.e
namespace regex
```

Arguments: ≡ `s`: string sequence to escape

Returns: An escaped `sequence` representing `s`.

Notes: Special regex characters are: {{{ . \ + * ? [^] $ ( ) { } = ! < >   : - }}}

Example 1:

```
 include std/regex.e as re
 sequence search_s = re:escape("Payroll is $***15.00")
 -- search_s = "Payroll is \\$\\*\\*\\*15\\.00"
```

find

returns the first match of `re` in `haystack`. You can optionally start at the position `from`.

Signature:

```
find(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT, integer size = get_ovector_size(re, 30))
```

```
public function
include regex.e
namespace regex
```

Arguments: ≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : defaults to DEFAULT. See Match Time Option Constants. The only options that may be set when calling find are ANCHORED, NEWLINE_CR, NEWLINE_LF, NEWLINE_CRLF, NEWLINE_ANY NEWLINE_ANYCRLF NOTBOL, NOTEOL, NOTEMPTY, and NO_UTF8_CHECK. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.
≡ `size` : internal (how large an array the C backend should allocate). Defaults to 90, in rare cases this number may need to be increased in order to accomodate complex regex expressions.

Returns: An **object**, which is either an atom of 0, meaning nothing matched or a sequence of index pairs. These index pairs may be fewer than the number of groups specified. These index pairs may be the invalid index pair {0,0}.

The first pair is the starting and ending indeces of the sub-string that matches the expression. This pair may be followed by indeces of the groups. The groups are subexpressions in the regular expression surrounded by parenthesis ().

Now, it is possible to get a match without having all of the groups match. This can happen when there is a quantifier after a group. For example: '([01])*' or '([01])?'. In this case, the returned sequence of pairs will be missing the last group indeces for which there is no match. However, if the missing group is followed by a group that *does* match, {0,0} will be used as a place holder. You can ensure your groups match when your expression matches by keeping quantifiers

Example 1:

```
include std/regex.e as re
r = re:new("([A-Za-z]+) ([0-9]+)") -- John 20 or Jane 45
object result = re:find(r, "John 20")

-- The return value will be:
-- {
--    { 1, 7 }, -- Total match
--    { 1, 4 }, -- First grouping "John" ([A-Za-z]+)
--    { 6, 7 }  -- Second grouping "20" ([0-9]+)
-- }
```

find_all

returns all matches of `re` in `haystack` optionally starting at the sequence position `from`.

Signature:

```
find_all(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT, integer size = get_ovector_size(re, 30))
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : defaults to DEFAULT. See Match Time Option Constants.

Returns: A **sequence** of **sequences** that were returned by find and in the case of no matches this returns an empty **sequence**. Please see find for a detailed description of each member of the return sequence.

Example 1:

```
include std/regex.e as re
constant re_number = re:new("[0-9]+")
object matches = re:find_all(re_number, "10 20 30")

-- matches is:
-- {
--     {{1, 2}},
--     {{4, 5}},
--     {{7, 8}}
-- }
```

find_replace

replaces all matches of a regex with the replacement text.

Signature:

```
find_replace(regex ex, string text, sequence replacement, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex which will be used for matching
≡ `text` : a string on which search and replace will apply

≡ `replacement` : a string, used to replace each of the full matches
≡ `from` : optional start position
≡ `options` : options, defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: A **sequence**, the modified `text`. If there is no match with `re` the return value will be the same as `text` when it was passed in.

Comments: Special replacement operators:

- **`\`** -- Causes the next character to lose its special meaning.
- **`\n`** ~ ~~Inserts a 0x0A (LF) character.~~
- ~~**`\r`** ~~~ Inserts a 0x0D (CR) character.
- **`\t`** -- Inserts a 0x09 (TAB) character.
- **`\1`** to **`\9`** -- Recalls stored substrings from registers (\1, \2, \3, to \9).
- **`\0`** -- Recalls entire matched pattern.
- **`\u`** -- Convert next character to uppercase
- **`\l`** -- Convert next character to lowercase
- **`\U`** -- Convert to uppercase till `\E` or `\e`
- **`\L`** -- Convert to lowercase till `\E` or `\e`
- **`\E`** or **`\e`** -- Terminate a `\\U` or `\L` conversion

Example 1:

```
include std/regex.e
regex r = new(`([A-Za-z]+)\.([A-Za-z]+)`)
sequence details = find_replace(r, "hello.txt",
                                     `Filename: \U\1\e Extension: \U\2\e`)
-- details = "Filename: HELLO Extension: TXT"
```

replaces up to `limit` matches of `ex` in `text` except when `limit` is 0. When `limit` is 0, this routine replaces all of the matches.

find_replace_callback

When `limit` is positive,

Signature:

```
find_replace_callback(regex ex, string text, integer rid,
integer limit = 0, integer from = 1, option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments: ≡ `re` : a regex which will be used for matching
≡ `text` : a string on which search and replace will apply
≡ `rid` : routine id to execute for each match
≡ `limit` : the number of matches to process
≡ `from` : optional start position
≡ `options` : options, defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: A **sequence**, the modified `text`.

Comments: The callback should take one sequence. The first member of this sequence will be a a string representing the entire match and the subsequent members, if they exist, will be a strings for the captured groups within the regular expression.

The function rid. Must take one sequence parameter. The function needs to accept a sequence of strings and return a string. For each match, the function will be passed a sequence of strings. The first string is the entire match the subsequent strings are for the capturing groups. If a match succeeds with groups that don't exist, that place will contain a 0. If the sub-group does exist, the palce will contain the matching group string. for that group.

Example 1:

```
include std/text.e
function my_convert(sequence params)
    switch params[1] do
        case "1" then
            return "one "
        case "2" then
            return "two "
        case else
            return "unknown "
    end switch
end function

regex r = re:new(`\d`)
sequence result = re:find_replace_callback(r, "125",routine_id("my_convert"))
-- result = "one two unknown "


integer missing_data_flag = 0
regex r2 = re:new(`[A-Z][a-z]+ ([A-Z][a-z]+)?`)
function my_toupper( sequence params)
      -- here params[2] may be 0.
      return upper( params[1] )
end function

result = find_replace_callback(r2, "John Doe", routine_id("my_toupper"))
-- params[2] is "Doe"
-- result = "JOHN DOE"
printf(1, "result=%s\n", {result} )
result = find_replace_callback(r2, "Mary", routine_id("my_toupper"))
-- result = "MARY"
```

find_replace_limit

Signature:

```
find_replace_limit(regex ex, string text, sequence replacement,
integer limit, integer from = 1, option_spec options = DEFAULT)


public function
include regex.e
namespace regex
```

get_ovector_size

returns the number of capturing subpatterns (the ovector size) for a regex.

Signature:

```
get_ovector_size(regex ex, integer maxsize = 0)


public function
include regex.e
namespace regex
```

Arguments:
≡ `ex` : a regex
≡ `maxsize` : optional maximum number of named groups to get data from

Returns: An **integer**

has_match

tests if `re` matches any portion of `haystack`.

Signature:

```
has_match(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: An **atom**, 1 if `re` matches any portion of `haystack` or 0 if not.

is_match

tests if the entire `haystack` matches `re`.

Signature:

```
is_match(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: An **atom**, 1 if `re` matches the entire `haystack` or 0 if not.

matches

gets the matched text only.

Signature:

```
matches(regex re, string haystack, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `re` : a regex for a subject to be matched against
≡ `haystack` : a string in which to searched
≡ `from` : an integer setting the starting position to begin searching from. Defaults to 1
≡ `options` : defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or STRING_OFFSETS or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: Returns a **sequence** of strings, the first being the entire match and subsequent items being each of the captured groups or **ERROR_NOMATCH** of there is no match. The size of the sequence is the number of groups in the expression plus one (for the entire match).

If `options` contains the bit STRING_OFFSETS, then the result is different. For each item, a sequence is returned containing the matched text, the starting index in `haystack` and the ending index in `haystack`.

See Also: all_matches

Example 1:

```
include std/regex.e as re
constant re_name = re:new("([A-Z][a-z]+) ([A-Z][a-z]+)")

object matches = re:matches(re_name, "John Doe and Jane Doe")
-- matches is:
-- {
--   "John Doe", -- full match data
--   "John",     -- first group
--   "Doe"       -- second group
-- }

matches = re:matches(re_name, "John Doe and Jane Doe", 1, re:STRING_OFFSETS)
-- matches is:
-- {
--   { "John Doe", 1, 8 }, -- full match data
--   { "John",     1, 4 }, -- first group
--   { "Doe",      6, 8 }  -- second group
-- }
```

new

returns an allocated regular expression.

Signature:

```
new(string pattern, option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments:
≡ `pattern` : a sequence representing a human readable regular expression
≡ `options` : defaults to DEFAULT. See Compile Time Option Constants.

Returns: A **regex**, which other regular expression routines can work on or an atom to indicate an error. If an error, you can call error_message to get a detailed error message.

Comments: This is the only routine that accepts a human readable regular expression. The string is compiled and a regex is returned. Analyzing and compiling a regular expression is a costly operation and should not be done more than necessary. For instance, if your application looks for an email address among text frequently, you should create the regular expression as a constant accessible to your source code and any files that may use it, thus, the regular expression is analyzed and compiled only once per run of your application.

See Also: error_message, find, find_all

Example 1:

```
-- Bad Example
include std/regex.e as re

while sequence(line) do
    re:regex proper_name = re:new("[A-Z][a-z]+ [A-Z][a-z]+")
    if re:find(proper_name, line) then
        -- code
    end if
end while
```

Example 2:

```
-- Good Example
include std/regex.e as re
constant re_proper_name = re:new("[A-Z][a-z]+ [A-Z][a-z]+")
while sequence(line) do
    if re:find(re_proper_name, line) then
        -- code
    end if
end while
```

Example 3:

```
include std/regex.e as re
re:regex number = re:new("[0-9]+")
```

option_spec

Regular expression option specification type

Signature:

```
option_spec(object o)
```

```
public type
include regex.e
namespace regex
```

option_spec_to_string

Converts an option spec to a string.

Signature:

```
option_spec_to_string(option_spec o)
```

```
public function
include regex.e
namespace regex
```

regex

Regular expression type

Signature:

```
regex(object o)
```

```
public type
include regex.e
namespace regex
```

split

splits a string based on a regex as a delimiter.

Signature:

```
split(regex re, string text, integer from = 1,
option_spec options = DEFAULT)
```

```
public function
include regex.e
namespace regex
```

Arguments: ≡ `re` : a regex which will be used for matching
≡ `text` : a string on which search and replace will apply
≡ `from` : optional start position
≡ `options` : options, defaults to DEFAULT. See Match Time Option Constants. `options` can be any match time option or a sequence of valid options or it can be a value that comes from using or_bits on any two valid option values.

Returns: A **sequence** of string values split at the delimiter and if no delimiters were matched this **sequence** will be a one member sequence equal to `{text}`.

Example 1:

```
include std/regex.e as re
regex comma_space_re = re:new(`,\s`)
sequence data = re:split(comma_space_re,
                         "euphoria programming, source code, reference data")
-- data is
-- {
--   "euphoria programming",
--   "source code",
--   "reference data"
-- }
```

split_limit

Signature:

```
split_limit(regex re, string text, integer limit = 0, integer from = 1,
option_spec options = DEFAULT)


public function
include regex.e
namespace regex
```

search

Equality

- compare
- equal

Finding

- find
- find_from
- find_any
- match_any
- find_each
- find_all
- find_all_but
- NESTED_ANY
- NESTED_ALL
- NESTED_INDEX
- NESTED_BACKWARD
- find_nested
- rfind
- find_replace
- match_replace
- binary_search

search API

NESTED_ALL

Signature:

```
NESTED_ALL
```

```
public constant
include search.e
namespace search
```

NESTED_ANY

Signature:

```
NESTED_ANY
```

```
public constant
include search.e
namespace search
```

NESTED_BACKWARD

Signature:

```
NESTED_BACKWARD
```

```
public constant
include search.e
namespace search
```

NESTED_INDEX

Signature:

```
NESTED_INDEX
```

```
public constant
include search.e
namespace search
```

begins

tests whether a sequence is at the head of another one.

Signature:

```
begins(object sub_text, sequence full_text)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `sub_text` : an object to be looked for
≡ `full_text` : a sequence, the head of which is being inspected.

Returns: An **integer**, 1 if `sub_text` begins `full_text`, else 0.

See Also: ends, head

Example 1:

```
s = begins("abc", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

binary_search

locates a "needle" in an ordered "haystack". Starting and ending indices can be specified.

Signature:

```
binary_search(object needle, sequence haystack, integer start_point = 1,
integer end_point = 0)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `needle` : an object to look for
≡ `haystack` : a sequence to search in
≡ `start_point` : an integer, the index at which to start searching. Defaults to 1.
≡ `end_point` : an integer, the end point of the search. Defaults to 0, ie search to end.

Returns: An **integer**, either: # a positive integer `i`, which means `haystack[i]` equals `needle`. # a negative integer, `-i`, with `i` between adjusted start and end points. This means that `needle` is not in the searched slice of `haystack`, but would be at index `i` if it were there. # a negative integer `-i` with `i` out of the searched range. This means than `needle`might be either below the start point if `i` is below the start point, or above the end point if `i` is.

Comments:

• If `end_point` is not greater than zero, it is added to length(`haystack`) once only. Then, the end point of the search is adjusted to length(haystack) if out of bounds.
• The start point is adjusted to 1 if below 1.
• The way this function returns is very similar to what db_find_key does. They use variants of the same algorithm. The latter is all the more efficient as `haystack` is long.
• `haystack` is assumed to be in ascending order. Results are undefined if it is not.
• If duplicate copies of `needle` exist in the range searched on `haystack`, any of the possible contiguous indexes may be returned.

See Also: find, db_find_key

compare

tests two objects and returns the status of first object as being less than, equal, or greater than the first.

Signature:

```
compare(object compared, object reference)
```

```
<built-in> function
```

Arguments: ≡ `compared` : the compared object
≡ `reference` : the reference object

Returns: An **integer**,
• 0 -- if objects are identical
• 1 -- if `compared` is greater than `reference`
• -1 -- if `compared` is less than `reference`

Comments: Atoms are considered to be less than sequences; atoms are compared as ordinary reals. Sequences are compared alphabetically starting with the first element until a difference is found or one of the sequences is exhausted.

See Also: equal, relational operators, operations on sequences, sort

Example 1:

```
x = compare({1,2,{3,{4}},5}, {2-1,1+1,{3,{4}},6-1})
-- identical, x is 0
```

Example 2:

```
if compare("ABC", "ABCD") < 0 then   -- -1
    -- will be true: ABC is "less" because it is shorter
end if
```

Example 3:

```
x = compare('a', "a")
-- x will be -1 because 'a' is an atom
-- while "a" is a sequence
```

ends

tests whether a sequence ends another one.

Signature:

```
ends(object sub_text, sequence full_text)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `sub_text` : an object to be looked for
≡ `full_text` : a sequence, the tail of which is being inspected.

Returns: An **integer**, 1 if `sub_text` ends `full_text`, else 0.

See Also: begins, tail

Example 1:

```
s = ends("def", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

equal

tests two Euphoria objects to see if they are the same.

Signature:

```
equal(object left, object right)
```

```
<built-in> function
```

Arguments: ≡ `left` : one of the objects to test
≡ `right` : the other object

Returns: An **integer**, 1 if the two objects are identical, else 0.

Comments: This is equivalent to the expression: `compare(left, right) = 0`.

This routine, like most other built-in routines, is very fast. It does not have any subroutine call overhead.

See Also: compare

Example 1:

```
if equal(PI, 3.14) then
    puts(1, "give me a better value for PI!\n")
end if
```

Example 2:

```
if equal(name, "George") or equal(name, "GEORGE") then
   puts(1, "name is George\n")
end if
```

find

locates the first occurrence of a "needle," as an *element*, of a "haystack" from the "start" position.

Signature:

```
find(object needle, sequence haystack, integer start)
```

```
<built-in> function
```

Arguments: ≡ `needle` : an object whose presence is being queried
≡ `haystack` : a sequence, which is being looked up for `needle`
≡ `start` : an integer, the position at which to start searching. Defaults to 1.

Returns: An **integer**, 0 if `needle` is not on `haystack`, else the smallest index of an element of `haystack` that equals `needle`.

See Also: find, match, compare

Example 1:

```
location = find(11, {5, 8, 11, 2, 3})
-- location is set to 3
```

Example 2:

```
names = {"fred", "rob", "george", "mary", ""}
location = find("mary", names)
-- location is set to 4
```

find_all

finds all occurrences of an *element* inside a sequence from a specfied starting index.

Signature:

```
find_all(object needle, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `needle` : an object, what to look for
≡ `haystack` : a sequence to search in
≡ `start` : an integer, the starting index position (defaults to 1)

Returns: A **sequence**, the list of all indexes no less than `start` of elements of `haystack` that equal `needle`. This sequence is empty if no match found.

See Also: find, match, match_all

Example 1:

```
 s = find_all('A', "ABCABAB")
 -- s is {1,4,6}
```

find_all_but

locates all non-occurrences of an *element* inside a sequence from a specified starting index.

Signature:

```
find_all_but(object needle, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `needle` : an object, what to look for
≡ `haystack` : a sequence to search in
≡ `start` : an integer, the starting index position (defaults to 1)

Returns: A **sequence**, the list of all indexes no less than `start` of elements of `haystack` that not equal to `needle`. This sequence is empty if `haystack` only consists of `needle`.

See Also: find_all, match, match_all

Example 1:

```
 s = find_all_but('A', "ABCABAB")
 -- s is {2,3,5,7}
```

find_any

locates any *element* (obtained from a list) inside a sequence; returns the first location.

Signature:

```
find_any(object needles, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `needles` : a sequence, the list of items to look for
≡ `haystack` : a sequence, in which "needles" are looked for
≡ `start` : an integer, the starting point of the search. Defaults to 1.

Returns: An **integer**, the smallest index in `haystack` of an element of `needles`, or 0 if no needle is found.

Comments: This function may be applied to a string sequence or a complex sequence.

See Also: find

Example 1:

```
location = find_any("aeiou", "John Smith", 3)
-- location is 8
```

Example 2:

```
location = find_any("aeiou", "John Doe")
-- location is 2
```

find_each

locates all instances of any *element* from the needle sequence that occur in the haystack sequence; returns a list of indices.

Signature:

```
find_each(sequence needles, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needles` : a sequence, the list of items to look for
≡ `haystack` : a sequence, in which "needles" are looked for
≡ `start` : an integer, the starting point of the search. Defaults to 1.

Returns: A **sequence**, the list of indexes into `haystack` that point to an element that is also in `needles`.

Comments: This function may be applied to a string sequence or a complex sequence.

See Also: find, find_any

Example 1:

```
location = find_each("aeiou", "John Smith", 3)
-- location is {8}
```

Example 2:

```
location = find_each("aeiou", "John Doe")
-- location is {2,7,8}
```

find_from

is deprecated since **4.0.0**

Signature:

```
find_from(object needle, object haystack, integer start)
```

```
<built-in> function
```

Comments: In Euphoria 4.0.0 we have the ability to default parameters to procedures and functions. The built-in find therefore now has a `start` parameter that is defaulted to the beginning of the sequence. Thus, find can perform the identical functionality provided by `find_from`. In an undetermined future release of Euphoria, `find_from` will be removed.

See Also: find

find_nested

finds any *element* (among a list) in a sequence of arbitrary shape and nesting level.

Signature:

```
find_nested(object needle, sequence haystack, integer flags = 0,
integer rtn_id = types :NO_ROUTINE_ID)
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needle` : an object, either what to look up, or a list of items to look up
≡ `haystack` : a sequence, where to look up
≡ `flags` : options to the function, see Comments section. Defaults to 0.
≡ `routine` : an integer, the routine_id of an user supplied equal/find function. Defaults to types:NO_ROUTINE_ID.

Returns: A possibly empty **sequence**, of results, one for each hit.

Comments: Each item in the returned sequence is either a sequence of indexes, or a pair {sequence of indexes, index in `needle`}.

See Also: find, rfind, find_any, fetch

Example 1:

```
sequence s = find_nested(3, {5, {4, {3, {2}}}})
-- s is {2 ,2 ,1}
```

Example 2:

```
sequence s = find_nested({3, 2}, {1, 3, {2,3}},
                              NESTED_ANY + NESTED_BACKWARD + NESTED_ALL)
-- s is {{3,2}, {3,1}, {2}}
```

Example 3:

```
sequence s = find_nested({3, 2}, {1, 3, {2,3}},
                               NESTED_ANY + NESTED_INDEXES + NESTED_ALL)
-- s is {{{2}, 1}, {{3, 1}, 2}, {{3, 2}, 1}}
```

find_replace

replaces occurances of `needles`, as *elements*, in the `haystack` with `replacement`; every or just `max` number of needles is replaced.

Signature:

```
find_replace(object needle, sequence haystack, object replacement,
integer max = 0)
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needle` : an object to search and perhaps replace
≡ `haystack` : a sequence to be inspected
≡ `replacement` : an object to substitute for any (first) instance of `needle`
≡ `max` : an integer, 0 to replace all occurrences

Returns: A **sequence**, the modified `haystack`.

Comments: Replacements will not be made recursively on the part of `haystack` that was already changed.

If `max` is 0 or less, any occurrence of `needle` in `haystack` will be replaced by `replacement`. Otherwise, only the first `max` occurrences are.

See Also: find, replace, match_replace

Example 1:

```
s = find_replace('b',"The batty book was all but in Canada.", 'c', 0)
-- s is "The catty cook was all cut in Canada."
```

Example 2:

```
s = find_replace('/', "/euphoria/demo/unix", '\\', 2)
-- s is "\\euphoria\\demo/unix"
```

Example 3:

```
s = find_replace("theater", { "the", "theater", "theif" }, "theatre")
-- s is { "the", "theatre", "theif" }
```

is_in_list

tests if the `item` is in a list of values supplied by `list`.

Signature:

```
is_in_list(object item, sequence list)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `item` : The object to test for.
≡ `list` : A sequence of elements that `item` could be a member of.

Returns: An **integer**, 0 if `item` is not in the `list`, otherwise it returns 1.

Example 1:

```
if is_in_list(user_data, {100, 45, 2, 75, 121}) then
    procA(user_data)
end if
```

is_in_range

tests if the `item` is in a range of values supplied by `range_limits`.

Signature:

```
is_in_range(object item, sequence range_limits, sequence boundries = "[]")
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `item` : The object to test for.
≡ `range_limits` : A sequence of two or more elements. The first is assumed to be the smallest value and the last is assumed to be the highest value.
≡ `boundries`: a sequence. This determines if the range limits are inclusive or not. Must be one of "[]" (the default), "[)", "(]", or "()".

Returns: An **integer**, 0 if `item` is not in the `range_limits` otherwise it returns 1.

Comments:

• In `boundries`, square brackets mean *inclusive* and round brackets mean *exclusive*. Thus "[]" includes both limits in the range, while "()" excludes both limits. And, "[)" includes the lower limit and excludes the upper limits while "(]" does the reverse.

Example 1:

```
 if is_in_range(2, {2, 75}) then
     procA(user_data) -- Gets run (both limits included)
 end if
 if is_in_range(2, {2, 75}, "(]") then
     procA(user_data) -- Does not get run
 end if
```

lookup

returns the corresponding element from the target list, if the supplied item is in the source list.

Signature:

```
lookup(object find_item, sequence source_list, sequence target_list,
object def_value = 0)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `find_item`: an object that might exist in `source_list`.
≡ `source_list`: a sequence that might contain `pITem`.
≡ `target_list`: a sequence from which the corresponding item will be returned.
≡ `def_value`: an object (defaults to zero). This is returned when `find_item` is not in `source_list` **and** `target_list` is not longer than `source_list`.

Returns: an object
• If `find_item` is found in `source_list` then this is the corresponding element from `target_list`
• If `find_item` is not in `source_list` then if `target_list` is longer than `source_list` then the last item in `target_list` is returned otherwise `def_value` is returned.

Example 1:

```
 lookup('a', "cat", "dog") --> 'o'
 lookup('d', "cat", "dogx") --> 'x'
 lookup('d', "cat", "dog") --> 0
 lookup('d', "cat", "dog", -1) --> -1
 lookup("ant",{"ant","bear","cat"}, {"spider","seal","dog","unknown"})
             --> "spider"
 lookup("dog",{"ant","bear","cat"}, {"spider","seal","dog","unknown"})
             --> "unknown"
```

match

locates a "needle," as a *slice*, of a "haystack" beginning from the "start" index.

Signature:

```
match(sequence needle, sequence haystack, integer start)
```

```
<built-in> function
```

Arguments: ≡ `needle` : a sequence whose presence as a "substring" is being queried
≡ `haystack` : a sequence, which is being looked up for `needle` as a sub-sequence
≡ `start` : an integer, the point from which matching is attempted. Defaults to 1.

Returns: An **integer**, 0 if no slice of `haystack` is `needle`, else the smallest index at which such a slice starts.

See Also: find, compare, wildcard:is_match

Example 1:

```
 location = match("pho", "Euphoria")
 -- location is set to 3
```

match_all

locates needles, as *slices*, in a haystack.

Signature:

```
match_all(sequence needle, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needle` : a sequence, what to look for
≡ `haystack` : a sequence to search in
≡ `start` : an integer, the starting index position (defaults to 1)

Returns: A **sequence**, of integers, the list of all lower indexes, not less than `start`, of all slices in `haystack` that equal `needle`. The list may be empty.

See Also: match, regex:find_all find, find_all

Example 1:

```
 s = match_all("the", "the dog chased the cat under the table.")
 -- s is {1,16,30}
```

match_any

tests if any *element* from `needles` is in `haystack`.

Signature:

```
match_any(sequence needles, sequence haystack, integer start = 1)
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needles` : a sequence, the list of items to look for
≡ `haystack` : a sequence, in which "needles" are looked for
≡ `start` : an integer, the starting point of the search. Defaults to 1.

Returns: An **integer**, 0 if no matches, 1 if any matches.

Comments: This function may be applied to a string sequence or a complex sequence.

See Also: find_any

Example 1:

```
 ok = match_any("aeiou", "John Smith")
 -- okay is 1
 ok = match_any("xyz", "John Smith" )
 -- okay is 0
```

match_from

is deprecated since **4.0.0**

Signature:

```
match_from(sequence needle, sequence haystack, integer start)
```

```
<built-in> function
```

Comments: In Euphoria 4.0.0 we have the ability to default parameters to procedures and functions. The built-in match therefore now has a `start` parameter that is defaulted to the beginning of the sequence. Thus, match can perform the identical functionality provided by `match_from`. In an undetermined future release of Euphoria, `match_from` will be removed.

See Also: match

match_replace

locates a "needle," as a *slice*, in a "haystack", and replaces any or only the first few occurrences with a replacement.

Signature:

```
match_replace(object needle, sequence haystack, object replacement,
integer max = 0)


public function
include search.e
namespace search
```

Arguments:
≡ `needle` : an object to search and perhaps replace
≡ `haystack` : a sequence to be inspected
≡ `replacement` : an object to substitute for any (first) instance of `needle`
≡ `max` : an integer, 0 to replace all occurrences

Returns: A **sequence**, the modified `haystack`.

Comments: Replacements will not be made recursively on the part of `haystack` that was already changed.

If `max` is 0 or less, any occurrence of `needle` in `haystack` will be replaced by `replacement`. Otherwise, only the first `max` occurrences are.

If either `needle` or `replacement` are atoms they will be treated as if you had passed in a length-1 sequence containing the said atom.

See Also: find, replace, regex:find_replace, find_replace

Example 1:

```
s=match_replace("the","the cat ate the food under the table","THE",0)
-- s is "THE cat ate THE food under THE table"
```

Example 2:

```
s=match_replace("the","the cat ate the food under the table","THE",2)
-- s is "THE cat ate THE food under the table"
```

Example 3:

```
s = match_replace('/', "/euphoria/demo/unix", '\\', 2)
-- s is "\\euphoria\\demo/unix"
```

Example 4:

```
s = match_replace('a', "abracadabra", 'X')
-- s is now "XbrXcXdXbrX"
s = match_replace("ra", "abracadabra", 'X')
-- s is now "abXcadabX"
s = match_replace("a", "abracadabra", "aa")
-- s is now "aabraacaadaabraa"
s = match_replace("a", "abracadabra", "")
-- s is now "brcdbr"
```

rfind

locates a needle, as an *element*, in a haystack but in reverse order.

Signature:

```
rfind(object needle, sequence haystack, integer start = length(haystack))
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needle` : an object to search for
≡ `haystack` : a sequence to search in
≡ `start` : an integer, the starting index position (defaults to length(`haystack`))

Returns: An **integer**, 0 if no instance of `needle` can be found on `haystack` before index `start`, or the highest such index otherwise.

Comments: If `start` is less than 1, it will be added once to length(`haystack`) to designate a position counted backwards. Thus, if `start` is -1, the first element to be queried in `haystack` will be `haystack`[$-1], then `haystack`[$-2] and so on.

See Also: find, rmatch

Example 1:

```
location = rfind(11, {5, 8, 11, 2, 11, 3})
-- location is set to 5
```

Example 2:

```
names = {"fred", "rob", "rob", "george", "mary"}
location = rfind("rob", names)
-- location is set to 3
location = rfind("rob", names, -4)
-- location is set to 2
```

rmatch

locates a needle, as a *slice*, in a haystack but in reverse order.

Signature:

```
rmatch(sequence needle, sequence haystack,
integer start = length(haystack))
```

```
public function
include search.e
namespace search
```

Arguments:
≡ `needle` : a sequence to search for
≡ `haystack` : a sequence to search in
≡ `start` : an integer, the starting index position (defaults to length(`haystack`))

Returns: An **integer**, either 0 if no slice of `haystack` starting before `start` equals `needle`, else the highest lower index of such a slice.

Comments: If `start` is less than 1, it will be added once to length(`haystack`) to designate a position counted backwards. Thus, if `start` is -1, the first element to be queried in `haystack` will be `haystack`[$-1], then `haystack`[$-2] and so on.

See Also: rfind, match

Example 1:

```
location = rmatch("the", "the dog ate the steak from the table.")
-- location is set to 28 (3rd 'the')
location = rmatch("the","the dog ate the steak from the table.",-11)
-- location is set to 13 (2nd 'the')
```

vlookup

this returns the corresponding element from the target column, if the supplied item is in a source grid column.

Signature:

```
vlookup(object find_item, sequence grid_data, integer source_col,
integer target_col, object def_value = 0)
```

```
public function
include search.e
namespace search
```

Arguments: ≡ `find_item`: an object that might exist in `source_col`.
≡ `grid_data`: a 2D grid sequence that might contain `pITem`.
≡ `source_col`: an integer. The column number to look for `find_item`.
≡ `target_col`: an integer. The column number from which the corresponding item will be returned.
≡ `def_value`: an object (defaults to zero). This is returned when `find_item` is not found in the `source_col` column, or if found but the target column does not exist.

Returns: an object
• If `find_item` is found in the `source_col` column then this is the corresponding element from the `target_col` column.

Comments:

• If a row in the grid is actually a single atom, the row is ignored.
• If a row's length is less than the `source_col`, the row is ignored.

Example 1:

```
sequence grid
grid = {
       {"ant", "spider", "mortein"},
       {"bear", "seal", "gun"},
       {"cat", "dog", "ranger"},
       $
 }
vlookup("ant", grid, 1, 2, "?") --> "spider"
vlookup("ant", grid, 1, 3, "?") --> "mortein"
vlookup("seal", grid, 2, 3, "?") --> "gun"
vlookup("seal", grid, 2, 1, "?") --> "bear"
vlookup("mouse", grid, 2, 3, "?") --> "?"
```

sequence

Constants

ADD_PREPEND
ADD_APPEND
ADD_SORT_UP
ADD_SORT_DOWN
ROTATE_LEFT
ROTATE_RIGHT

Basic routines

binop_ok
fetch
store
valid_index
rotate

sequence API

ADD_APPEND

Signature:

```
ADD_APPEND
```

```
public enum
include sequence.e
namespace stdseq
```

ADD_PREPEND

Signature:

```
ADD_PREPEND
```

```
public enum
include sequence.e
namespace stdseq
```

ADD_SORT_DOWN

Signature:

```
ADD_SORT_DOWN
```

```
public enum
include sequence.e
namespace stdseq
```

ADD_SORT_UP

Signature:

```
ADD_SORT_UP
```

```
public enum
include sequence.e
namespace stdseq
```

BK_LEN

Indicates that `size` parameter is maximum length of sub-sequence. See breakup

Signature:

```
BK_LEN
```

```
public enum
include sequence.e
namespace stdseq
```

BK_PIECES

Indicates that `size` parameter is maximum number of sub-sequence. See breakup

Signature:

```
BK_PIECES
```

```
public enum
include sequence.e
namespace stdseq
```

COMBINE_SORTED

Signature:

```
COMBINE_SORTED
```

```
public enum
include sequence.e
namespace stdseq
```

COMBINE_UNSORTED

Signature:

```
COMBINE_UNSORTED
```

```
public enum
include sequence.e
namespace stdseq
```

RD_INPLACE

removes items while preserving the original order of the unique items.

Signature:

```
RD_INPLACE
```

```
public enum
include sequence.e
namespace stdseq
```

See Also: remove_dups

RD_PRESORTED

Assumes that the elements in `source_data` are already sorted. If they

Signature:

```
RD_PRESORTED
```

```
public enum
include sequence.e
namespace stdseq
```

See Also: remove_dups

RD_SORT

returns the unique elements in ascending sorted order.

Signature:

```
RD_SORT
```

```
public enum
include sequence.e
namespace stdseq
```

See Also: remove_dups

ROTATE_LEFT

Signature:

```
ROTATE_LEFT
```

```
public constant
include sequence.e
namespace stdseq
```

ROTATE_RIGHT

Signature:

```
ROTATE_RIGHT
```

```
public constant
include sequence.e
namespace stdseq
```

SEQ_NOALT

Indicates that remove_subseq() must not replace removed sub-sequences

Signature:

```
SEQ_NOALT
```

```
public constant
include sequence.e
namespace stdseq
```

STDFLTR_ALPHA

this is a predefined routine_id for use with filter().

Signature:

```
STDFLTR_ALPHA
```

```
public constant
```

Comments: Used to filter out non-alphabetic characters from a string.

Example 1:

```
 -- Collect only the alphabetic characters from 'text'
  result = filter(text, STDFLTR_ALPHA)
```

add_item

adds an item to the sequence if its not already there. If it already exists

Signature:

```
add_item(object needle, sequence haystack, integer pOrder = 1)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `needle` : object to add.
≡ `haystack` : sequence to add it to.
≡ `order` : an integer; determines how the `needle` affects the `haystack`. It can be added to the front (prepended), to the back (appended), or sorted after adding. The default is to prepend it.

Returns: A **sequence**, which is `haystack` with `needle` added to it.

Comments: An error occurs if an invalid `order` argument is supplied.

The following enum is provided for specifying `order`:

- `ADD_PREPEND` -- prepend `needle` to `haystack`. This is the default option.
- `ADD_APPEND` -- append `needle` to `haystack`.
- `ADD_SORT_UP` -- sort `haystack` in ascending order after inserting `needle`
- `ADD_SORT_DOWN` -- sort `haystack` in descending order after inserting `needle`

Example 1:

```
 s = add_item( 1, {3,4,2}, ADD_PREPEND ) -- prepend
 -- s is {1,3,4,2}
```

Example 2:

```
 s = add_item( 1, {3,4,2}, ADD_APPEND ) -- append
 -- s is {3,4,2,1}
```

Example 3:

```
 s = add_item( 1, {3,4,2}, ADD_SORT_UP ) -- ascending
 -- s is {1,2,3,4}
```

Example 4:

```
 s = add_item( 1, {3,4,2}, ADD_SORT_DOWN ) -- descending
 -- s is {4,3,2,1}
```

Example 5:

```
 s = add_item( 1, {3,1,4,2} )
 -- s is {3,1,4,2} -- Item was already in list so no change.
```

append

adds an object as the last element of a sequence.

Signature:

```
append(sequence target, object x)
```

```
<built-in> function
```

Arguments: ≡ `source` : the sequence to add to
≡ `x` : the object to add

Returns: A **sequence**, whose first elements are those of `target` and whose last element is `x`.

Comments: The length of the resulting sequence will be `length(target) + 1`, no matter what `x` is.

If `x` is an atom this is equivalent to `result = target & x`. If `x` is a sequence it is not equivalent.

The extra storage is allocated automatically and very efficiently with Euphoria's dynamic storage allocation. The case where the `target` is appended to itself is highly optimized (as in Example 1 below).

See Also: prepend, &

Example 1:

```
sequence x

  x = {}
  for i = 1 to 10 do
     x = append(x, i)
  end for
  -- x is now {1,2,3,4,5,6,7,8,9,10}
```

Example 2:

```
sequence x, y, z

x = {"fred", "barney"}
y = append(x, "wilma")
-- y is now {"fred", "barney", "wilma"}

z = append(append(y, "betty"), {"bam", "bam"})
-- z is now {"fred", "barney", "wilma", "betty", {"bam", "bam"}}
```

apply

applies a function to every element of a sequence, returning a new sequence of

Signature:

```
apply(sequence source, integer rid, object userdata = {})
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:

- `source` : the sequence to map
- `rid` : the routine_id of function to use as converter
- `userdata` : an object passed to each invocation of `rid`. If omitted, {} is used.

Returns: A **sequence**, the length of `source`. Each element there is the corresponding element in `source` mapped using the routine referred to by `rid`.

Comments: The supplied routine must take two arguments. The type of the first argment must be compatible with all the elements in `source`. The second argument is an `object` containing `userdata`.

See Also: filter

Example 1:

```
function greeter(object o, object d)
    return o[1] & ", " & o[2] & d
end function

s = apply({{"Hello", "John"}, {"Goodbye", "John"}},routine_id("greeter"),"!")
-- s is {"Hello, John!", "Goodbye, John!"}
```

binop_ok

checks whether a operation between two objects is possible.

Signature:

```
binop_ok(object a, object b)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `a` : first object
≡ `b` : second object

Returns: An **integer**, 1 if an operation is possible between `a` and `b`, or else 0.

Comments: An operation between two objects is possible when the have compatible shapes.

See Also: series

Example 1:

```
i = binop_ok({1,2,3},{4,5})
--> i is 0
-- lengths are not compatible

i = binop_ok({1,2,3},4)
--> i is 1
-- atom to sequence operations are possible

i = binop_ok({1,2,3},{4,{5,6},7})
--> i is 1
```

breakup

breaks up a sequence into multiple sequences of a given length.

Signature:

```
breakup(sequence source, object size, integer style = BK_LEN)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `source` : the sequence to be broken up into sub-sequences.
≡ `size` : an object, if an integer it is either the maximum length of each resulting sub-sequence or the maximum number of sub-sequences to break `source` into.
If `size` is a sequence, it is a list of element counts for the sub-sequences it creates.
≡ `style` : an integer, Either BK_LEN if `size` integer represents the sub-sequences' maximum length, or BK_PIECES if the `size` integer represents the maximum number of sub-sequences (pieces) to break `source` into.

Returns: A **sequence**, of sequences.

Comments:

...... ♦ When `size` is an integer and `style` is BK_LEN then: ...
The sub-sequences have length `size`, except possibly the last one, which may be shorter. For example if `source` has 11 items and `size` is 3, then the first three sub-sequences will get 3 items each and the remaining 2 items will go into the last sub-sequence. If `size` is less than 1 or greater than the length of the `source`, the `source` is returned as the only sub-sequence.

When `size` is an integer and `style` is BK_PIECES...
There is exactly `size` sub-sequences created. If the `source` is not evenly divisible into that many pieces, then the lefthand sub-sequences will contain one more element than the right-hand sub-sequences. For example, if source contains 10 items and we break it into 3 pieces, piece #1 gets 4 elements, piece #2 gets 3 items and piece #3 gets 3 items - a total of 10. If source had 11 elements then the pieces will have 4,4, and 3 respectively.

When `size` is a sequence...**
The style parameter is ignored in this case. The source will be broken up according to the counts contained in the size parameter. For example, if `size` was {3,4,0,1} then piece #1 gets 3 items, #2 gets 4 items, #3 gets 0 items, and #4 gets 1 item. Note that if not all items from source are placed into the sub-sequences defined by `size`, and *extra* sub-sequence is appended that contains the remaining items from `source`.

In all cases, when concatenated these sub-sequences will be identical to the original `source`.

See Also: split flatten

Example 1:

```
s = breakup("5545112133234454", 4)
-- s is {"5545", "1121", "3323", "4454"}
```

Example 2:

```
s = breakup("12345", 2)
-- s is {"12", "34", "5"}
```

Example 3:

```
s = breakup({1,2,3,4,5,6}, 3)
-- s is {{1,2,3}, {4,5,6}}
```

Example 4:

```
s = breakup("ABCDEF", 0)
-- s is {"ABCDEF"}
```

build_list

implements "List Comprehension" or building a list based on the contents of another list.

Signature:

```
build_list(sequence source, object transformer, integer singleton = 1,
object user_data = {})
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : A sequence. The list of items to base the new list upon.

≡ `transformer` : One or more routine_ids. These are routine_id of functions that must receive three parameters (object x, sequence i, object u) where 'x' is an item in the `source` list, 'i' contains the position that 'x' is found in the `source` list and the length of `source`, and 'u' is the `user_data` value. Each transformer must return a two-element sequence. If the first element is zero, then build_list() continues on with the next transformer function for the same 'x'. If the first element is not zero, the second element is added to the new list being built (other elements are ignored) and build_list skips the rest of the transformers and processes the next element in `source`.
≡ `singleton` : An integer. If zero then the transformer functions return multiple list elements. If not zero then the transformer functions return a single item (which might be a sequence).
≡ `user_data` : Any object. This is passed unchanged to each transformer function.

Returns: A **sequence**, The new list of items.

Comments:

• If the transformer is -1, then the source item is just copied.

Example 1:

```
function remitem(object x, sequence i, object q)
 if (x < q) then
  return {0} -- no output
 else
  return {1,x} -- copy 'x'
 end if
end function

sequence s
-- Remove negative elements (x < 0)
s = build_list({-3, 0, 1.1, -2, 2, 3, -1.5}, routine_id("remitem"), , 0)
-- s is {0, 1.1, 2, 3}
```

columnize

converts a set of sub sequences into a set of "columns."

Signature:

```
columnize(sequence source, object cols = {}, object defval = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : sequence containing the sub-sequences
≡ `cols` : either a specific column number or a set of column numbers. Default is 0, which returns the maximum number of columns.
≡ `defval` : an object. Used when a column value is not available. Default is 0

Comments: Any atoms found in `source` are treated as if they are a one-element sequence.

Example 1:

```
s = columnize({{1, 2}, {3, 4}, {5, 6}})
-- s is { {1,3,5}, {2,4,6}}
```

Example 2:

```
s = columnize({{1, 2}, {3, 4}, {5, 6, 7}})
-- s is { {1,3,5}, {2,4,6}, {0,0,7} }
s = columnize({{1, 2}, {3, 4}, {5, 6, 7}},,-999})
    --> Change the not-available value.
-- s is { {1,3,5}, {2,4,6}, {-999,-999,7} }
```

Example 3:

```
s = columnize({{1, 2}, {3, 4}, {5, 6, 7}}, 2)
```

```
 -- s is { {2,4,6} } -- Column 2 only
```

Example 4:

```
 s = columnize({{1, 2}, {3, 4}, {5, 6, 7}}, {2,1})
 -- s is { {2,4,6}, {1,3,5} } -- Column 2 then column 1
```

Example 5:

```
 s = columnize({"abc", "def", "ghi"})
 -- s is {"adg", "beh", "cfi" }
```

combine

combines all the sub-sequences into a single (optionally sorted) list.

Signature:

```
combine(sequence source_data, integer proc_option = COMBINE_SORTED)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source_data` : A sequence that contains sub-sequences to be combined.
≡ `proc_option` : An integer; COMBINE_UNSORTED to return a non-sorted list and COMBINE_SORTED (the default) to return a sorted list.

Returns: A **sequence**, that contains all the elements from all the first-level of sub-sequences from `source_data`.

Comments: The elements in the sub-sequences do not have to be pre-sorted.

Only one level of sub-sequence is combined.

Example 1:

```
 sequence s = { {4,7,9}, {7,2,5,9}, {0,4}, {5}, {6,5}}
 combine(s, COMBINE_SORTED)   --> {0,2,4,4,5,5,5,6,7,7,9,9}
 combine(s, COMBINE_UNSORTED) --> {4,7,9,7,2,5,9,0,4,5,6,5}
```

Example 2:

```
 sequence s = { {"cat", "dog"}, {"fish", "whale"}, {"wolf"}, {"snail", "worm"}}
 combine(s)                   --> {"cat","dog","fish","snail","whale","wolf","worm"}
 combine(s, COMBINE_UNSORTED) --> {"cat","dog","fish","whale","wolf","snail","worm"}
```

Example 3:

```
 sequence s = { "cat", "dog","fish", "whale", "wolf", "snail", "worm"}
 combine(s)                   --> "aaacdeffghhiilllmnooorsstwww"
 combine(s, COMBINE_UNSORTED) --> "catdogfishwhalewolfsnailworm"
```

extract

Picks out from a sequence a set of elements according to the supplied set of indexes.

Signature:

```
extract(sequence source, sequence indexes)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence from which to extract elements

≡ `indexes` : a sequence of atoms, the indexes of the elements to be fetched in `source`.

Returns: A **sequence**, of the same length as `indexes`.

See Also: slice

Example 1:

```
s = extract({11,13,15,17},{3,1,2,1,4})
-- s is {15,11,13,11,17}
```

fetch

retrieves an element nested arbitrarily deeply within a sequence.

Signature:

```
fetch(sequence source, sequence indexes)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence from which to fetch
≡ `indexes` : a sequence of integers, the path to follow to reach the element to return.

Returns: An **object**, which is `source[indexes[1]][indexes[2]]...[indexes[$]]`

Comments: The last element of `indexes` may be a pair {lower,upper}, in which case a slice of the innermost referenced sequence is returned.

See Also: store, Subscripting of Sequences

Example 1:

```
x = fetch({0,1,2,3,{"abc","def","ghi"},6},{5,2,3})
-- x is 'f', or 102.
```

filter

filters a sequence based on a user supplied comparator function.

Signature:

```
filter(sequence source, object rid, object userdata = {},
object rangetype = "")
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:

• `source` : sequence to filter
• `rid` : Either a routine_id of function to use as comparator or one of the predefined compаritors.
• `userdata` : an object passed to each invocation of `rid`. If omitted, {} is used.
• `rangetype`: A sequence. Only used when `rid` is "in" or "out". This is used to let the function know how to interpret `userdata`. When `rangetype` is an empty string (which is the default), then `userdata` is treated as a set of zero or more discrete items such that "in" will only return items from `source` that are in the set of item in `userdata` and "out" returns those not in `userdata`. The other values for `rangetype` mean that `userdata` must be a set of exactly two items, that represent the lower and upper limits of a range of values.

Returns: A **sequence**, made of the elements in `source` which passed the comparitor test.

Comments:

• The only items from `source` that are returned are those that pass the test.
• When `rid` is a routine_id, that user defined routine must be a function. Each item in `source`, along with the `userdata` is passed to the function. The function must return a non-zero atom if the item is to be included in the result sequence, otherwise it should return zero to exclude it from the result.
• The predefined comparitors are:

See Also: apply

Example 1:

```
function mask_nums(atom a, object t)
    if sequence(t) then
        return 0
    end if
    return and_bits(a, t) != 0
end function

function even_nums(atom a, atom t)
    return and_bits(a,1) = 0
end function

constant data = {5,8,20,19,3,2,10}
filter(data, routine_id("mask_nums"), 1) --> {5,19,3}
filter(data, routine_id("mask_nums"), 2) -->{19, 3, 2, 10}
filter(data, routine_id("even_nums")) -->{8, 20, 2, 10}

-- Using 'in' and 'out' with sets.
filter(data, "in", {3,4,5,6,7,8}) -->{5,8,3}
filter(data, "out", {3,4,5,6,7,8}) -->{20,19,2,10}

-- Using 'in' and 'out' with ranges.
filter(data, "in",  {3,8}, "[]") --> {5,8,3}
filter(data, "in",  {3,8}, "[)") --> {5,3}
filter(data, "in",  {3,8}, "(]") --> {5,8}
filter(data, "in",  {3,8}, "()") --> {5}
filter(data, "out", {3,8}, "[]") --> {20,19,2,10}
filter(data, "out", {3,8}, "[)") --> {8,20,19,2,10}
filter(data, "out", {3,8}, "(]") --> {20,19,3,2,10}
filter(data, "out", {3,8}, "()") --> {8,20,19,3,2,10}
```

Example 3:

```
function quiksort(sequence s)
 if length(s) < 2 then
  return s
 end if
 return  quiksort( filter(s[2..$], "<=", s[1]) ) &
         s[1] &
         quiksort(filter(s[2..$], ">", s[1]))
end function
? quiksort( {5,4,7,2,4,9,1,0,4,32,7,54,2,5,8,445,67} )
--> {0,1,2,2,4,4,4,5,5,7,7,8,9,32,54,67,445}
```

flatten

removes all nesting from a sequence.

Signature:

```
flatten(sequence s, object delim = "")
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `s` : the sequence to flatten out.
≡ `delim` : An optional delimiter to place after each flattened sub-sequence (except the last one).

Returns: A **sequence**, of atoms, all the atoms in `s` enumerated.

Comments:

- If you consider a sequence as a tree, then the enumeration is performed by left-right reading of the tree. The elements are simply read left to right, without any care for braces.
- Empty sub-sequences are stripped out entirely.

Example 1:

```
s = flatten({{18, 19}, 45, {18.4, 29.3}})
-- s is {18, 19, 45, 18.4, 29.3}
```

Example 2:

```
s = flatten({18,{ 19, {45}}, {18.4, {}, 29.3}})
-- s is {18, 19, 45, 18.4, 29.3}
```

Example 3:

```
Using the delimiter argument
s = flatten({"abc", "def", "ghi"}, ", ")
-- s is "abc, def, ghi"
```

head

Return the first `size` item(s) of a sequence.

Signature:

```
head(sequence source, atom size=1)
```

```
<built-in> function
```

Arguments:

≡ `source` : the sequence from which elements will be returned
≡ `size` : an integer; how many elements, at most, will be returned. Defaults to 1.

Returns: A **sequence**, `source` if its length is not greater than `size`, or the `size` first elements of `source` otherwise.

See Also: tail, mid, slice

Example 1:

```
s2 = head("John Doe", 4)
-- s2 is John
```

Example 2:

```
s2 = head("John Doe", 50)
-- s2 is John Doe
```

Example 3:

```
s2 = head({1, 5.4, "John", 30}, 3)
-- s2 is {1, 5.4, "John"}
```

insert

inserts an object into a sequence as a new element at a given location.

Signature:

```
insert(sequence target, object what, integer index)
```

```
<built-in> function
```

Arguments:

≡ `target` : the sequence to insert into
≡ `what` : the object to insert

≡ `index` : an integer, the position in `target` where `what` should appear

Returns: A **sequence**, which is `target` with one more element at `index`, which is `what`.

Comments: `target` can be a sequence of any shape, and `what` any kind of object.

The length of the returned sequence is always `length(target) + 1`.

Inserting a sequence into a string returns a sequence which is no longer a string.

See Also: remove, splice, append, prepend

Example 1:

```
s = insert("John Doe", " Middle", 5)
-- s is {'J','o','h','n'," Middle",' ','D','o','e'}
```

Example 2:

```
s = insert({10,30,40}, 20, 2)
-- s is {10,20,30,40}
```

join

joins sequences together using a delimiter.

Signature:

```
join(sequence items, object delim = " ")
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `items` : the sequence of items to join.
≡ `delim` : an object, the delimiter to join by. Defaults to " ".

Comments: This function may be applied to a string sequence or a complex sequence

See Also: split, split_any, breakup

Example 1:

```
result = join({"John", "Middle", "Doe"})
-- result is "John Middle Doe"
```

Example 2:

```
result = join({"John", "Middle", "Doe"}, ",")
-- result is "John,Middle,Doe"
```

length

returns the length of an object.

Signature:

```
length(object target)
```

```
<built-in> function
```

Arguments: ≡ `target` : the object being queried

Returns: An **integer**, the number of elements involved with `target`.

Comments:

- An atom always has a length of one.
- The length of a sequence is the number of elements in the sequence; nested sequences still count as a single element.

• The length of each sequence is stored internally by the interpreter for fast access. In some other languages this operation requires a search through memory for an end marker.

See Also: append, prepend, &

Example 1:

```
 length({{1,2}, {3,4}, {5,6}})   -- 3
 length("")  -- 0
 length({})  -- 0
 length( 7 )   -- 1
 length( 3.14 ) -- 1
```

mapping

changes each item from `source_arg` found in `from_set` into the

Signature:

```
mapping(object source_arg, sequence from_set, sequence to_set,
integer one_level = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `source_arg` : Any Euphoria object to be transformed.
≡ `from_set` : A sequence of objects representing the only items from `source_arg` that are actually transformed.
≡ `to_set` : A sequence of objects representing the transformed equivalents of those found in `from_set`.
≡ `one_level` : An integer. 0 (the default) means that mapping applies to every atom in every level of sub-sequences. 1 means that mapping only applies to the items at the first level in `source_arg`.

Returns: An **object**, The transformed version of `source_arg`.

Comments:

• When `one_level` is zero or omitted, for each item in `source_arg`,
…… ♦ if it is an atom then it may be transformed
…… ♦ if it is a sequence, then the mapping is performed recursively on the sequence.
…… ♦ This option required `from_set` to only contain atoms and contain no sub-sequences.
• When `one_level` is not zero, for each item in `source_arg`,
…… ♦ regardless of whether it is an atom or sequence, if it is found in `from_set` then it is mapped to the corresponding object in `to_set`..
• Mapping occurs when an item in `source_arg` is found in `from_set`, then it is replaced by the corresponding object in `to_set`.

Example 1:

```
 res = mapping("The Cat in the Hat", "aeiou", "AEIOU")
 -- res is now "ThE CAt In thE HAt"
```

mid

returns a slice of a sequence, given by a starting point and a length.

Signature:

```
mid(sequence source, atom start, atom len)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence some elements of which will be returned
≡ `start` : an integer, the lower index of the slice to return
≡ `len` : an integer, the length of the slice to return

Returns: A **sequence**, made of at most `len` elements of `source`. These elements are at contiguous positions in `source` starting at `start`.

Comments: `len` may be negative, in which case it is added `length(source)` once.

See Also: head, tail, slice

Example 1:

```
s2 = mid("John Middle Doe", 6, 6)
-- s2 is Middle
```

Example 2:

```
s2 = mid("John Middle Doe", 6, 50)
-- s2 is Middle Doe
```

Example 3:

```
s2 = mid({1, 5.4, "John", 30}, 2, 2)
-- s2 is {5.4, "John"}
```

Example 4:

```
s2 = mid({1, 5.4, "John", 30}, 2, -1)
-- s2 is {5.4, "John", 30}
```

minsize

ensures that the returned sequence is at least the minimum specified length.

Signature:

```
minsize(object source_data,
integer min_size = floor(length(source_data)* 1.5), object new_data = 0)


public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source_data` : An object that might need extending.
≡ `min_size`: An integer. The minimum length that `source_data` must be. The default is to increase the length of `source_data# by 50%`.
≡ new_data`: An object. This used to when` source_data `needs to be extended, in which case it is appended as many times as required to make the length equal to` min_size##. The default is 0.

Returns: A **sequence**. The padded sequence, unchanged if its size was not less than `min_size` on input.

Example 1:

```
sequence s
s = minsize({4,3,6,2,7,1,2}, 10, -1) --> {4,3,6,2,7,1,2,-1,-1,-1}
s = minsize({4,3,6,2,7,1,2},  5, -1) --> {4,3,6,2,7,1,2}
```

pad_head

pads the beginning of a sequence with an object so as to meet a minimum

Signature:

```
pad_head(object target, integer size, object ch = ' ')
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `target` : the sequence to pad.
≡ `size` : an integer, the target minimum size for `target`
≡ `padding` : an object, usually the character to pad to (defaults to ' ').

Returns: A **sequence**, either `target` if it was long enough, or a sequence of length `size` whose last elements are those of `target` and whose first few head elements all equal `padding`.

Comments: `pad_head`() will not remove characters. If `length(target)` is greater than `size`, this function simply returns `target`. See head() if you wish to truncate long sequences.

See Also: trim_head, pad_tail, head

Example 1:

```
s = pad_head("ABC", 6)
-- s is "   ABC"

s = pad_head("ABC", 6, '-')
-- s is "---ABC"
```

pad_tail

pads the end of a sequence with an object so as to meet a minimum length condition.

Signature:

```
pad_tail(object target, integer size, object ch = ' ')


public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `target` : the sequence to pad.
≡ `size` : an integer, the target minimum size for `target`
≡ `padding` : an object, usually the character to pad to (defaults to ' ').

Returns: A **sequence**, either `target` if it was long enough, or a sequence of length `size` whose first elements are those of `target` and whose last few head elements all equal `padding`.

Comments: `pad_tail`() will not remove characters. If `length(target)` is greater than `size`, this function simply returns `target`. See tail() if you wish to truncate long sequences.

See Also: trim_tail, pad_head, tail

Example 1:

```
s = pad_tail("ABC", 6)
-- s is "ABC   "

s = pad_tail("ABC", 6, '-')
-- s is "ABC---"
```

patch

changes a sequence slice, possibly with padding

Signature:

```
patch(sequence target, sequence source, integer start,
object filler = ' ')
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `target` : a sequence, a modified copy of which will be returned
≡ `source` : a sequence, to be patched inside or outside `target`
≡ `start` : an integer, the position at which to patch
≡ `filler` : an object, used for filling gaps. Defaults to ' '

Returns: A **sequence**, which looks like `target`, but a slice starting at `start` equals `source`.

Comments: In some cases, this call will result in the same result as replace().

If `source` doesn't fit into `target` because of the lengths and the supplied `start` value, gaps will be created, and `filler` is used to fill them in.

Notionally, `target` has an infinite amount of `filler` on both sides, and `start` counts position relative to where `target` actually starts. Then, notionally, a replace() operation is performed.

See Also: mid, replace

Example 1:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, 11,'0')
-- s is now "John Doe00abc"
```

Example 2:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, -1)
-- s is now "abcohn Doe"
Note that there was no gap to fill
Since -1 = 1 - 2, the patching started 2 positions before the initial 'J'
```

Example 3:

```
sequence source = "abc", target = "John Doe"
sequence s = patch(target, source, 6)
-- s is now "John Dabc"
```

pivot

returns a sequence of three sub-sequences. The sub-sequences contain

Signature:

```
pivot(object data_p, object pivot_p = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `data_p` : Either an atom or a list. An atom is treated as if it is one-element sequence.
≡ `pivot_p` : An object. Default is zero.

Returns: A **sequence**, { {less than pivot}, {equal to pivot}, {greater than pivot} }

Comments: `pivot()` is used as a split up a sequence relative to a specific value.

Example 1:

```
pivot( {7, 2, 8.5, 6, 6, -4.8, 6, 6, 3.341, -8, "text"}, 6 )
-- Ans: {{2, -4.8, 3.341, -8}, {6, 6, 6, 6}, {7, 8.5, "text"}}
pivot( {4, 1, -4, 6, -1, -7, 9, 10} )
-- Ans: {{-4, -1, -7}, {}, {4, 1, 6, 9, 10}}
pivot( 5 )
-- Ans: {{}, {}, {5}}
```

Example 2:

```
function quiksort(sequence s)
    if length(s) < 2 then
        return s
    end if

    sequence k = pivot(s, s[rand(length(s))])

    return quiksort(k[1]) & k[2] & quiksort(k[3])
end function

sequence t2 = {5,4,7,2,4,9,1,0,4,32,7,54,2,5,8,445,67}
? quiksort(t2) --> {0,1,2,2,4,4,4,5,5,7,7,8,9,32,54,67,445}
```

prepend

adds an object as the first element of a sequence.

Signature:

```
prepend(sequence target, object x)
```

```
<built-in> function
```

Arguments: ≡ `source` : the sequence to add to
≡ `x` : the object to add

Returns: A **sequence**, whose last elements are those of `target` and whose first element is `x`.

Comments: The length of the returned sequence will be `length(target) + 1` always.

If `x` is an atom this is the same as `result = x & target`. If `x` is a sequence it is not the same.

The case where `target` is prepended to itself is handled very efficiently.

See Also: append, &

Example 1:

```
prepend({1,2,3}, {0,0})  -- {{0,0}, 1, 2, 3}
-- Compare with concatenation:
{0,0} & {1,2,3}    -- {0, 0, 1, 2, 3}
```

Example 2:

```
s = {}
for i = 1 to 10 do
   s = prepend(s, i)
end for
-- s is {10,9,8,7,6,5,4,3,2,1}
```

project

creates a list of sequences based on selected elements from sequences in the source.

Signature:

```
project(sequence source, sequence coords)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : a list of sequences.

≡ `coords` : a list of index lists.

Returns: A **sequence**, with the same length as `source`. Each of its elements is a sequence, the length of `coords`. Each innermost sequence is made of the elements from the corresponding source sub-sequence.

Comments: For each sequence in `source`, a set of sub-sequences is created; one for each index list in `coords`. An index list is just a sequence containing indexes for items in a sequence.

See Also: vslice, extract

Example 1:

```
s = project({ "ABCD",  "789"},  {{1,2}, {3,1}, {2}})
-- s is {{"AB","CA","B"},{"78","97","8"}}
```

remove

removes an item, or a range of items from a sequence.

Signature:

```
remove(sequence target, atom start, atom stop=start)
```

```
<built-in> function
```

Arguments: ≡ `target` : the sequence to remove from.
≡ `start` : an atom, the (starting) index at which to remove
≡ `stop` : an atom, the index at which to stop removing (defaults to `start`)

Returns: A **sequence**, obtained from `target` by carving the `start..stop` slice out of it.

Comments: A new sequence is created. `target` can be a string or complex sequence.

See Also: replace, insert, splice, remove_all

Example 1:

```
s = remove("Johnn Doe", 4)
-- s is "John Doe"
```

Example 2:

```
s = remove({1,2,3,3,4}, 4)
-- s is {1,2,3,4}
```

Example 3:

```
s = remove("John Middle Doe", 6, 12)
-- s is "John Doe"
```

Example 4:

```
s = remove({1,2,3,3,4,4}, 4, 5)
-- s is {1,2,3,4}
```

remove_all

removes all occurrences of some object from a sequence.

Signature:

```
remove_all(object needle, sequence haystack)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `needle` : the object to remove.
≡ `haystack` : the sequence to remove from.

Returns: A **sequence**, of length at most `length(haystack)`, and which has the same elements, without any copy of `needle` left

Comments: This function weeds elements out, not sub-sequences.

See Also: remove, replace

Example 1:

```
s = remove_all( 1, {1,2,4,1,3,2,4,1,2,3} )
-- s is {2,4,3,2,4,2,3}
```

Example 2:

```
s = remove_all('x', "I'm toox secxksy for my shixrt.")
-- s is "I'm too secksy for my shirt."
```

remove_dups

removes duplicate elements.

Signature:

```
remove_dups(sequence source_data, integer proc_option = RD_PRESORTED)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source_data` : A sequence that may contain duplicated elements
≡ `proc_option` : One of RD_INPLACE, RD_PRESORTED, or RD_SORT.
…… ♦ RD_INPLACE removes items while preserving the original order of the unique items.
…… ♦ RD_PRESORTED assumes that the elements in `source_data` are already sorted. If they are not already sorted, this option merely removed adjacent duplicate elements.
…… ♦ RD_SORT will return the unique elements in ascending sorted order.

Returns: A **sequence**, that contains only the unique elements from `source_data`.

Example 1:

```
sequence s = { 4,7,9,7,2,5,5,9,0,4,4,5,6,5}
? remove_dups(s, RD_INPLACE) --> {4,7,9,2,5,0,6}
? remove_dups(s, RD_SORT) --> {0,2,4,5,6,7,9}
? remove_dups(s, RD_PRESORTED) --> {4,7,9,7,2,5,9,0,4,5,6,5}
? remove_dups(sort(s), RD_PRESORTED) --> {0,2,4,5,6,7,9}
```

remove_item

removes an item from the sequence.

Signature:

```
remove_item(object needle, sequence haystack)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `needle` : object to remove.
≡ `haystack` : sequence to remove it from.

Returns: A **sequence**, which is `haystack` with `needle` removed from it.

Comments: If `needle` is not in `haystack` then `haystack` is returned unchanged.

Example 1:

```
s = remove_item( 1, {3,4,2,1} ) --> {3,4,2}
s = remove_item( 5, {3,4,2,1} ) --> {3,4,2,1}
```

remove_subseq

removes all sub-sequences from the supplied sequence, optionally

Signature:

```
remove_subseq(sequence source_list, object alt_value = SEQ_NOALT)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source_list` : A sequence from which sub-sequences are removed.
≡ `alt_value` : An object. The default is SEQ_NOALT, which causes sub-sequences to be physically removed, otherwise any other value will be used to replace the sub-sequence.

Returns: A **sequence**, which contains only the atoms from `source_list` and optionally the `alt_value` where sub-sequences used to be.

Example 1:

```
sequence s = remove_subseq({4,6,"Apple",0.1, {1,2,3}, 4})
-- 's' is now {4, 6, 0.1, 4} -- length now 4
s = remove_subseq({4,6,"Apple",0.1, {1,2,3}, 4}, -1)
-- 's' is now {4, 6, -1, 0.1, -1, 4} -- length unchanged.
```

repeat

creates a sequence whose all elements are identical, with given length.

Signature:

```
repeat(object item, atom count)
```

```
<built-in> function
```

Arguments: ≡ `item` : an object, to which all elements of the result will be equal
≡ `count` : an atom, the requested length of the result sequence. This must be a value from zero to 0x3FFFFFFF. Any floating point values are first floored.

Returns: A **sequence**, of length `count` each element of which is `item`.

Comments: When you repeat() a sequence or a floating-point number the interpreter does not actually make multiple copies in memory. Rather, a single copy is "pointed to" a number of times.

See Also: repeat_pattern, series

Example 1:

```
repeat(0, 10)   -- {0,0,0,0,0,0,0,0,0,0}

repeat("JOHN", 4)  -- {"JOHN", "JOHN", "JOHN", "JOHN"}
-- The interpreter will create only one copy of "JOHN"
-- in memory and create a sequence containing four references to it.
```

repeat_pattern

returns a periodic sequence for a given a pattern and a count.

Signature:

```
repeat_pattern(object pattern, integer count)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `pattern` : the sequence whose elements are to be repeated
≡ `count` : an integer, the number of times the pattern is to be repeated.

Returns: A **sequence**, empty on failure, and of length `count*length(pattern)` otherwise. The first elements of the returned sequence are those of `pattern`. So are those that follow, on to the end.

See Also: repeat, series

Example 1:

```
s = repeat_pattern({1,2,5},3)
-- s is {1,2,5,1,2,5,1,2,5}
```

replace

replaces a slice in a sequence by an object.

Signature:

```
replace(sequence target, object replacement, integer start,
integer stop=start)
```

```
<built-in> function
```

Arguments:
≡ `target` : the sequence in which replacement will be done.
≡ `replacement` : an object, the item to replace with.
≡ `start` : an integer, the starting index of the slice to replace.
≡ `stop` : an integer, the stopping index of the slice to replace.

Returns: A **sequence**, which is made of `target` with the `start..stop` slice removed and replaced by `replacement`, which is splice()d in.

Comments:

• A new sequence is created. `target` can be a string or complex sequence of any shape.

• To replace by just one element, enclose `replacement` in curly braces, which will be removed at replace time.

See Also: splice, remove, remove_all

Example 1:

```
s = replace("John Middle Doe", "Smith", 6, 11)
-- s is "John Smith Doe"

s = replace({45.3, "John", 5, {10, 20}}, 25, 2, 3)
-- s is {45.3, 25, {10, 20}}
```

retain_all

keeps all occurrences of a set of objects from a sequence and removes all others.

Signature:

```
retain_all(object needles, sequence haystack)
```

```
public function
include sequence.e
```

```
namespace stdseq
```

Arguments: ≡ `needles` : the set of objects to retain.
≡ `haystack` : the sequence to remove items not in `needles`.

Returns: A **sequence** containing only those objects from `haystack` that are also in `needles`.

See Also: remove, replace, remove_all

Example 1:

```
 s = retain_all( {1,3,5}, {1,2,4,1,3,2,4,1,2,3} ) --> {1,1,3,1,3}
 s = retain_all("0123456789", "+34 (04) 555-44392") -> "340455544392"
```

reverse

reverses the order of elements in a sequence.

Signature:

```
reverse(object target, integer pFrom = 1, integer pTo = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `target` : the sequence to reverse.
≡ `pFrom` : an integer, the starting point. Defaults to 1.
≡ `pTo` : an integer, the end point. Defaults to 0.

Returns: A **sequence**, if `target` is a sequence, the same length as `target` and the same elements, but those with index between `pFrom` and `pTo` appear in reverse order.

Comments: In the result sequence, some or all top-level elements appear in reverse order compared to the original sequence. This does not reverse any sub-sequences found in the original sequence.

The `pTo` parameter can be negative, which indicates an offset from the last element. Thus `-1` means the second-last element and `0` means the last element.

Example 1:

```
 reverse({1,3,5,7})          -- {7,5,3,1}
 reverse({1,3,5,7,9}, 2, -1) -- {1,7,5,3,9}
 reverse({1,3,5,7,9}, 2)     -- {1,9,7,5,3}
 reverse({{1,2,3}, {4,5,6}}) -- {{4,5,6}, {1,2,3}}
 reverse({99})               -- {99}
 reverse({})                 -- {}
 reverse(42)                 -- 42
```

rotate

rotates a slice of a sequence.

Signature:

```
rotate(sequence source, integer shift, integer start = 1,
integer stop = length(source))
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : sequence to be rotated
≡ `shift` : direction and count to be shifted (`ROTATE_LEFT` or `ROTATE_RIGHT`)
≡ `start` : starting position for shift, defaults o 1
≡ `stop` : stopping position for shift, defaults to `length(source)`

Comments: Use `amount * direction` to specify the shift. direction is either `ROTATE_LEFT` or `ROTATE_RIGHT`. This enables to shift multiple places in a single call. For instance, use `ROTATE_LEFT * 5` to rotate left, 5 positions.

A null shift does nothing and returns source unchanged.

See Also: slice, head, tail

Example 1:

```
s = rotate({1, 2, 3, 4, 5}, ROTATE_LEFT)
-- s is {2, 3, 4, 5, 1}
```

Example 2:

```
s = rotate({1, 2, 3, 4, 5}, ROTATE_RIGHT * 2)
-- s is {4, 5, 1, 2, 3}
```

Example 3:

```
s = rotate({11,13,15,17,19,23}, ROTATE_LEFT, 2, 5)
-- s is {11,15,17,19,13,23}
```

Example 4:

```
s = rotate({11,13,15,17,19,23}, ROTATE_RIGHT, 2, 5)
-- s is {11,19,13,15,17,23}
```

series

returns a new sequence built as a series from a given object.

Signature:

```
series(object start, object increment, integer count = 2,
integer op = '+')
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `start` : the initial value from which to start
≡ `increment` : the value to recursively add to `start` to get new elements
≡ `count` : an integer, the number of items in the returned sequence. The default is 2.
≡ `operation` : an integer, the type of operation used to build the series. Can be either '+' for a linear series or '*' for a geometric series. The default is '+'.

Returns: An **object**, either 0 on failure or a sequence containing the series.

Comments:

- The first item in the returned series is always `start`.
- A *linear* series is formed by **adding** `increment` to `start`.
- A *geometric* series is formed by **multiplying** `increment` by `start`.
- If `count` is negative, or if `start` **op** `increment` is invalid, then 0 is returned.

Otherwise, a sequence, of length `count+1`, staring with `start` and whose adjacent elements differ by `increment`, is returned.

See Also: repeat_pattern

Example 1:

```
s = series( 1, 4, 5)
-- s is {1, 5, 9, 13, 17}
s = series( 1, 2, 6, '*')
-- s is {1, 2, 4, 8, 16, 32}
s = series({1,2,3}, 4, 2)
-- s is {{1,2,3}, {5,6,7}}
s = series({1,2,3}, {4,-1,10}, 2)
-- s is {{1,2,3}, {5,1,13}}
```

shuffle

shuffles the elements of a sequence.

Signature:

```
shuffle(object seq)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `seq`: the sequence to shuffle.

Returns: A **sequence**

Comments: The input sequence does not have to be in any specific order and can contain duplicates. The output will be in an unpredictable order, which might even be the same as the input order.

Example 1:

```
 shuffle({1,2,3,3}) -- {3,1,3,2}
 shuffle({1,2,3,3}) -- {2,3,1,3}
 shuffle({1,2,3,3}) -- {1,2,3,3}
```

sim_index

calculates the similarity between two sequences.

Signature:

```
sim_index(sequence A, sequence B)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `A` : A sequence.
≡ `B` : A sequence.

Returns: An **atom**, the closer to zero, the more the two sequences are alike.

Comments: The calculation is weighted to give mismatched elements towards the front of the sequences larger scores. This means that sequences that differ near the begining are considered more un-alike than mismatches towards the end of the sequences. Also, unmatched elements from the first sequence are weighted more than unmatched elements from the second sequence.

Two identical sequences return zero. A non-zero means that they are not the same and larger values indicate a larger differences.

Example 1:

```
 ? sim_index("sit",      "sin")      --> 0.08784
 ? sim_index("sit",      "sat")      --> 0.32394
 ? sim_index("sit",      "skit")     --> 0.34324
 ? sim_index("sit",      "its")      --> 0.68293
 ? sim_index("sit",      "kit")      --> 0.86603

 ? sim_index("knitting", "knitting") --> 0.00000
 ? sim_index("kitting",  "kitten")   --> 0.09068
 ? sim_index("knitting", "knotting") --> 0.27717
 ? sim_index("knitting", "kitten")   --> 0.35332
 ? sim_index("abacus","zoological")  --> 0.76304
```

slice

returns a portion of the supplied sequence.

Signature:

```
slice(sequence source, atom start = 1, atom stop = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `source` : the sequence from which to get a portion
≡ `start` : an integer, the starting point of the portion. Default is 1.
≡ `stop` : an integer, the ending point of the portion. Default is length(source).

Returns: A **sequence**.

Comments:

• If the supplied `start` is less than 1 then it set to 1.
• If the supplied `stop` is less than 1 then `length(source)` is added to it. In this way, 0 represents the end of `source`, -1 represents one element in from the end of `source` and so on.
• If the supplied `stop` is greater than `length(source)` then it is set to the end.
• After these adjustments, and if `source[start..stop]` makes sense, it is returned, otherwise, `{}` is returned.

See Also: head, mid, tail

Example 1:

```
s2 = slice("John Doe", 6, 8)--> "Doe"
s2 = slice("John Doe", 6, 50) --> "Doe"
s2 = slice({1, 5.4, "John", 30}, 2, 3) --> {5.4, "John"}
s2 = slice({1,2,3,4,5}, 2, -1) --> {2,3,4}
s2 = slice({1,2,3,4,5}, 2) --> {2,3,4,5}
s2 = slice({1,2,3,4,5}, , 4) --> {1,2,3,4}
```

splice

inserts an object as a new slice in a sequence at a given position.

Signature:

```
splice(sequence target, object what, integer index)
```

```
<built-in> function
```

Arguments:
≡ `target` : the sequence to insert into
≡ `what` : the object to insert
≡ `index` : an integer, the position in `target` where `what` should appear

Returns: A **sequence**, which is `target` with one or more elements, those of `what`, inserted at locations starting at `index`.

Comments: `target` can be a sequence of any shape, and `what` any kind of object.

The length of this new sequence is the sum of the lengths of `target` and `what`. `splice`() is equivalent to insert() when `what` is an atom, but not when it is a sequence.

Splicing a string into a string results into a new string.

See Also: insert, remove, replace, &

Example 1:

```
s = splice("John Doe", " Middle", 5)
-- s is "John Middle Doe"
```

Example 2:

```
s = splice({10,30,40}, 20, 2)
-- s is {10,20,30,40}
```

split

splits a sequence on separator delimiters into a number of sub-sequences.

Signature:

```
split(sequence st, object delim = ' ', integer no_empty = 0,
integer limit = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence to split.
≡ `delim` : an object (default is ' '). The delimiter that separates items in `source`.
≡ `no_empty` : an integer (default is 0). If not zero then all zero-length sub-sequences are removed from the returned sequence. Use this when leading, trailing and duplicated delimiters are not significant.
≡ `limit` : an integer (default is 0). The maximum number of sub-sequences to create. If zero, there is no limit.

Returns: A **sequence**, of sub-sequences of `source`. Delimiters are removed.

Comments: This function may be applied to a string sequence or a complex sequence.

If `limit` is > 0, this is the maximum number of sub-sequences that will created, otherwise there is no limit.

See Also: split_any, breakup, join

Example 1:

```
result = split("John Middle Doe")
-- result is {"John", "Middle", "Doe"}
```

Example 2:

```
result = split("John,Middle,Doe", ",",, 2) -- Only want 2 sub-sequences.
-- result is {"John", "Middle,Doe"}
```

Example 3:

```
result = split("John||Middle||Doe|", '|') -- Each '|' is significant by default
-- result is {"John","","Middle","","Doe",""}
result = split("John||Middle||Doe|", '|', 1) -- Adjacent '|' are just a single delim,
                                              -- and leading/trailing '|' ignored.
-- result is {"John","Middle","Doe"}
```

split_any

splits a sequence by any of the separators in the list of delimiters;

Signature:

```
split_any(sequence source, object delim = ", \t|", integer limit = 0,
integer no_empty = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence to split.

≡ `delim` : a list of delimiters to split by. The default set is comma, space, tab and bar.
≡ `limit` : an integer (default is 0). The maximum number of sub-sequences to create. If zero, there is no limit.
≡ `no_empty` : an integer (default is 0). If not zero then all zero-length sub-sequences removed from the returned sequence. Use this when leading, trailing and duplicated delimiters are not significant.

Comments:

• This function may be applied to a string sequence or a complex sequence.
• It works like `split`(), but in this case `delim` is a set of potential delimiters rather than a single delimiter.
• If `delim` is an empty set, the `source` is returned in a sequence.

See Also: split, breakup, join

Example 1:

```
result = split_any("One,Two|Three Four") -- Default delims
-- result is {"One", "Two", "Three", "Four"}
result = split_any("192.168.1.103:8080", ".:") -- Using dot and colon
-- result is {"192","168","1","103","8080"}
result = split_any("One,Two|Three Four",, 2) -- limited to two splits
-- result is {"One", "Two", "Three Four"}
result = split_any(",One,,Two| Three|| Four,"  ) -- Allow Empty option
-- result is {"","One","","Two","","Three","","","Four",""}
result = split_any(",One,,Two| Three|| Four,",,,1) -- No Empty option
-- result is {"One", "Two", "Three", "Four"}
result = split_any(",One,,Two| Three|| Four,", "") -- Empty delimiters
-- result is {",One,,Two| Three|| Four,"}
```

store

stores an object at a location nested arbitrarily deeply within a sequence.

Signature:

```
store(sequence target, sequence indexes, object x)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `target` : the sequence in which to store something
≡ `indexes` : a sequence of integers, the path to follow to reach the place where to store
≡ `x` : the object to store.

Returns: A **sequence**, a **copy** of `target` with the specified place `indexes` modified by storing `x` into it.

Comments: If the last element of `indexes` is a pair of integers, `x` will be stored as a slice three, the bounding indexes being given in the pair as {lower,upper}..

An object passed as an argument to a routine is always a copy of the original; changes to this copy never alter the original object. To modify the original object you always have to explicitly assign the output of a function back to the original.

Arguments in Euphoria behave as if they follow the conventional "pass by value" paradigm, but use references for efficiency. Actual copying is performed only when necessary.

See Also: fetch, Subscripting of Sequences

Example 1:

```
s = store({0,1,2,3,{"abc","def","ghi"},6},{5,2,3},108)
-- s is {0,1,2,3,{"abc","del","ghi"},6}
```

tail

return the last `size` item or items of a sequence.

Signature:

```
tail(sequence source, atom size=length(source) - 1)
```

```
<built-in> function
```

Arguments: ≡ `source` : the sequence to get the tail of.
≡ `size` : an integer, the number of items to return. (defaults to length(source) - 1)

Returns: A **sequence**, of length at most `size`. If the length is less than `size`, then `source` was returned. Otherwise, the `size` last elements of `source` were returned.

Comments: `source` can be any type of sequence, including nested sequences.

See Also: head, mid, slice

Example 1:

```
s2 = tail("John Doe", 3)
-- s2 is "Doe"
```

Example 2:

```
s2 = tail("John Doe", 50)
-- s2 is "John Doe"
```

Example 3:

```
s2 = tail({1, 5.4, "John", 30}, 3)
-- s2 is {5.4, "John", 30}
```

transform

transforms the input sequence by using one or more user-supplied transformers.

Signature:

```
transform(sequence source_data, object transformer_rids)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source_data` : A sequence to be transformed.
≡ `transformer_rids` : An object. One or more routine_ids used to transform the input.

Returns: The source **sequence**, that has been transformed.

Comments:

• This works by calling each transformer in order, passing to it the result of the previous transformation. Of course, the first transformer gets the original sequence as passed to this routine.
• Each transformer routine takes one or more parameters. The first is a source sequence to be transformed and others are any user data that may have been supplied to the `transform` routine.
• Each transformer routine returns a transformed sequence.
• The `transformer_rids` parameters is either a single routine_id or a sequence of routine_ids. In this second case, the routine_id may actually be a multi-element sequence containing the real routine_id and some user data to pass to the transformer routine. If there is no user data then the transformer is called with only one parameter.

Example 1:

```
res = transform(" hello    ", {
    { routine_id("trim"), " ", 0 },
    routine_id("upper")
})
--> "HELLO"
```

transmute

replaces all instances of any element from the current_items sequence that occur in the

Signature:

```
transmute(sequence source_data, sequence current_items, sequence new_items,
integer start = 1, integer limit = length(source_data))
```

```
public function
include sequence.e
namespace stdseq
```

Arguments:
≡ `source_data` : a sequence, the data that might contain elements from `current_items`
≡ `current_items` : a sequence, the set of items to look for in `source_data`. Matching data is replaced with the corresponding data from `new_items`.
≡ `new_items` : a sequence, the set of replacement data for any matches found.
≡ `start` : an integer, the starting point of the search. Defaults to 1.
≡ `limit` : an integer, the maximum number of replacements to be made. Defaults to length(source_data).

Returns: A **sequence**, an updated version of `source_data`.

Comments: By default, this routine operates on single elements from each of the arguments. That is to say, it scans `source_data` for elements that match any single element in `current_items` and when matched, replaces that with a single element from `new_items`.

See Also: find, match, replace, mapping

Example 1:

```
transmute(SomeString, "hts", "123")
```

Example 2:

For example, to find all occurrances of "sh","th", and "sch" you have the `current_items` as `{{}, "sh", "th", "sch"}`. Note that for the purposes of determine the corresponding replacement data, the leading empty sequence is not counted, so in this example "th" is the second item.

```
res = transmute("the school shoes", {{}, "sh", "th", "sch"}, "123")
  -- res becomes "2e 3ool 1oes"
```

Example 3:

```
res = transmute("the school shoes", {{}, "sh", "th", "sch"}, {{}, "SH", "TH", "SCH"})
  -- res becomes "THe SCHool SHoes"
```

Example 4:

```
res = transmute("the school shoes", {{}, "sh", "th", "sch"}, {{}, "", "", ""})
  -- res becomes "e ool oes"
```

Example 5:

```
res = transmute("the school shoes", {{}, "sh", 't', "sch"}, {{}, 'x', "TH", "SCH"})
  -- res becomes "THhe SCHool xoes"
```

Example 6:

```
res = transmute("John Smith enjoys uncooked apples.", "aeiouy", "YUOIEA")
-- res is "JIhn SmOth UnjIAs EncIIkUd YpplUs."
```

valid_index

checks whether a valid index exists in a sequence.

Signature:

```
valid_index(sequence st, object x)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `s` : the sequence for which to check
≡ `x` : an object, the index to check.

Returns: An **integer**, 1 if `s[x]` is valid, or else 0.

See Also: Subscripting of Sequences

Example 1:

```
i = valid_index({51,27,33,14},2)
--> i is 1
```

vslice

performs a vertical slice on a nested sequence

Signature:

```
vslice(sequence source, atom colno, object error_control = 0)
```

```
public function
include sequence.e
namespace stdseq
```

Arguments: ≡ `source` : the sequence to take a vertical slice from
≡ `colno` : an atom, the column number to extract (rounded down)
≡ `error_control` : an object which says what to do if some element does not exist. Defaults to 0 (crash in such a circumstance).

Returns: A **sequence**, usually of the same length as `source`, made of all the `source[x][colno]`.

Comments: If it is not possible to return the sequence of all `source[x][colno]]` for all available `x`, the outcome is decided by `error_control`:
• If 0 (the default), program is aborted.
• If a nonzero atom, the short vertical slice is returned.
• Otherwise, elements of `error_control` will be taken to make for any missing element. The elements are selected from the first to the last, as needed and this cycles again from the first.

See Also: slice, project

Example 1:

```
s = vslice({{5,1}, {5,2}, {5,3}}, 2)
-- s is {1,2,3}

s = vslice({{5,1}, {5,2}, {5,3}}, 1)
-- s is {5,5,5}
```

serialize

Routines

serialize API

deserialize

convert a serialized object into a standard Euphoria object.

Signature:

```
deserialize(object sdata, integer pos = 1)
```

```
public function
include serialize.e
namespace serialize
```

Arguments: ≡ `sdata` : either a sequence containing one or more concatenated serialized objects or an open file handle. If this is a file handle, the current position in the file is assumed to be at a serialized object in the file.
≡ `pos` : optional index into `sdata`. If omitted 1 is assumed. The index must point to the start of a serialized object.

Returns: The return **value**, depends on the input type.
• If `sdata` is a file handle then this function returns a Euphoria object that had been stored in the file, and moves the current file to the first byte after the stored object.
• If `sdata` is a sequence then this returns a two-element sequence. The *first* element is the Euphoria object that corresponds to the serialized object that begins at index `pos`, and the *second* element is the index position in the input parameter just after the serialized object.

Comments: A serialized object is one that has been returned from the serialize function.

Example 1:

```
 sequence objcache
  objcache = serialize(FirstName) &
             serialize(LastName) &
             serialize(PhoneNumber) &
             serialize(Address)

  sequence res
  integer pos = 1
  res = deserialize( objcache , pos)
  FirstName = res[1] pos = res[2]
  res = deserialize( objcache , pos)
  LastName = res[1] pos = res[2]
  res = deserialize( objcache , pos)
  PhoneNumber = res[1] pos = res[2]
  res = deserialize( objcache , pos)
  Address = res[1] pos = res[2]
```

Example 2:

```
 sequence objcache
  objcache = serialize({FirstName,
                        LastName,
                        PhoneNumber,
                            Address})

  sequence res
  res = deserialize( objcache )
  FirstName = res[1][1]
  LastName = res[1][2]
  PhoneNumber = res[1][3]
  Address = res[1][4]
```

Example 3:

```
 integer fh
  fh = open("cust.dat", "wb")
  puts(fh, serialize(FirstName))
  puts(fh, serialize(LastName))
  puts(fh, serialize(PhoneNumber))
  puts(fh, serialize(Address))
  close(fh)

  fh = open("cust.dat", "rb")
  FirstName = deserialize(fh)
  LastName = deserialize(fh)
  PhoneNumber = deserialize(fh)
  Address = deserialize(fh)
  close(fh)
```

Example 4:

```
 integer fh
  fh = open("cust.dat", "wb")
  puts(fh, serialize({FirstName,
                      LastName,
                      PhoneNumber,
                      Address}))
  close(fh)

  sequence res
  fh = open("cust.dat", "rb")
  res = deserialize(fh)
  close(fh)
  FirstName = res[1]
  LastName = res[2]
  PhoneNumber = res[3]
  Address = res[4]
```

dump

saves a Euphoria object to disk in a binary format.

Signature:

```
dump(sequence data, sequence filename)
```

```
public function
include serialize.e
namespace serialize
```

Arguments:
≡ `data` : any Euphoria object.
≡ `filename` : the name of the file to save it to.

Returns: An **integer**, 0 if the function fails, otherwise the number of bytes in the created file.

Comments: If the named file does not exist it is created, otherwise it is overwritten.

You can use the load function to recover the data from the file.

Example 1:

```
 include std/serialize.e
 integer size = dump(myData, theFileName)
 if size = 0 then
     puts(1, "Failed to save data to file\n")
```

```
else
    printf(1, "Saved file is %d bytes long\n", size)
end if
```

load

restores a Euphoria object that has been saved to disk by dump.

Signature:

```
load(sequence filename)
```

```
public function
include serialize.e
namespace serialize
```

Arguments: ≡ `filename` : the name of the file to restore it from.

Returns: A **sequence**, the first element is the result code. If the result code is 0 then it means that the function failed, otherwise the restored data is in the second element.

Comments: This is used to load back data from a file created by the dump function.

Example 1:

```
include std/serialize.e
sequence mydata = load(theFileName)
if mydata[1] = 0 then
    puts(1, "Failed to load data from file\n")
else
    mydata = mydata[2] -- Restored data is in second element.
end if
```

serialize

converts a standard Euphoria object into a serialized version of it.

Signature:

```
serialize(object x)
```

```
public function
include serialize.e
namespace serialize
```

Arguments: ≡ `euobj` : any Euphoria object.

Returns: A **sequence**, this is the serialized version of the input object.

Comments: A serialized object is one that has been converted to a set of byte values. This can then by written directly out to a file for storage.

You can use the deserialize function to convert it back into a standard Euphoria object.

Example 1:

```
integer fh
 fh = open("cust.dat", "wb")
 puts(fh, serialize(FirstName))
 puts(fh, serialize(LastName))
 puts(fh, serialize(PhoneNumber))
 puts(fh, serialize(Address))
 close(fh)

 fh = open("cust.dat", "rb")
 FirstName = deserialize(fh)
 LastName = deserialize(fh)
 PhoneNumber = deserialize(fh)
 Address = deserialize(fh)
```

```
          close(fh)
```

Example 2:

```
         integer fh
          fh = open("cust.dat", "wb")
          puts(fh, serialize({FirstName,
                              LastName,
                              PhoneNumber,
                              Address}))
          close(fh)

          sequence res
          fh = open("cust.dat", "rb")
          res = deserialize(fh)
          close(fh)
          FirstName = res[1]
          LastName = res[2]
          PhoneNumber = res[3]
          Address = res[4]
```

---

socket

Error Information

- error_code
- OK
- ERR_ACCESS
- ERR_ADDRINUSE
- ERR_ADDRNOTAVAIL
- ERR_AFNOSUPPORT
- ERR_AGAIN
- ERR_ALREADY
- ERR_CONNABORTED
- ERR_CONNREFUSED
- ERR_CONNRESET
- ERR_DESTADDRREQ
- ERR_FAULT
- ERR_HOSTUNREACH
- ERR_INPROGRESS
- ERR_INTR
- ERR_INVAL
- ERR_IO
- ERR_ISCONN
- ERR_ISDIR
- ERR_LOOP
- ERR_MFILE
- ERR_MSGSIZE
- ERR_NAMETOOLONG
- ERR_NETDOWN
- ERR_NETRESET
- ERR_NETUNREACH
- ERR_NFILE
- ERR_NOBUFS
- ERR_NOENT
- ERR_NOTCONN
- ERR_NOTDIR

Socket Backend Constants

These values are used by the Euphoria backend to pass information to this library. The TYPE constants are used to identify to the info function which family of constants are being retrieved (AF protocols, socket types, and socket options, respectively).

Socket Type Euphoria Constants

These values are used to retrieve the known values for `family` and `sock_type` parameters of the create function from the Euphoria backend. (The reason for doing it this way is to retrieve the values defined in C, instead of duplicating them here.) These constants are guarranteed to never change, and to be the same value across platforms.

Socket Type Constants

These values are passed as the `family` and `sock_type` parameters of the create function. They are OS-dependent.

Select Accessor Constants

Use with the result of select.

Shutdown Options

Pass one of the following to the `method` parameter of shutdown.

Socket Options

Pass to the `optname` parameter of the functions get_option and set_option.

These options are highly OS specific and are normally not needed for most socket communication. They are provided here for your convenience. If you should need to set socket options, please refer to your OS reference material.

There may be other values that your OS defines and some defined here are not supported on all operating systems.

Socket Options In Common

Send Flags

Pass to the `flags` parameter of send and receive

socket API

AF_APPLETALK

Appletalk

Signature:

```
AF_APPLETALK
```

```
public constant
include socket.e
namespace sockets
```

AF_BTH

Bluetooth (currently Windows-only)

Signature:

```
AF_BTH
```

```
public constant
include socket.e
namespace sockets
```

AF_INET

IPv4 Internet protocols

Signature:

```
AF_INET
```

```
public constant
include socket.e
namespace sockets
```

AF_INET6

IPv6 Internet protocols

Signature:

```
AF_INET6
```

```
public constant
include socket.e
namespace sockets
```

AF_UNIX

Local communications

Signature:

```
AF_UNIX
```

```
public constant
include socket.e
namespace sockets
```

AF_UNSPEC

Address family is unspecified

Signature:

```
AF_UNSPEC
```

```
public constant
include socket.e
namespace sockets
```

EAF_APPLETALK

Appletalk

Signature:

```
EAF_APPLETALK
```

```
public constant
include socket.e
namespace sockets
```

EAF_BTH

Bluetooth (currently Windows-only)

Signature:

```
EAF_BTH
```

```
public constant
include socket.e
namespace sockets
```

EAF_INET

IPv4 Internet protocols

Signature:

```
EAF_INET
```

```
public constant
include socket.e
namespace sockets
```

EAF_INET6

IPv6 Internet protocols

Signature:

```
EAF_INET6
```

```
public constant
include socket.e
namespace sockets
```

EAF_UNIX

Local communications

Signature:

```
EAF_UNIX
```

```
public constant
include socket.e
namespace sockets
```

EAF_UNSPEC

Address family is unspecified

Signature:

```
EAF_UNSPEC
```

```
public constant
include socket.e
namespace sockets
```

ERR_ACCESS

Permission has been denied. This can happen when using a send_to call on a broadcast

Signature:

```
ERR_ACCESS
```

```
public constant
include socket.e
namespace sockets
```

ERR_ADDRINUSE

Address is already in use.

Signature:

```
ERR_ADDRINUSE
```

```
public constant
include socket.e
namespace sockets
```

ERR_ADDRNOTAVAIL

The specified address is not a valid local IP address on this computer.

Signature:

```
ERR_ADDRNOTAVAIL
```

```
public constant
include socket.e
namespace sockets
```

ERR_AFNOSUPPORT

Address family not supported by the protocol family.

Signature:

```
ERR_AFNOSUPPORT
```

```
public constant
include socket.e
namespace sockets
```

ERR_AGAIN

Kernel resources to complete the request are temporarly unavailable.

Signature:

```
ERR_AGAIN
```

```
public constant
include socket.e
namespace sockets
```

ERR_ALREADY

Operation is already in progress.

Signature:

```
ERR_ALREADY
```

```
public constant
include socket.e
namespace sockets
```

ERR_CONNABORTED

Software has caused a connection to be aborted.

Signature:

```
ERR_CONNABORTED
```

```
public constant
include socket.e
namespace sockets
```

ERR_CONNREFUSED

Connection was refused.

Signature:

```
ERR_CONNREFUSED
```

```
public constant
include socket.e
namespace sockets
```

ERR_CONNRESET

An incomming connection was supplied however it was terminated by the remote peer.

Signature:

```
ERR_CONNRESET
```

```
public constant
include socket.e
namespace sockets
```

ERR_DESTADDRREQ

Destination address required.

Signature:

```
ERR_DESTADDRREQ
```

```
public constant
include socket.e
namespace sockets
```

ERR_FAULT

Address creation has failed internally.

Signature:

```
ERR_FAULT
```

```
public constant
include socket.e
namespace sockets
```

ERR_HOSTUNREACH

No route to the host specified could be found.

Signature:

```
ERR_HOSTUNREACH
```

```
public constant
include socket.e
namespace sockets
```

ERR_INPROGRESS

A blocking call is inprogress.

Signature:

```
ERR_INPROGRESS
```

```
public constant
include socket.e
namespace sockets
```

ERR_INTR

A blocking call was cancelled or interrupted.

Signature:

```
ERR_INTR
```

```
public constant
include socket.e
```

```
namespace sockets
```

ERR_INVAL

An invalid sequence of command calls were made, for instance trying to `accept`

Signature:

```
ERR_INVAL
```

```
public constant
include socket.e
namespace sockets
```

ERR_IO

An I/O error occurred while making the directory entry or allocating the

Signature:

```
ERR_IO
```

```
public constant
include socket.e
namespace sockets
```

ERR_ISCONN

Socket is already connected.

Signature:

```
ERR_ISCONN
```

```
public constant
include socket.e
namespace sockets
```

ERR_ISDIR

An empty pathname was specified. (Unix Domain Socket).

Signature:

```
ERR_ISDIR
```

```
public constant
include socket.e
namespace sockets
```

ERR_LOOP

Too many symbolic links were encountered. (Unix Domain Socket).

Signature:

```
ERR_LOOP
```

```
public constant
include socket.e
namespace sockets
```

ERR_MFILE

The queue is not empty upon routine call.

Signature:

```
ERR_MFILE
```

```
public constant
include socket.e
namespace sockets
```

ERR_MSGSIZE

Message is too long for buffer size. This would indicate an internal error to

Signature:

```
ERR_MSGSIZE
```

```
public constant
include socket.e
namespace sockets
```

ERR_NAMETOOLONG

Component of the path name exceeded 255 characters or the entire path

Signature:

```
ERR_NAMETOOLONG
```

```
public constant
include socket.e
namespace sockets
```

ERR_NETDOWN

The network subsystem is down or has failed

Signature:

```
ERR_NETDOWN
```

```
public constant
include socket.e
namespace sockets
```

ERR_NETRESET

Network has dropped it's connection on reset.

Signature:

```
ERR_NETRESET
```

```
public constant
include socket.e
namespace sockets
```

ERR_NETUNREACH

Network is unreachable.

Signature:

```
ERR_NETUNREACH
```

```
public constant
include socket.e
namespace sockets
```

ERR_NFILE

Not a file. (Unix Domain Sockets).

Signature:

```
ERR_NFILE
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOBUFS

No buffer space is available.

Signature:

```
ERR_NOBUFS
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOENT

Named socket does not exist. (Unix Domain Socket).

Signature:

```
ERR_NOENT
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOTCONN

Socket is not connected.

Signature:

```
ERR_NOTCONN
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOTDIR

Component of the path prefix is not a directory. (Unix Domain Socket).

Signature:

```
ERR_NOTDIR
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOTINITIALISED

Socket system is not initialized (Windows only)

Signature:

```
ERR_NOTINITIALISED
```

```
public constant
include socket.e
namespace sockets
```

ERR_NOTSOCK

The descriptor is not a socket.

Signature:

```
ERR_NOTSOCK
```

```
public constant
include socket.e
namespace sockets
```

ERR_OPNOTSUPP

Operation is not supported on this type of socket.

Signature:

```
ERR_OPNOTSUPP
```

```
public constant
include socket.e
namespace sockets
```

ERR_PROTONOSUPPORT

Protocol not supported.

Signature:

```
ERR_PROTONOSUPPORT
```

```
public constant
include socket.e
namespace sockets
```

ERR_PROTOTYPE

Protocol is the wrong type for the socket.

Signature:

```
ERR_PROTOTYPE
```

```
public constant
include socket.e
namespace sockets
```

ERR_ROFS

The name would reside on a read-only file system. (Unix Domain Socket).

Signature:

```
ERR_ROFS
```

```
public constant
include socket.e
namespace sockets
```

ERR_SHUTDOWN

The socket has been shutdown. Possibly a send/receive call after a shutdown took

Signature:

```
ERR_SHUTDOWN
```

```
public constant
include socket.e
namespace sockets
```

ERR_SOCKTNOSUPPORT

Socket type is not supported.

Signature:

```
ERR_SOCKTNOSUPPORT
```

```
public constant
include socket.e
namespace sockets
```

ERR_TIMEDOUT

Connection has timed out.

Signature:

```
ERR_TIMEDOUT
```

```
public constant
include socket.e
namespace sockets
```

ERR_WOULDBLOCK

The operation would block on a socket marked as non-blocking.

Signature:

```
ERR_WOULDBLOCK
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_DGRAM

Supports datagrams (connectionless, unreliable messages of a

Signature:

```
ESOCK_DGRAM
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_RAW

Provides raw network protocol access.

Signature:

```
ESOCK_RAW
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_RDM

Provides a reliable datagram layer that does not guarantee ordering.

Signature:

```
ESOCK_RDM
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_SEQPACKET

Obsolete and should not be used in new programs

Signature:

```
ESOCK_SEQPACKET
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_STREAM

Provides sequenced, reliable, two-way, connection-based byte streams.

Signature:

```
ESOCK_STREAM
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_TYPE_AF

Signature:

```
ESOCK_TYPE_AF
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_TYPE_OPTION

These values are used to retrieve the known values for `family` and

Signature:

```
ESOCK_TYPE_OPTION
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_TYPE_TYPE

Signature:

```
ESOCK_TYPE_TYPE
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_UNDEFINED_VALUE

when a particular constant was not defined by C,the backend returns this value

Signature:

```
ESOCK_UNDEFINED_VALUE
```

```
public constant
include socket.e
namespace sockets
```

ESOCK_UNKNOWN_FLAG

if the backend doesn't recognize the flag in question

Signature:

```
ESOCK_UNKNOWN_FLAG
```

```
public constant
include socket.e
namespace sockets
```

MSG_CONFIRM

Tell the link layer that forward progress happened: you got a

Signature:

```
MSG_CONFIRM
```

```
public constant
include socket.e
namespace sockets
```

MSG_CTRUNC

indicates that some control data were discarded due to lack of space in

Signature:

```
MSG_CTRUNC
```

```
public constant
include socket.e
namespace sockets
```

MSG_DONTROUTE

Do not use a gateway to send out the packet, only send to hosts on

Signature:

```
MSG_DONTROUTE
```

```
public constant
include socket.e
namespace sockets
```

MSG_DONTWAIT

Enables non-blocking operation; if the operation would block, EAGAIN

Signature:

```
MSG_DONTWAIT
```

```
public constant
include socket.e
namespace sockets
```

MSG_EOR

Terminates a record (when this notion is supported, as for sockets of

Signature:

```
MSG_EOR
```

```
public constant
include socket.e
namespace sockets
```

MSG_ERRQUEUE

indicates that no data was received but an extended error from the

Signature:

```
MSG_ERRQUEUE
```

```
public constant
include socket.e
namespace sockets
```

MSG_FIN

Signature:

```
MSG_FIN
```

```
public constant
include socket.e
namespace sockets
```

MSG_MORE

The caller has more data to send. This flag is used with TCP sockets

Signature:

```
MSG_MORE
```

```
public constant
include socket.e
namespace sockets
```

MSG_NOSIGNAL

Requests not to send SIGPIPE on errors on stream oriented sockets when

Signature:

```
MSG_NOSIGNAL
```

```
public constant
include socket.e
namespace sockets
```

MSG_OOB

Sends out-of-band data on sockets that support this notion (e.g., of

Signature:

```
MSG_OOB
```

```
public constant
include socket.e
namespace sockets
```

MSG_PEEK

This flag causes the receive operation to return data from the

Signature:

```
MSG_PEEK
```

```
public constant
include socket.e
namespace sockets
```

MSG_PROXY

Signature:

```
MSG_PROXY
```

```
public constant
include socket.e
namespace sockets
```

MSG_RST

Signature:

```
MSG_RST
```

```
public constant
include socket.e
namespace sockets
```

MSG_SYN

Signature:

```
MSG_SYN
```

```
public constant
include socket.e
namespace sockets
```

MSG_TRUNC

indicates that the trailing portion of a datagram was discarded because

Signature:

```
MSG_TRUNC
```

```
public constant
include socket.e
namespace sockets
```

MSG_TRYHARD

Signature:

```
MSG_TRYHARD
```

```
public constant
include socket.e
namespace sockets
```

MSG_WAITALL

This flag requests that the operation block until the full request is

Signature:

```
MSG_WAITALL
```

```
public constant
include socket.e
namespace sockets
```

OK

No error occurred.

Signature:

```
OK
```

```
public constant
include socket.e
namespace sockets
```

SCM_TIMESTAMP

Signature:

```
SCM_TIMESTAMP
```

```
public constant
include socket.e
namespace sockets
```

SCM_TIMESTAMPING

Signature:

```
SCM_TIMESTAMPING
```

```
public constant
include socket.e
namespace sockets
```

SCM_TIMESTAMPNS

Signature:

```
SCM_TIMESTAMPNS
```

```
public constant
include socket.e
namespace sockets
```

SD_BOTH

Shutdown both send and receive operations.

Signature:

```
SD_BOTH
```

```
public constant
include socket.e
namespace sockets
```

SD_RECEIVE

Shutdown the receive operations.

Signature:

```
SD_RECEIVE
```

```
public constant
include socket.e
namespace sockets
```

SD_SEND

Shutdown the send operations.

Signature:

```
SD_SEND
```

```
public constant
```

```
include socket.e
namespace sockets
```

SELECT_IS_ERROR

Boolean (1/0) value indicating the error state.

Signature:

```
SELECT_IS_ERROR
```

```
public enum
include socket.e
namespace sockets
```

SELECT_IS_READABLE

Boolean (1/0) value indicating the readability.

Signature:

```
SELECT_IS_READABLE
```

```
public enum
include socket.e
namespace sockets
```

SELECT_IS_WRITABLE

Boolean (1/0) value indicating the writeability.

Signature:

```
SELECT_IS_WRITABLE
```

```
public enum
include socket.e
namespace sockets
```

SELECT_SOCKET

The socket

Signature:

```
SELECT_SOCKET
```

```
public enum
include socket.e
namespace sockets
```

SOCKET_SOCKADDR_IN

Accessor index for the sockaddr_in pointer of a socket type

Signature:

```
SOCKET_SOCKADDR_IN
```

```
export enum
include socket.e
namespace sockets
```

SOCKET_SOCKET

Accessor index for socket handle of a socket type

Signature:

```
SOCKET_SOCKET
```

```
export enum
include socket.e
namespace sockets
```

SOCK_DGRAM

Supports datagrams (connectionless, unreliable messages of a

Signature:

```
SOCK_DGRAM
```

```
public constant
include socket.e
namespace sockets
```

SOCK_RAW

Provides raw network protocol access.

Signature:

```
SOCK_RAW
```

```
public constant
include socket.e
namespace sockets
```

SOCK_RDM

Provides a reliable datagram layer that does not guarantee ordering.

Signature:

```
SOCK_RDM
```

```
public constant
include socket.e
namespace sockets
```

SOCK_SEQPACKET

Obsolete and should not be used in new programs

Signature:

```
SOCK_SEQPACKET
```

```
public constant
include socket.e
namespace sockets
```

SOCK_STREAM

Provides sequenced, reliable, two-way, connection-based byte streams.

Signature:

```
SOCK_STREAM
```

```
public constant
include socket.e
namespace sockets
```

SOL_SOCKET

Signature:

```
SOL_SOCKET
```

```
public constant
include socket.e
namespace sockets
```

SO_ACCEPTCONN

Signature:

```
SO_ACCEPTCONN
```

```
public constant
include socket.e
namespace sockets
```

SO_ATTACH_FILTER

Signature:

```
SO_ATTACH_FILTER
```

```
public constant
include socket.e
namespace sockets
```

SO_BINDTODEVICE

=== LINUX Socket Filtering Options

Signature:

```
SO_BINDTODEVICE
```

```
public constant
include socket.e
```

SO_BROADCAST

Signature:

```
SO_BROADCAST
```

```
public constant
include socket.e
namespace sockets
```

SO_BSDCOMPAT

Signature:

```
SO_BSDCOMPAT
```

```
public constant
include socket.e
namespace sockets
```

SO_CONNDATA

Signature:

```
SO_CONNDATA
```

```
public constant
include socket.e
namespace sockets
```

SO_CONNDATALEN

Signature:

```
SO_CONNDATALEN
```

```
public constant
include socket.e
namespace sockets
```

SO_CONNOPT

Signature:

```
SO_CONNOPT
```

```
public constant
include socket.e
namespace sockets
```

SO_CONNOPTLEN

Signature:

```
SO_CONNOPTLEN
```

```
public constant
include socket.e
namespace sockets
```

SO_DEBUG

Signature:

```
SO_DEBUG
```

```
public constant
include socket.e
namespace sockets
```

SO_DETACH_FILTER

Signature:

```
SO_DETACH_FILTER
```

```
public constant
include socket.e
namespace sockets
```

SO_DISCDATA

Signature:

```
SO_DISCDATA
```

```
public constant
include socket.e
namespace sockets
```

SO_DISCDATALEN

Signature:

```
SO_DISCDATALEN
```

```
public constant
include socket.e
namespace sockets
```

SO_DISCOPT

Signature:

```
SO_DISCOPT
```

```
public constant
include socket.e
namespace sockets
```

SO_DISCOPTLEN

Signature:

```
SO_DISCOPTLEN
```

```
public constant
include socket.e
namespace sockets
```

SO_DOMAIN

Signature:

```
SO_DOMAIN
```

```
public constant
include socket.e
namespace sockets
```

SO_DONTLINGER

Signature:

```
SO_DONTLINGER
```

```
public constant
include socket.e
namespace sockets
```

SO_DONTROUTE

Signature:

```
SO_DONTROUTE
```

```
public constant
include socket.e
namespace sockets
```

SO_ERROR

Signature:

```
SO_ERROR
```

```
public constant
include socket.e
namespace sockets
```

SO_KEEPALIVE

Signature:

```
SO_KEEPALIVE
```

```
public constant
include socket.e
namespace sockets
```

SO_LINGER

Signature:

```
SO_LINGER
```

```
public constant
include socket.e
namespace sockets
```

SO_MARK

Signature:

```
SO_MARK
```

```
public constant
include socket.e
namespace sockets
```

SO_MAXDG

Signature:

```
SO_MAXDG
```

```
public constant
include socket.e
namespace sockets
```

SO_MAXPATHDG

Signature:

```
SO_MAXPATHDG
```

```
public constant
include socket.e
namespace sockets
```

SO_NO_CHECK

Signature:

```
SO_NO_CHECK
```

```
public constant
include socket.e
namespace sockets
```

SO_OOBINLINE

=== Windows Socket Options

Signature:

```
SO_OOBINLINE
```

```
public constant
include socket.e
namespace sockets
```

SO_OPENTYPE

Signature:

```
SO_OPENTYPE
```

```
public constant
include socket.e
namespace sockets
```

SO_PASSCRED

Signature:

```
SO_PASSCRED
```

```
public constant
include socket.e
namespace sockets
```

SO_PASSSEC

Signature:

```
SO_PASSSEC
```

```
public constant
include socket.e
namespace sockets
```

SO_PEERCRED

Signature:

```
SO_PEERCRED
```

```
public constant
include socket.e
```

```
namespace sockets
```

SO_PEERNAME

Signature:

```
SO_PEERNAME
```

```
public constant
include socket.e
namespace sockets
```

SO_PEERSEC

Signature:

```
SO_PEERSEC
```

```
public constant
include socket.e
namespace sockets
```

SO_PRIORITY

Signature:

```
SO_PRIORITY
```

```
public constant
include socket.e
namespace sockets
```

SO_PROTOCOL

Signature:

```
SO_PROTOCOL
```

```
public constant
include socket.e
namespace sockets
```

SO_RCVBUF

Signature:

```
SO_RCVBUF
```

```
public constant
include socket.e
namespace sockets
```

SO_RCVBUFFORCE

Signature:

```
SO_RCVBUFFORCE
```

```
public constant
include socket.e
namespace sockets
```

SO_RCVLOWAT

Signature:

```
SO_RCVLOWAT
```

```
public constant
include socket.e
namespace sockets
```

SO_RCVTIMEO

Signature:

```
SO_RCVTIMEO
```

```
public constant
include socket.e
namespace sockets
```

SO_REUSEADDR

Signature:

```
SO_REUSEADDR
```

```
public constant
include socket.e
namespace sockets
```

SO_REUSEPORT

Signature:

```
SO_REUSEPORT
```

```
public constant
include socket.e
namespace sockets
```

SO_RXQ_OVFL

Pass to the `flags` parameter of send and receive

Signature:

```
SO_RXQ_OVFL
```

```
public constant
include socket.e
namespace sockets
```

SO_SECURITY_AUTHENTICATION

Signature:

```
SO_SECURITY_AUTHENTICATION
```

```
public constant
include socket.e
namespace sockets
```

SO_SECURITY_ENCRYPTION_NETWORK

Signature:

```
SO_SECURITY_ENCRYPTION_NETWORK
```

```
public constant
include socket.e
namespace sockets
```

SO_SECURITY_ENCRYPTION_TRANSPORT

Signature:

```
SO_SECURITY_ENCRYPTION_TRANSPORT
```

```
public constant
include socket.e
namespace sockets
```

SO_SNDBUF

Signature:

```
SO_SNDBUF
```

```
public constant
include socket.e
namespace sockets
```

SO_SNDBUFFORCE

Signature:

```
SO_SNDBUFFORCE
```

```
public constant
include socket.e
namespace sockets
```

SO_SNDLOWAT

Signature:

```
SO_SNDLOWAT
```

```
public constant
include socket.e
namespace sockets
```

SO_SNDTIMEO

Signature:

```
SO_SNDTIMEO
```

```
public constant
include socket.e
namespace sockets
```

SO_SYNCHRONOUS_ALTERT

Signature:

```
SO_SYNCHRONOUS_ALTERT
```

```
public constant
include socket.e
namespace sockets
```

SO_SYNCHRONOUS_NONALERT

=== LINUX Socket Options

Signature:

```
SO_SYNCHRONOUS_NONALERT
```

```
public constant
include socket.e
namespace sockets
```

SO_TIMESTAMP

Signature:

```
SO_TIMESTAMP
```

```
public constant
include socket.e
namespace sockets
```

SO_TIMESTAMPING

Signature:

```
SO_TIMESTAMPING
```

```
public constant
include socket.e
namespace sockets
```

SO_TIMESTAMPNS

Signature:

```
SO_TIMESTAMPNS
```

```
public constant
include socket.e
namespace sockets
```

SO_TYPE

Signature:

```
SO_TYPE
```

```
public constant
include socket.e
namespace sockets
```

SO_USELOOPBACK

Signature:

```
SO_USELOOPBACK
```

```
public constant
include socket.e
namespace sockets
```

accept

Produces a new socket for an incoming connection.

Signature:

```
accept(socket sock)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `sock`: the server socket

Returns: An **atom**, on error
A **sequence**, `{socket client, sequence client_ip_address}` on success.

Comments: Using this function allows communication to occur on a "side channel" while the main server socket remains available for new connections.

`accept`() must be called after `bind`() and `listen`().

bind

Joins a socket to a specific local internet address and port so

Signature:

```
bind(socket sock, sequence address, integer port = - 1)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
- ≡ `sock` : the socket
- ≡ `address` : the address to bind the socket to
- ≡ `port` : optional, if not specified you must include :PORT in the address parameter.

Returns: An **integer**, 0 on success and -1 on failure.

Example 1:

```
 -- Bind to all interfaces on the default port 80.
 success = bind(socket, "0.0.0.0")
 -- Bind to all interfaces on port 8080.
 success = bind(socket, "0.0.0.0:8080")
 -- Bind only to the 243.17.33.19 interface on port 345.
 success = bind(socket, "243.17.33.19", 345)
```

close

Closes a socket.

Signature:

```
close(socket sock)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `sock`: the socket to close

Returns: An **integer**, 0 on success and -1 on error.

Comments: It may take several minutes for the OS to declare the socket as closed.

connect

Establish an outgoing connection to a remote computer. Only works with TCP sockets.

Signature:

```
connect(socket sock, sequence address, integer port = - 1)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
- ≡ `sock` : the socket
- ≡ `address` : ip address to connect, optionally with :PORT at the end
- ≡ `port` : port number

Returns: An **integer**, 0 for success and non-zero on failure. See the `ERR_*` constants for supported values.

Comments: `address` can contain a port number. If it does not, it has to be supplied to the `port` parameter.

Example 1:

```
success = connect(sock, "11.1.1.1") -- uses default port 80
success = connect(sock, "11.1.1.1:110") -- uses port 110
success = connect(sock, "11.1.1.1", 345) -- uses port 345
```

create

Create a new socket

Signature:

```
create(integer family, integer sock_type, integer protocol)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `family`: an integer
≡ `sock_type`: an integer, the type of socket to create
≡ `protocol`: an integer, the communication protocol being used

`family` options:
- AF_UNIX
- AF_INET
- AF_INET6
- AF_APPLETALK
- AF_BTH

`sock_type` options:
- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW
- SOCK_RDM
- SOCK_SEQPACKET

Returns: An **object**, an atom, representing an integer code on failure, else a sequence representing a valid socket id.

Example 1:

```
socket = create(AF_INET, SOCK_STREAM, 0)
```

error_code

Get the error code.

Signature:

```
error_code()
```

```
public function
include socket.e
namespace sockets
```

Returns: Integer OK on no error, otherwise any one of the `ERR_` constants to follow.

get_option

Get options for a socket.

Signature:

```
get_option(socket sock, integer level, integer optname)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `sock` : the socket
≡ `level` : an integer, the option level
≡ `optname` : requested option (See Socket Options)

Returns: An **object**, either:
• On error, {"ERROR",error_code}.
• On success, either an atom or a sequence containing the option value, depending on the option.

Comments: Primarily for use in multicast or more advanced socket applications. Level is the option level, and option_name is the option for which values are being sought. Level is usually SOL_SOCKET.

info

Get constant definitions from the backend.

Signature:

```
info(integer Type)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `type` : The type of information requested.

Returns: A **sequence**, containing the list of definitions from the backend. The resulting list can be indexed into using the Euphoria constants. Or an atom indicating an error.

See Also: Socket Options, Socket Backend Constants, Socket Type Euphoria Constants

Example 1:

```
object result = info(ESOCK_TYPE_AF)
-- result = { AF_UNIX, AF_INET, AF_INET6, AF_APPLETALK, AF_BTH, AF_UNSPEC }
```

listen

Start monitoring a connection. Only works with TCP sockets.

Signature:

```
listen(socket sock, integer backlog)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `sock` : the socket
≡ `backlog` : the number of connection requests that can be kept waiting before the OS refuses to hear any more.

Returns: An **integer**, 0 on success and an error code on failure.

Comments: Once the socket is created and bound, this will indicate to the operating system that you are ready to being listening for connections.

The value of `backlog` is strongly dependent on both the hardware and the amount of time it takes the program to process each connection request.

This function must be executed after bind().

receive

Receive data from a bound socket.

Signature:

```
receive(socket sock, atom flags = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `sock` : the socket to get data from
≡ `flags` : flags (see Send Flags)

Returns: A **sequence**, either a full string of data on success, or an atom indicating the error code.

Comments: This function will not return until data is actually received on the socket, unless the flags parameter contains MSG_DONTWAIT.

MSG_DONTWAIT only works on Linux kernels 2.4 and above. To be cross-platform you should use select to determine if a socket is readable, i.e. has data waiting.

receive_from

Receive a UDP packet from a given socket

Signature:

```
receive_from(socket sock, atom flags = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `sock`: the server socket
≡ `flags` : flags (see Send Flags)

Returns: A `sequence` containing { client_ip, client_port, data } or an `atom` error code.

See Also: send_to

select

Determine the read, write and error status of one or more sockets.

Signature:

```
select(object sockets_read, object sockets_write, object sockets_err,
integer timeout = 0, integer timeout_micro = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `sockets_read` : either one socket or a sequence of sockets to check for reading.
≡ `sockets_write` : either one socket or a sequence of sockets to check for writing.
≡ `sockets_err` : either one socket or a sequence of sockets to check for errors.
≡ `timeout` : maximum time to wait to determine a sockets status, seconds part
≡ `timeout_micro` : maximum time to wait to determine a sockets status, microsecond

part

Returns: A **sequence**, of the same size of all unique sockets containing { socket, read_status, write_status, error_status } for each socket passed 2 to the function. Note that the sockets returned are not guaranteed to be in any particular order.

send

Send TCP data to a socket connected remotely.

Signature:

```
send(socket sock, sequence data, atom flags = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `sock` : the socket to send data to
≡ `data` : a sequence of atoms, what to send
≡ `flags` : flags (see Send Flags)

Returns: An **integer**, the number of characters sent, or -1 for an error.

send_to

Send a UDP packet to a given socket

Signature:

```
send_to(socket sock, sequence data, sequence address, integer port = - 1,
atom flags = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `sock`: the server socket
≡ `data`: the data to be sent
≡ `ip`: the ip where the data is to be sent to (ip:port) is acceptable
≡ `port`: the port where the data is to be sent on (if not supplied with the ip)
≡ `flags` : flags (see Send Flags)

Returns: An `integer` status code.

See Also: receive_from

service_by_name

Get service information by name.

Signature:

```
service_by_name(sequence name, object protocol = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments:
≡ `name` : service name.
≡ `protocol` : protocol. Default is not to search by protocol.

Returns: A **sequence**, containing { official protocol name, protocol, port number } or an atom indicating the error code.

See Also: service_by_port

Example 1:

```
object result = getservbyname("http")
-- result = { "http", "tcp", 80 }
```

service_by_port

Get service information by port number.

Signature:

```
service_by_port(integer port, object protocol = 0)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `port` : port number.
≡ `protocol` : protocol. Default is not to search by protocol.

Returns: A **sequence**, containing { official protocol name, protocol, port number } or an atom indicating the error code.

See Also: service_by_name

Example 1:

```
object result = getservbyport(80)
-- result = { "http", "tcp", 80 }
```

set_option

Set options for a socket.

Signature:

```
set_option(socket sock, integer level, integer optname, object val)
```

```
public function
include socket.e
namespace sockets
```

Arguments: ≡ `sock` : an atom, the socket id
≡ `level` : an integer, the option level
≡ `optname` : requested option (See Socket Options)
≡ `val` : an object, the new value for the option

Returns: An **integer**, 0 on success, -1 on error.

Comments: Primarily for use in multicast or more advanced socket applications. Level is the option level, and option_name is the option for which values are being set. Level is usually SOL_SOCKET.

See Also: get_option

shutdown

Partially or fully close a socket.

Signature:

```
shutdown(socket sock, atom method = SD_BOTH)
```

```
public function
include socket.e
```

```
namespace sockets
```

Arguments: ≡ `sock` : the socket to shutdown
≡ `method` : the method used to close the socket

Returns: An **integer**, 0 on success and -1 on error.

Comments: Three constants are defined that can be sent to `method`:
• SD_SEND - shutdown the send operations.
• SD_RECEIVE - shutdown the receive operations.
• SD_BOTH - shutdown both send and receive operations.

It may take several minutes for the OS to declare the socket as closed.

socket

Socket type

Signature:

```
socket(object o)
```

```
public type
include socket.e
namespace sockets
```

sort

Constants

- ASCENDING
- NORMAL_ORDER
- DESCENDING
- REVERSE_ORDER

Routines

- sort
- custom_sort
- sort_columns
- merge
- insertion_sort

sort API

ASCENDING

ascending sort order, always the default.

Signature:

```
ASCENDING
```

```
public constant
include sort.e
namespace stdsort
```

DESCENDING

descending sort order, which is the reverse of `ASCENDING`.

Signature:

```
DESCENDING
```

```
public constant
include sort.e
namespace stdsort
```

NORMAL_ORDER

The normal sort order used by the custom comparison routine.

Signature:

```
NORMAL_ORDER
```

```
public constant
include sort.e
namespace stdsort
```

REVERSE_ORDER

Reverses the sense of the order returned by a custom comparison routine.

Signature:

```
REVERSE_ORDER
```

```
public constant
include sort.e
namespace stdsort
```

custom_sort

sorts the elements of a sequence according to a user-defined order.

Signature:

```
custom_sort(integer custom_compare, sequence x, object data = {},
integer order = NORMAL_ORDER)
```

```
public function
include sort.e
namespace stdsort
```

Arguments: ≡ `custom_compare` : an integer, the routine-id of the user defined routine that compares two items which appear in the sequence to sort.
≡ `x` : the sequence of items to be sorted.
≡ `data` : an object, either {} (no custom data, the default), an atom or a non-empty sequence.
≡ `order` : an integer, either `NORMAL_ORDER` (the default) or `REVERSE_ORDER`.

Returns: A **sequence**, a copy of the original sequence in sorted order

Comments:

• If some user data is being provided, that data must be either an atom or a sequence with at least one element. **NOTE** only the first element is passed to the user defined comparison routine, any other elements are just ignored. The user data is not used or inspected it in any way other than passing it to the user defined routine.

• The user defined routine must return an integer *comparison result*
…… ♦ a **negative** value if object A must appear before object B
…… ♦ a **positive** value if object B must appear before object A
…… ♦ 0 if the order does not matter > **NOTE:** The meaning of the value returned by the user-defined routine is reversed when `order = REVERSE_ORDER`. The default is `order = NORMAL_ORDER`, which sorts in order returned by the custom comparison routine. <

• When no user data is provided, the user defined routine must accept two objects (A, B) and return just the *comparison result*.

• When some user data is provided, the user defined routine must take three objects (A, B , data). It must return either...
…… ♦ an integer, which is a *comparison result*
…… ♦ a two-element sequence, in which the first element is a *comparison result* and the second element is the updated user data that is to be used for the next call to the user defined routine.

• The elements of `x` can be atoms or sequences. Each time that the sort needs to compare two items in the sequence, it calls the user-defined function to determine the order.

• This function uses the shell sort algorithm. This sort is not stable; elements that are considered equal might change position relative to each other.

See Also: compare, sort

Example 1:

```
constant students = {{"Anne",18},   {"Bob",21},
                     {"Chris",16},  {"Diane",23},
                     {"Eddy",17},   {"Freya",16},
                     {"George",20}, {"Heidi",20},
                     {"Ian",19}}
sequence sorted_byage
function byage(object a, object b)
 ----- If the ages are the same,
    -- compare the names otherwise just compare ages.
    if equal(a[2], b[2]) then
        return compare(upper(a[1]), upper(b[1]))
    end if
    return compare(a[2], b[2])
end function

sorted_byage = custom_sort( routine_id("byage"), students )
-- result is {{"Chris",16}, {"Freya",16},
--           {"Eddy",17},   {"Anne",18},
--           {"Ian",19},    {"George",20},
--           {"Heidi",20}, {"Bob",21},
--           {"Diane",23}}

sorted_byage =
         custom_sort( routine_id("byage"), students,, REVERSE_ORDER )
-- result is {{"Diane",23}, {"Bob",21},
--           {"Heidi",20}, {"George",20},
--           {"Ian",19},    {"Anne",18},
--           {"Eddy",17},   {"Freya",16},
--           {"Chris",16}}
--
```

Example 2:

```
constant students = {{"Anne","Baxter",18}, {"Bob","Palmer",21},
                     {"Chris","du Pont",16},{"Diane","Fry",23},
                     {"Eddy","Ammon",17},{"Freya","Brash",16},
                     {"George","Gungle",20},{"Heidi","Smith",20},
                     {"Ian","Sidebottom",19}}
sequence sorted
function colsort(object a, object b, sequence cols)
    integer sign
    for i = 1 to length(cols) do
        if cols[i] < 0 then
            sign = -1
            cols[i] = -cols[i]
        else
            sign = 1
        end if
        if not equal(a[cols[i]], b[cols[i]]) then
          return sign * compare(upper(a[cols[i]]), upper(b[cols[i]]))
        end if
    end for

    return 0
end function

-- Order is age:descending, Surname, Given Name
sequence column_order = {-3,2,1}
sorted = custom_sort(routine_id("colsort"), students, {column_order})
-- result is
{
    {"Diane","Fry",23},
    {"Bob","Palmer",21},
    {"George","Gungle",20},
    {"Heidi","Smith",20},
    {"Ian","Sidebottom",19},
    {"Anne", "Baxter", 18 },
    {"Eddy","Ammon",17},
    {"Freya","Brash",16},
    {"Chris","du Pont",16}
}

sorted =
      custom_sort( routine_id("colsort"), students,
                   {column_order}, REVERSE_ORDER )
-- result is
{
    {"Chris","du Pont",16},
    {"Freya","Brash",16},
    {"Eddy","Ammon",17},
    {"Anne", "Baxter", 18 },
    {"Ian","Sidebottom",19},
    {"Heidi","Smith",20},
    {"George","Gungle",20},
    {"Bob","Palmer",21},
    {"Diane","Fry",23}
}
```

insertion_sort

sorts a sequence, and optionally another object together.

Signature:

```
insertion_sort(sequence s, object e = "", integer compfunc = - 1,
object userdata = "")
```

```
public function
include sort.e
namespace stdsort
```

Arguments: ≡ `s` : a sequence, holding data to be sorted.
≡ `e` : an object. If this is an atom, it is sorted in with `s`. If this is a non-empty sequence then `s` and `e` are both sorted independantly using this `insertion_sort` function and then the results are merged and returned.

≡ `compfunc` : an integer, either -1 or the routine id of a user-defined comparision function.

Returns: A **sequence**, consisting of `s` and `e` sorted together.

Comments:

• This routine is usually a lot faster than the standard sort when `s` and `e` are partially sorted before calling the function. For example, you can use this routine to quickly add to a sorted list.
• The input sequences do not have to be the same size.
• The user-defined comparision function must accept two objects and return an integer. It returns -1 if the first object must appear before the second one, and 1 if the first object must after before the second one, and 0 if the order does not matter.

See Also: compare, sort, merge

Example 1:

```
sequence X = {}
while true do
   newdata = get_data()
   if compare(-1, newdata) then
      exit
   end if
   X = insertion_sort(X, newdata)
   process(new_data)
end while
```

merge

Merge two pre-sorted sequences into a single sequence.

Signature:

```
merge(sequence a, sequence b, integer compfunc = - 1,
object userdata = "")
```

```
public function
include sort.e
namespace stdsort
```

Arguments: ≡ `a` : a sequence, holding pre-sorted data.
≡ `b` : a sequence, holding pre-sorted data.
≡ `compfunc` : an integer, either -1 or the routine id of a user-defined comparision function.

Returns: A **sequence**, consisting of `a` and `b` merged together.

Comments:

• If `a` or `b` is not already sorted, the resulting sequence might not be sorted either.
• The input sequences do not have to be the same size.
• The user-defined comparision function must accept two objects and return an integer. It returns -1 if the first object must appear before the second one, and 1 if the first object must after before the second one, and 0 if the order does not matter.

See Also: compare, sort

Example 1:

```
sequence X,Y
X = sort( {5,3,7,1,9,0} ) --> {0,1,3,5,7,9}
Y = sort( {6,8,10,2} ) --> {2,6,8,10}
? merge(X,Y) --> {0,1,2,3,5,6,7,8,9,10}
```

sort

sorts the elements of a sequence into ascending order.

Signature:

```
sort(sequence x, integer order = ASCENDING)
```

```
public function
include sort.e
namespace stdsort
```

Arguments:
≡ `x` : The sequence to be sorted.
≡ `order` : the sort order. Default is `ASCENDING`.

Returns: A **sequence**, a copy of the original sequence in ascending order

Comments: The elements can be atoms or sequences.

The standard `compare()` routine is used to compare elements. This means that "`y` is greater than `x`" is defined by `compare(y, x)=1`.

This function uses the shell sort algorithm. This sort is not stable; elements that are considered equal might change position relative to each other.

See Also: compare, custom_sort

Example 1:

```
constant student_ages = {18,21,16,23,17,16,20,20,19}
sequence sorted_ages
sorted_ages = sort( student_ages )
-- result is {16,16,17,18,19,20,20,21,23}
```

sort_columns

sorts the rows in a sequence according to a user-defined

Signature:

```
sort_columns(sequence x, sequence column_list)
```

```
public function
include sort.e
namespace stdsort
```

Arguments:
≡ `x` : a sequence, holding the sequences to be sorted.
≡ `column_list` : a list of columns indexes `x` is to be sorted by.

Returns: A **sequence**, a copy of the original sequence in sorted order.

Comments: `x` must be a sequence of sequences.

A non-existent column is treated as coming before an existing column. This allows sorting of records that are shorter than the columns in the column list.

By default, columns are sorted in ascending order. To sort in descending order, make the column number negative.

This function uses the shell sort algorithm. This sort is not stable; elements that are considered equal might change position relative to each other.

See Also: compare, sort

Example 1:

```
sequence dirlist
dirlist = dir("c:\\temp")
sequence sorted
-- Order is Size:descending, Name
sorted = sort_columns( dirlist, {-D_SIZE, D_NAME} )
```

stack

stack API

FIFO

FIFO : first item in is first item out (like people standing in line).

Signature:

```
FIFO
```

```
public constant
include stack.e
namespace stack
```

FILO

FILO: first item in is last item out (like a stack of plates).

Signature:

```
FILO
```

```
public constant
include stack.e
namespace stack
```

at

fetches a value from the stack without removing it from the stack.

Signature:

```
at(stack sk, integer idx = 1)


public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack being queried
≡ `idx` : an integer, the place to inspect. The default is 1 (top item).

Returns: An **object**, the `idx`-th item of the stack.

Comments:

• For `FIFO` stacks (queues), the top item is the oldest item in the stack.
• For `FILO` stacks, the top item is the newest item in the stack.

`idx` can be less than 1, in which case it refers relative to the end item. Thus, 0 stands for the end element.

See Also: size, top, peek_top, peek_end

Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)

at(sk, 0)  --> 5
at(sk, -1) --> "abc"
at(sk, 1)  --> 2.3
at(sk, 2)  --> "abc"
```

Example 2:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
at(sk, 0)  --> 2.3
at(sk, -1) --> "abc"
at(sk, 1)  --> 5
at(sk, 2)  --> "abc"
```

clear

wipes out a stack.

Signature:

```
clear(stack sk)


public procedure
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to clear.

See Also: new, is_empty

dup

repeats the top element of a stack.

Signature:

```
dup(stack sk)
```

```
public procedure
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack.

Comments:

- For `FIFO` stacks (queues), the top item is the oldest item in the stack.
- For `FILO` stacks, the top item is the newest item in the stack.

Example 1:

```
 stack sk = new(FILO)

 push(sk,5)
 push(sk,"abc")
 push(sk, "")

 dup(sk)

 peek_top(sk,1) --> ""
 peek_top(sk,2) --> "abc"
 size(sk)       --> 3

 dup(sk)

 peek_top(sk,1) --> ""
 peek_top(sk,2) --> ""
 peek_top(sk,3) --> "abc"
 size(sk)       --> 4
```

Example 1:

```
 stack sk = new(FIFO)

 push(sk, 5)
 push(sk, "abc")
 push(sk, "")

 dup(sk)

 peek_top(sk, 1) --> 5
 peek_top(sk, 2) --> "abc"
 size(sk)        --> 3

 dup(sk)

 peek_top(sk, 1) --> 5
 peek_top(sk, 2) --> 5
 peek_top(sk, 3) --> "abc"
 size(sk)        --> 4
```

is_empty

tests if a stack is empty.

Signature:

```
is_empty(stack sk)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack being queried.

Returns: An **integer**, 1 if the stack is empty, else 0.

See Also: size

last

retrieves the end element on a stack.

Signature:

```
last(stack sk)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to inspect.

Returns: An **object**, the end element on a stack.

Comments: This call is equivalent to `at(sk,0)`.

See Also: at, pop, peek_end, top

Example 1:

```
 stack sk = new(FILO)

 push(sk,5)
 push(sk,"abc")
 push(sk, 2.3)

 last(sk) --> 5
```

Example 2:

```
 stack sk = new(FIFO)

 push(sk,5)
 push(sk,"abc")
 push(sk, 2.3)

 last(sk) --> 2.3
```

new

creates a new stack.

Signature:

```
new(integer typ = FILO)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `stack_type` : an integer, defining the semantics of the stack. The default is FILO.

Returns: An empty **stack**, note that the variable storing the stack must not be an integer. The resources allocated for the stack will be automatically cleaned up if the reference count of the returned value drops to zero, or if passed in a call to delete.

Comments: There are two sorts of stacks, designated by the types `FIFO` and `FILO`:
• A `FIFO` stack is one where the first item to be pushed is popped first. People standing in queue form a `FIFO` stack.
• A `FILO` stack is one where the item pushed last is popped first. A column of coins is of the `FILO` kind.

See Also: is_empty

peek_end

gets an element, relative to the end, from a stack.

Signature:

```
peek_end(stack sk, integer idx = 1)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to get from.
≡ `idx` : integer. The n-th item from the end to get from the stack. The default is 1.

Returns: An **item**, from the stack, which is **not** removed from the stack.

Comments:

• For `FIFO` stacks (queues), the end item is the newest item in the stack.
• For `FILO` stacks, the end item is the oldest item in the stack.

When `idx` is omitted the 'end' of the stack is returned. When `idx` is supplied, it represents the N-th item from the end to be returned. Thus an `idx` of 2 returns the 2nd item from the end, a value of 3 returns the 3rd item from the end, etc ...

See Also: pop, top, is_empty, size, peek_top

Example 1:

```
stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_end(sk) -- 3
? peek_end(sk,2) -- 2
? peek_end(sk,3) -- 1
? peek_end(sk,4) -- *error*
? peek_end(sk, size(sk)) -- 3 (top item)
```

Example 2:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_end(sk) -- 1
? peek_end(sk,2) -- 2
? peek_end(sk,3) -- 3
? peek_end(sk,4) -- *error*
? peek_end(sk, size(sk)) -- 3 (top item)
```

peek_top

gets an element, relative to the top, from a stack.

Signature:

```
peek_top(stack sk, integer idx = 1)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to get from.
≡ `idx` : integer. The n-th item to get from the stack. The default is 1.

Returns: An **item**, from the stack, which is **not** removed from the stack.

Comments: This is identical to pop except that it does not remove the item.

• For `FIFO` stacks (queues), the top item is the oldest item in the stack.
• For `FILO` stacks, the top item is the newest item in the stack.

When `idx` is omitted the 'top' of the stack is returned. When `idx` is supplied, it represents the N-th item from the top to be returned. Thus an `idx` of 2 returns the 2nd item from the top, a value of 3 returns the 3rd item from the top, etc ...

See Also: pop, top, is_empty, size, peek_end

Example 1:

```
stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_top(sk) -- 1
? peek_top(sk,2) -- 2
? peek_top(sk,3) -- 3
? peek_top(sk,4) -- *error*
? peek_top(sk, size(sk)) -- 3 (end item)
```

Example 2:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? peek_top(sk) -- 3
? peek_top(sk,2) -- 2
? peek_top(sk,3) -- 1
? peek_top(sk,4) -- *error*
? peek_top(sk, size(sk)) -- 1 (end item)
```

pop

removes an element from a stack.

Signature:

```
pop(stack sk, integer idx = 1)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to pop
≡ `idx` : integer. The n-th item to pick from the stack. The default is 1.

Returns: An **item**, from the stack, which is also removed from the stack.

Comments:

• For `FIFO` stacks (queues), the top item is the oldest item in the stack.
• For `FILO` stacks, the top item is the newest item in the stack.

When `idx` is omitted the 'top' of the stack is removed and returned. When `idx` is supplied, it represents the N-th item from the top to be removed and returned. Thus an `idx` of 2 returns the 2nd item from the top, a value of 3 returns the 3rd item from the top, etc ...

See Also: push, top, is_empty

Example 1:

```
stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? size(sk) -- 3
? pop(sk) -- 1
? size(sk) -- 2
? pop(sk) -- 2
? size(sk) -- 1
? pop(sk) -- 3
? size(sk) -- 0
? pop(sk) -- *error*
```

Example 2:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
? size(sk) -- 3
? pop(sk) -- 3
? size(sk) -- 2
? pop(sk) -- 2
? size(sk) -- 1
? pop(sk) -- 1
? size(sk) -- 0
? pop(sk) -- *error*
```

Example 3:

```
stack sk = new(FILO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
push(sk, 4)
-- stack contains {1,2,3,4} (oldest to newest)
? size(sk) -- 4
? pop(sk, 2) -- Pluck out the 2nd newest item .. 3
? size(sk) -- 3
-- stack now contains {1,2,4}
```

Example 4:

```
stack sk = new(FIFO)
push(sk, 1)
push(sk, 2)
push(sk, 3)
push(sk, 4)
-- stack contains {1,2,3,4} (oldest to newest)
? size(sk) -- 4
? pop(sk, 2) -- Pluck out the 2nd oldest item .. 2
? size(sk) -- 3
-- stack now contains {1,3,4}
```

push

adds an element to a stack.

Signature:

```
push(stack sk, object value)
```

```
public procedure
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to augment
≡ `value` : an object, the value to push.

Comments: `value` appears at the end of `FIFO` stacks and the top of `FILO` stacks. The size of the stack increases by one.

See Also: pop, top

Example 1:

```
stack sk = new(FIFO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)
top(sk)  --> 5
last(sk) --> 2.3
```

Example 2:

```
stack sk = new(FILO)

push(sk,5)
push(sk,"abc")
push(sk, 2.3)
top(sk)  --> 2.3
last(sk) --> 5
```

set

updates a value on the stack.

Signature:

```
set(stack sk, object val, integer idx = 1)
```

```
public procedure
include stack.e
namespace stack
```

Arguments:
≡ `sk` : the stack being queried
≡ `val` : an object, the value to place on the stack
≡ `idx` : an integer, the place to inspect. The default is 1 (the top item)

Comments:

• For `FIFO` stacks (queues), the top item is the oldest item in the stack.
• For `FILO` stacks, the top item is the newest item in the stack.

`idx` can be less than 1, in which case it refers to an element relative to the end of the stack. Thus, 0 stands for the end element.

See Also: size, top

size

returns the number elements in a stack.

Signature:

```
size(stack sk)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack being queried.

Returns: An **integer**, the number of elements in `sk`.

stack

A stack is a sequence of objects with some internal data.

Signature:

```
stack(object obj_p)
```

```
public type
include stack.e
```

```
namespace stack
```

swap

swaps the top two elements of a stack.

Signature:

```
swap(stack sk)
```

```
public procedure
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to swap.

Returns: A **copy**, of the original **stack**, with the top two elements swapped.

Comments:

• For `FIFO` stacks (queues), the top item is the oldest item in the stack.
• For `FILO` stacks, the top item is the newest item in the stack.

Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
push(sk, "")

? peek_top(sk, 1) --> ""
? peek_top(sk, 2) --> 2.3

swap(sk)

? peek_top(sk, 1) --> 2.3
? peek_top(sk, 2) --> ""
```

Example 2:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)
push(sk, "")

peek_top(sk, 1) --> 5
peek_top(sk, 2) --> "abc"

swap(sk)

peek_top(sk, 1) --> "abc"
peek_top(sk, 2) --> 5
```

top

retrieves the top element on a stack.

Signature:

```
top(stack sk)
```

```
public function
include stack.e
namespace stack
```

Arguments: ≡ `sk` : the stack to inspect.

Returns: An **object**, the top element on a stack.

Comments: This call is equivalent to `at(sk,1)`.

See Also: at, pop, peek_top, last

Example 1:

```
stack sk = new(FILO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)

top(sk) --> 2.3
```

Example 1:

```
stack sk = new(FIFO)

push(sk, 5)
push(sk, "abc")
push(sk, 2.3)

top(sk) --> 5
```

---

stats

Routines

- small
- largest
- smallest
- range
- ST_FULLPOP
- ST_SAMPLE
- ST_ALLNUM
- ST_IGNSTR
- ST_ZEROSTR
- stdev
- avedev
- sum
- count
- average
- geomean
- harmean
- movavg
- emovavg
- median
- raw_frequency
- mode
- central_moment
- sum_central_moments
- skewness
- kurtosis

---

stats API

ST_ALLNUM

The supplied data consists of only atoms.

Signature:

```
ST_ALLNUM
```

```
public enum
include stats.e
namespace stats
```

ST_FULLPOP

The supplied data is the entire population.

Signature:

```
ST_FULLPOP
```

```
public enum
include stats.e
namespace stats
```

ST_IGNSTR

Any sub-sequences (eg. strings) in the supplied data are ignored.

Signature:

```
ST_IGNSTR
```

```
public enum
include stats.e
namespace stats
```

ST_SAMPLE

The supplied data is only a random sample of the population.

Signature:

```
ST_SAMPLE
```

```
public enum
include stats.e
namespace stats
```

ST_ZEROSTR

Any sub-sequences (eg. strings) in the supplied data are assumed to

Signature:

```
ST_ZEROSTR
```

```
public enum
include stats.e
namespace stats
```

avedev

returns the average of the absolute deviations of data points from their mean.

Signature:

```
avedev(sequence data_set, object subseq_opt = ST_ALLNUM,
integer population_type = ST_SAMPLE)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the mean of the absolute deviations.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.
≡ `population_type` : an integer. ST_SAMPLE (the default) assumes that `data_set` is a random sample of the total population. ST_FULLPOP means that `data_set` is the entire population.

Returns: An **atom** , the deviation from the mean.
An empty **sequence**, means that there is no meaningful data to calculate from.

Comments: `avedev`() is a measure of the variability in a data set. Its statistical properties are less well behaved than those of the standard deviation, which is why it is used less.

The numbers in `data_set` can either be the entire population of values or just a random subset. You indicate which in the `population_type` parameter. By default `data_set` represents a sample and not the entire population. When using this function with sample data, the result is an *estimated* deviation.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

The equation for absolute average deviation is: avedev(X) ==> SUM(ABS(X{1..N} - MEAN(X))) / N

Example 1:

```
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,7} )
    --> Ans: 1.966666667
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,7},, ST_FULLPOP )
    --> Ans: 1.84375
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR  )
    --> Ans: 1.99047619
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR,ST_FULLPOP )
    --> Ans: 1.857777778
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, 0 )
     --> Ans: 2.225
 ? avedev( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, 0, ST_FULLPOP )
    --> Ans: 2.0859375
```

average

returns the average (mean) of the data points.

Signature:

```
average(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : A list of 1 or more numbers for which you want the mean.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **object**,
• `{}` (the empty sequence) if there are no atoms in the set.
• an atom (the mean) if there are one or more atoms in the set.

Comments: `average`() is the theoretical probable value of a randomly selected item from the set.

Example 1:

```
? average( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR )
 Ans: 5.13333333
```

central_moment

returns the distance between a supplied value and the mean, to some supplied

Signature:

```
central_moment(sequence data_set, object datum, integer order_mag = 1,
object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers whose mean is used.
≡ `datum`: either a single value or a list of values for which you require the central moments.
≡ `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **object**. The same data type as `datum`. This is the set of calculated central moments.

Comments: For each of the items in `datum`, its central moment is calculated as ... CM = power(ITEM - AVG, MAGNITUDE)

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
central_moment("the cat is the hatter", "the",1)
               --> {23.14285714, 11.14285714, 8.142857143}
```

```
 central_moment("the cat is the hatter", 't',2)
                 -->   535.5918367
 central_moment("the cat is the hatter", 't',3)
                 -->   12395.12536
```

count

returns the count of all the atoms in an object.

Signature:

```
count(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : either an atom or a list.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **integer**, the number of atoms in the set. When `data_set` is an atom, 1 is returned.

Comments: This returns the number of numbers in `data_set`

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
 ? count( {7,2,8.5,6,6,-4.8,6,6,3.341,-8,"text"} ) -- Ans: 10
 ? count( {"cat", "dog", "lamb", "cow", "rabbit"} ) -- Ans: 0 (no atoms)
 ? count( 5 ) -- Ans: 1
```

emovavg

returns the exponential moving average of a set of data points.

Signature:

```
emovavg(object data_set, atom smoothing_factor)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want a moving average.
≡ `smoothing_factor` : an atom, the smoothing factor, typically between 0 and 1.

Returns: A **sequence**, made of the requested averages, or `{}` if `data_set` is empty or the supplied period is less than one.

Comments: A moving average is used to smooth out a set of data points over a period.

The formula used is:
: $Y_i = Y_{i-1} + F * (X_i - Y_{i-1})$

Note that only atom elements are included and any sub-sequences elements are ignored.

The smoothing factor controls how data is smoothed. 0 smooths everything to 0, and 1 means no smoothing at all.

Any value for `smoothing_factor` outside the 0.0..1.0 range causes `smoothing_factor` to be set to the periodic factor `(2/(N+1))`.

Example 1:

```
 ? emovavg( {7,2,8,5,6}, 0.75 )
  -- Ans: {6.65,3.1625,6.790625,5.44765625,5.861914063}
 ? emovavg( {7,2,8,5,6}, 0.25 )
  -- Ans: {5.95,4.9625,5.721875,5.54140625,5.656054687}
 ? emovavg( {7,2,8,5,6}, -1 )
  -- Ans: {6.066666667,4.711111111,5.807407407,5.538271605,5.69218107}
```

geomean

returns the geometric mean of the atoms in a sequence.

Signature:

```
geomean(object data_set, object subseq_opt = ST_ALLNUM)


public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : the values to take the geometric mean of.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **atom**, the geometric mean of the atoms in `data_set`. If there is no atom to take the mean of, 1 is returned.

Comments: The geometric mean of `N` atoms is the N-th root of their product. Signs are ignored.

This is useful to compute average growth rates.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

See Also: average

Example 1:

```
 ? geomean({3, "abc", -2, 6}, ST_IGNSTR)
         --> prints out power(36,1/3) = 3,30192724889462669
 ? geomean({1,2,3,4,5,6,7,8,9,10})
         --> = 4.528728688
```

harmean

returns the harmonic mean of the atoms in a sequence.

Signature:

```
harmean(sequence data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : the values to take the harmonic mean of.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **atom**, the harmonic mean of the atoms in `data_set`.

Comments: The harmonic mean is the inverse of the average of their inverses.

This is useful in engineering to compute equivalent capacities and resistances.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

See Also: average

Example 1:

```
? harmean({3, "abc", -2, 6}, ST_IGNSTR) -- =  0.
? harmean({{2, 3, 4}) -- 3 / (1/2 + 1/3 + 1/4) = 2.769230769
```

kurtosis

returns a measure of the spread of values in a dataset when compared to a

Signature:

```
kurtosis(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers whose kurtosis is required.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **object**. If this is an atom it is the kurtosis measure of the data set. Othewise it is a sequence containing an error integer. The return value {0} indicates that an empty dataset was passed, {1} indicates that the standard deviation is zero (all values are the same).

Comments: Generally speaking, a negative return indicates that most of the values are further from the mean, while positive values indicate that most values are nearer to the mean.

The larger the magnitude of the returned value, the more the data is 'peaked' or 'flatter' in that direction.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
kurtosis("thecatisthehatter")     --> -1.737889192
```

largest

returns the largest of the data points that are atoms.

Signature:

```
largest(object data_set)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers among which you want the largest.

Returns: An **object**, either of:

- an atom (the largest value) if there is at least one atom item in the set
- `{}` if there *is* no largest value.

Comments: Any `data_set` element which is not an atom is ignored.

Example 1:

```
largest( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"} ) -- Ans: 8
largest( {"just","text"} ) -- Ans: {}
```

median

returns the mid point of the data points.

Signature:

```
median(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the mean.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **object**, either `{}` if there are no items in the set, or an **atom** (the median) otherwise.

Comments: `median`() is the item for which half the items are below it and half are above it.

All elements are included; any sequence elements are assumed to have the value zero.

Example 1:

```
? median( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: 5
```

mode

returns the most frequent point or points of the data set.

Signature:

```
mode(sequence data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the mode.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: A **sequence**. The list of modal items in the data set.

Comments: It is possible for the `mode`() to return more than one item when more than one item in the set has the same highest frequency count.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
 mode( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: {6}
 mode( {8,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: {8,6}
```

movavg

returns the average (mean) of the data points for overlaping periods. This

Signature:

```
movavg(object data_set, object period_delta)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want a moving average.
≡ `period_delta` : an object, either
• an integer representing the size of the period, or
• a list of weightings to apply to the respective period positions.

Returns: A **sequence**, either the requested averages or `{}` if the Data sequence is empty or the supplied period is less than one.

If a list of weights was supplied, the result is a weighted average; otherwise, it is a simple average.

Comments: A moving average is used to smooth out a set of data points over a period.
For example, given a period of 5: # the first returned element is the average of the first five data points [1..5], # the second returned element is the average of the second five data points [2..6],
and so on
until the last returned value is the average of the last 5 data points [$-4 .. $].

When `period_delta` is an atom, it is rounded down to the width of the average. When it is a sequence, the width is its length. If there are not enough data points, zeroes are inserted.

Note that only atom elements are included and any sub-sequence elements are ignored.

Example 1:

```
 ? movavg( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8}, 10 )
  -- Ans: {5.8, 5.4, 5.5, 5.1, 4.7, 4.9}
 ? movavg( {7,2,8,5,6}, 2 )
  -- Ans: {4.5, 5, 6.5, 5.5}
 ? movavg( {7,2,8,5,6}, {0.5, 1.5} )
  -- Ans: {3.25, 6.5, 5.75, 5.75}
```

range

determines a number of *range* statistics for the data set.

Signature:

```
range(object data_set)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the range data.

Returns: A **sequence**, empty if no atoms were found, else like {Lowest, Highest, Range, Mid-range}

Comments: Any sequence element in `data_set` is ignored.

Example 1:

```
 ? range( {7,2,8,5,6,6,4,8,6,16,3,3,4,1,8,"text"} ) -- Ans: {1, 16, 15, 8.5}
```

raw_frequency

returns the frequency of each unique item in the data set.

Signature:

```
raw_frequency(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the frequencies.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: A **sequence**. This will contain zero or more 2-element sub-sequences. The first element is the frequency count and the second element is the data item that was counted. The returned values are in descending order, meaning that the highest frequencies are at the beginning of the returned list.

Comments: If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
? raw_frequency("the cat is the hatter")
```

This returns

```
{
{5,116},
{4,32},
{3,104},
{3,101},
{2,97},
{1,115},
{1,114},
{1,105},
{1,99}
}
```

skewness

returns a measure of the asymmetry of a data set.

Signature:

```
skewness(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers whose mean is used.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **atom**. The skewness measure of the data set.

Comments: Usually the data_set is a probablity distribution but it can be anything. This value is used to assess how suitable the data set is in representing the required analysis. It can help detect if there are too many extreme values in the data set.

Generally speaking, a negative return indicates that most of the values are lower than the mean, while positive values indicate that most values are greater than the mean. However this might not be the case when there are a few extreme values on one side of the mean.

The larger the magnitude of the returned value, the more the data is skewed in that direction.

A returned value of zero indicates that the mean and median values are identical and that the data is symmetrical.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
skewness("the cat is the hatter") --> -1.36166186
skewness("thecatisthehatter")     --> 0.1093730315
```

small

returns the index for the k-th smallest value from the supplied set of numbers.

Signature:

```
small(sequence data_set, integer ordinal_idx)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : The list of values from which the smallest value is chosen.
≡ `ordinal_idx` : The relative index of the desired smallest value.

Returns: A **sequence**, {The k-th smallest value, its index in the set}

Comments: `small()` is used to return a value based on its size relative to all the other elements in the sequence. When `index` is 1, the smallest index is returned. Use `index = length(data_set)` to return the highest.

If `ordinal_idx` is less than one, or greater then length of `data_set`, an empty sequence is returned.

The set of values does not have to be in any particular order. The values may be any Euphoria object.

Example 1:

```
small( {4,5,6,8,5,4,3,"text"}, 3 )
--> Ans: {4,1} (The 3rd smallest value)
small( {4,5,6,8,5,4,3,"text"}, 1 )
--> Ans: {3,7} (The 1st smallest value)
small( {4,5,6,8,5,4,3,"text"}, 7 )
--> Ans: {8,4} (The 7th smallest value)
small( {"def", "qwe", "abc", "try"}, 2 )
--> Ans: {"def", 1} (The 2nd smallest value)
small( {1,2,3,4}, -1)
--> Ans: {} -- no-value
small( {1,2,3,4}, 10)
--> Ans: {} -- no-value
```

smallest

returns the smallest of the data points.

Signature:

```
smallest(object data_set)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : A list of 1 or more numbers for which you want the smallest. **Note:** only atom elements are included and any sub-sequences elements are ignored.

Returns: An **object**, either of:
• an atom (the smallest value) if there is at least one atom item in the set

• `{}` if there *is* no largest value.

Comments: Any `data_set` element which is not an atom is ignored.

Example 1:

```
 ? smallest( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"} ) -- Ans: 1
 ? smallest( {"just","text"} ) -- Ans: {}
```

stdev

returns the standard deviation based of the population.

Signature:

```
stdev(sequence data_set, object subseq_opt = ST_ALLNUM,
integer population_type = ST_SAMPLE)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers for which you want the estimated standard deviation.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.
≡ `population_type` : an integer. ST_SAMPLE (the default) assumes that `data_set` is a random sample of the total population. ST_FULLPOP means that `data_set` is the entire population.

Returns: An **atom**, the estimated standard deviation. An empty **sequence** means that there is no meaningful data to calculate from.

Comments: `stdev`() is a measure of how values are different from the average.

The numbers in `data_set` can either be the entire population of values or just a random subset. You indicate which in the `population_type` parameter. By default `data_set` represents a sample and not the entire population. When using this function with sample data, the result is an *estimated* standard deviation.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
 ? stdev( {4,5,6,7,5,4,3,7} )                              -- Ans: 1.457737974
 ? stdev( {4,5,6,7,5,4,3,7} ,, ST_FULLPOP)                 -- Ans: 1.363589014
 ? stdev( {4,5,6,7,5,4,3,"text"} , ST_IGNSTR)              -- Ans: 1.345185418
 ? stdev( {4,5,6,7,5,4,3,"text"}, ST_IGNSTR, ST_FULLPOP )  -- Ans: 1.245399698
 ? stdev( {4,5,6,7,5,4,3,"text"} , 0)                      -- Ans: 2.121320344
 ? stdev( {4,5,6,7,5,4,3,"text"}, 0, ST_FULLPOP )          -- Ans: 1.984313483
```

sum

returns the sum of all the atoms in an object.

Signature:

```
sum(object data_set, object subseq_opt = ST_ALLNUM)
```

```
public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : Either an atom or a list of numbers to sum.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **atom**, the sum of the set.

Comments: `sum`() is used as a measure of the magnitude of a sequence of positive values.

If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

The equation is:

sum(X) ==> SUM(X{1..N})

Example 1:

```
? sum( {7,2,8.5,6,6,-4.8,6,6,3.341,-8,"text"}, 0 ) -- Ans: 32.041
```

sum_central_moments

returns sum of the central moments of each item in a data set.

Signature:

```
sum_central_moments(object data_set, integer order_mag = 1,
object subseq_opt = ST_ALLNUM)


public function
include stats.e
namespace stats
```

Arguments: ≡ `data_set` : a list of 1 or more numbers whose mean is used.
≡ `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
≡ `subseq_opt` : an object. When this is ST_ALLNUM (the default) it means that `data_set` is assumed to contain no sub-sequences otherwise this gives instructions about how to treat sub-sequences. See comments for details.

Returns: An **atom**. The total of the central moments calculated for each of the items in `data_set`.

Comments: If the data can contain sub-sequences, such as strings, you need to let the the function know about this otherwise it assumes every value in `data_set` is an number. If that is not the case then the function will crash. So it is important that if it can possibly contain sub-sequences that you tell this function what to do with them. Your choices are to ignore them or assume they have the value zero. To ignore them, use ST_IGNSTR as the `subseq_opt` parameter value otherwise use ST_ZEROSTR. However, if you know that `data_set` only contains numbers use the default `subseq_opt` value, ST_ALLNUM. **Note** It is faster if the data only contains numbers.

Example 1:

```
sum_central_moments("the cat is the hatter", 1) --> -8.526512829e-14
sum_central_moments("the cat is the hatter", 2) --> 19220.57143
sum_central_moments("the cat is the hatter", 3) --> -811341.551
sum_central_moments("the cat is the hatter", 4) --> 56824083.71
```

task

General Notes

For a complete overview of the task system, please see the mini-guide Multitasking in Euphoria.

task API

task_clock_start

restarts the clock used for scheduling real-time tasks.

Signature:

```
task_clock_start()
```

```
<built-in> procedure
```

Comments: Call this routine, some time after calling task_clock_stop(), when you want scheduling of real-time tasks to continue.

task_clock_stop() and `task_clock_start`() can be used to freeze the scheduling of real-time tasks.

`task_clock_start`() causes the scheduled times of all real-time tasks to be incremented by the amount of time since task_clock_stop() was called. This allows a game, simulation, or other program to continue smoothly.

Time-shared tasks are not affected.

See Also: task_clock_stop, task_schedule, task_yield, task_suspend, task_delay

Example 1:

```
-- freeze the game while the player answers the phone
task_clock_stop()
while get_key() = -1 do
end while
task_clock_start()
```

task_clock_stop

stops the scheduling of real-time tasks.

Signature:

```
task_clock_stop()
```

```
<built-in> procedure
```

Comments: Call `task_clock_stop`() when you want to take time out from scheduling real-time tasks. For instance, you want to temporarily suspend a game or simulation for a period of time.

Scheduling will resume when task_clock_start() is called.

Time-shared tasks can continue. The current task can also continue, unless it is a real-time task and it yields.

The time() function is not affected by this.

See Also: task_clock_start, task_schedule, task_yield, task_suspend, task_delay

task_create

creates a new task, given a home procedure and the arguments passed to it.

Signature:

```
task_create(integer rid, sequence args)
```

```
<built-in> function
```

Arguments: ≡ `rid` : an integer, the routine_id of a user-defined Euphoria procedure.
≡ `args` : a sequence, the list of arguments that will be passed to this procedure when the task starts executing.

Returns: An **atom**, a task identifier, created by the system. It can be used to identify this task to the other Euphoria multitasking routines.

Comments: `task_create`() creates a new task, but does not start it executing. You must call task_schedule() for this purpose.

Each task has its own set of private variables and its own call stack. Global and local variables are shared between all tasks.

If a run-time error is detected, the traceback will include information on all tasks, with the offending task listed first.

Many tasks can be created that all run the same procedure, possibly with different parameters.

A task cannot be based on a function, since there would be no way of using the function result.

Each task id is unique. `task_create`() never returns the same task id as it did before. Task id's are integer-valued atoms and can be as large as the largest integer-valued atom (15 digits).

See Also: task_schedule, task_yield, task_suspend, task_self

Example 1:

```
 mytask = task_create(routine_id("myproc"), {5, 9, "ABC"})
```

task_delay

suspends a task for a short period, allowing other tasks to run in the meantime.

Signature:

```
task_delay(atom delaytime)
```

```
public procedure
include task.e
namespace task
```

Arguments: ≡ `delaytime` : an atom, the duration of the delay in seconds.

Comments: This procedure is similar to sleep(), but allows for other tasks to run by yielding on a regular basis. Like sleep(), its argument needs not being an integer.

See Also: sleep

task_list

gets a sequence containing the task id's for all active or suspended tasks.

Signature:

```
task_list()
```

```
<built-in> function
```

Returns: A **sequence**, of atoms, the list of all task that are or may be scheduled.

Comments: This function lets you find out which tasks currently exist. Tasks that have terminated are not included. You can pass a task id to task_status() to find out more about a particular task.

See Also: task_status, task_create, task_schedule, task_yield, task_suspend

Example 1:

```
sequence tasks

tasks = task_list()
for i = 1 to length(tasks) do
    if task_status(tasks[i]) > 0 then
        printf(1, "task %d is active\n", tasks[i])
    end if
end for
```

task_schedule

schedules a task to run using a scheduling parameter.

Signature:

```
task_schedule(atom task_id, object schedule)
```

```
<built-in> procedure
```

Arguments: ≡ `task_id` : an atom, the identifier of a task that did not terminate yet.
≡ `schedule` : an object, describing when and how often to run the task.

Comments: `task_id` must have been returned by task_create().

The task scheduler, which is built-in to the Euphoria run-time system, will use `schedule` as a guide when scheduling this task. It may not always be possible to achieve the desired number of consecutive runs, or the desired time frame. For instance, a task might take so long before yielding control, that another task misses its desired time window.

`schedule` is being interpreted as follows:

`schedule` is an integer:

This defines `task_id` as time shared, and tells the task scheduler how many times it should the task in one burst before it considers running other tasks. `schedule` must be greater than zero then.

Increasing this count will increase the percentage of CPU time given to the selected task, while decreasing the percentage given to other time-shared tasks. Use trial and error to find the optimal trade off. It will also increase the efficiency of the program, since each actual task switch wastes a bit of time.

`schedule` is a sequence:

In this case, it must be a pair of positive atoms, the first one not being less than the second one. This defines `task_id` as a real time task. The pair states the minimum and maximum times, in seconds, to wait before running the task. The pair also sets the time interval for subsequent runs of the task, until the next call to `task_schedule`() or task_suspend().

Real-time tasks have a higher priority. Time-shared tasks are run when no real-time task is ready to execute.

-

See Also: task_create, task_yield, task_suspend

Example 1:

```
-- Task t1 will be executed up to 10 times in a row before
-- other time-shared tasks are given control. If a real-time
-- task needs control, t1 will lose control to the real-time task.
task_schedule(t1, 10)

-- Task t2 will be scheduled to run some time between 4 and 5 seconds
-- from now. Barring any rescheduling of t2, it will continue to
-- execute every 4 to 5 seconds thereafter.
task_schedule(t2, {4, 5})
```

task_self

returns the task id of the current task.

Signature:

```
task_self()
```

```
<built-in> function
```

Comments: This value may be needed, if a task wants to schedule or suspend itself.

See Also: task_create, task_schedule, task_yield, task_suspend

Example 1:

```
-- schedule self
task_schedule(task_self(), {5.9, 6.0})
```

task_status

returns the status of a task.

Signature:

```
task_status(atom task_id)
```

```
<built-in> function
```

Arguments: ≡ `task_id` : an atom, the id of the task being queried.

Returns: An **integer**,
• -1 -- task does not exist, or terminated
• 0 -- task is suspended
• 1 -- task is active

Comments: A task might want to know the status of one or more other tasks when deciding whether to proceed with some processing.

See Also: task_list, task_create, task_schedule, task_suspend

Example 1:

```
integer s

s = task_status(tid)
if s = 1 then
    puts(1, "ACTIVE\n")
elsif s = 0 then
    puts(1, "SUSPENDED\n")
else
    puts(1, "DOESN'T EXIST\n")
end if
```

task_suspend

suspends execution of a task.

Signature:

```
task_suspend(atom task_id)
```

```
<built-in> procedure
```

Arguments: ≡ `task_id` : an atom, the id of the task to suspend.

Comments: A suspended task will not be executed again unless there is a call to task_schedule() for the task.

`task_id` is a task id returned from task_create(). - Any task can suspend any other task. If a task suspends itself, the suspension will start as soon as the task calls task_yield().

Suspending a task and never scheduling it again is how to kill a task. There is no task_kill() primitives because undead tasks were creating too much trouble and confusion. As a general fact, nothing that impacts a running task can be effective as long as the task has not yielded.

See Also: task_create, task_schedule, task_self, task_yield

Example 1:

```
-- suspend task 15
task_suspend(15)

-- suspend current task
task_suspend(task_self())
```

task_yield

yields control to the scheduler.

Signature:

```
task_yield()
```

```
<built-in> procedure
```

Comments: The scheduler can then choose another task to run, or perhaps let the current task continue running.

Tasks should call `task_yield`() periodically so other tasks will have a chance to run. Only when `task_yield`() is called, is there a way for the scheduler to take back control from a task. This is what is known as cooperative multitasking.

A task can have calls to `task_yield`() in many different places in its code, and at any depth of subroutine call.

The scheduler will use the current scheduling parameter (see task_schedule), in determining when to return to the current task.

When control returns, execution will continue with the statement that follows `task_yield`(). The call-stack and all private variables will remain as they were when `task_yield`() was called. Global and local variables may have changed, due to the execution of other tasks.

Tasks should try to call `task_yield`() often enough to avoid causing real-time tasks to miss their time window, and to avoid blocking time-shared tasks for an excessive period of time. On the other hand, there is a bit of overhead in calling `task_yield`(), and this overhead is slightly larger when an actual switch to a different task takes place. A `task_yield`() where the same task continues executing takes less time.

A task should avoid calling `task_yield`() when it is in the middle of a delicate operation that requires exclusive access to some data. Otherwise a race condition could occur, where one task might interfere with an operation being carried out by another task. In some cases a task might need to mark some data as "locked" or "unlocked" in order to prevent this possibility. With cooperative multitasking, these concurrency issues are much less of a problem than with the preemptive multitasking that other languages support.

See Also: task_create, task_schedule, task_suspend

Example 1:

```
-- From Language war game.
-- This small task deducts life support energy from either the
-- large Euphoria ship or the small shuttle.
-- It seems to run "forever" in an infinite loop,
-- but it's actually a real-time task that is called
-- every 1.7 to 1.8 seconds throughout the game.
-- It deducts either 3 units or 13 units of life support energy each time.

procedure task_life()
-- independent task: subtract life support energy
    while TRUE do
        if shuttle then
            p_energy(-3)
        else
            p_energy(-13)
        end if
        task_yield()
    end while
end procedure
```

text

Routines

- sprintf
- sprint

---

text API

---

dequote

removes 'quotation' text from the argument.

Signature:

```
dequote(sequence text_in, object quote_pairs = {{"\"", "\""}},
integer esc = - 1)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `text_in` : The string or set of strings to de-quote.
≡ `quote_pairs` : A set of one or more sub-sequences of two strings, or an atom representing a single character to be used as both the open and close quotes. The first string in each sub-sequence is the opening quote to look for, and the second string is the closing quote. The default is "\"", "\"" which means that the output is 'quoted' if it is enclosed by double-quotation marks.
≡ `esc` : A single escape character. If this is not negative (the default), then this is used to 'escape' any embedded occurrences of the quote characters. In which case the 'escape' character is also removed.

Returns: A **sequence**, the original text but with 'quote' strings stripped of quotes.

Example 1:

```
 -- Using the defaults.
 s = dequote("\"The small man\"")
 -- 's' now contains "The small man"
```

Example 2:

```
 -- Using the defaults.
 s = dequote("(The small ?(?) man)", {{"(",")"}}, '?')
 -- 's' now contains "The small () man"
```

escape

escapes special characters in a string.

Signature:

```
escape(sequence s, sequence what = "\"")
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `s`: string to escape
≡ `what`: sequence of characters to escape defaults to escaping a double quote.

Returns: An escaped `sequence` representing `s`.

See Also: quote

Example 1:

```
sequence s = escape("John \"Mc\" Doe")
puts(1, s)
-- output is: John \"Mc\" Doe
```

format

formats a set of arguments into a string based on a supplied pattern.

Signature:

```
format(sequence format_pattern, object arg_list = {})
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `format_pattern` : A sequence: the pattern string that contains zero or more tokens.
≡ `arg_list` : An object: Zero or more arguments used in token replacement.

Returns: A string **sequence**, the original `format_pattern` but with tokens replaced by corresponding arguments.

Comments: The `format_pattern` string contains text and argument tokens. The resulting string is the same as the format string except that each token is replaced by an item from the argument list.

A token has the form **`[<Q>]`**, where <Q> is are optional qualifier codes.

The qualifier. `<Q>` is a set of zero or more codes that modify the default way that the argument is used to replace the token. The default replacement method is to convert the argument to its shortest string representation and use that to replace the token. This may be modified by the following codes, which can occur in any order.

Clearly, certain combinations of these qualifier codes do not make sense and in those situations, the rightmost clashing code is used and the others are ignored.

Any tokens in the format that have no corresponding argument are simply removed from the result. Any arguments that are not used in the result are ignored.

Any sequence argument that is not a string will be converted to its *pretty* format before being used in token replacement.

If a token is going to be replaced by a zero-length argument, all white space following the token until the next non-whitespace character is not copied to the result string.

Qualifiers:

Qualifier	Usage
N	('N' is an integer) The index of the argument to use
{id}	Uses the argument that begins with "id=" where "id" is an identifier name.

%envvar%	Uses the Environment Symbol 'envar' as an argument
w	For string arguments, if capitalizes the first letter in each word
u	For string arguments, it converts it to upper case.
l	For string arguments, it converts it to lower case.
<	For numeric arguments, it left justifies it.
>	For string arguments, it right justifies it.
c	Centers the argument.
z	For numbers, it zero fills the left side.
:S	('S' is an integer) The maximum size of the resulting field. Also, if 'S' begins with '0' the field will be zero-filled if the argument is an integer
.N	('N' is an integer) The number of digits after the decimal point
+	For positive numbers, show a leading plus sign
(	For negative numbers, enclose them in parentheses
b	For numbers, causes zero to be all blanks
s	If the resulting field would otherwise be zero length, this ensures that at least one space occurs between this token's field
t	After token replacement, the resulting string up to this point is trimmed.
X	Outputs integer arguments using hexadecimal digits.
B	Outputs integer arguments using binary digits.
?	The corresponding argument is a set of two strings. This uses the first string if the previous token's argument is not the value 1 or a zero-length string, otherwise it uses the second string.
[	Does not use any argument. Outputs a left-square-bracket symbol
,X	Insert thousands separators. The <X> is the character to use. If this is a dot "." then the decimal point is rendered using a comma. Does not apply to zero-filled fields. N.B. if hex or binary output was specified, the separators are every 4 digits otherwise they are every three digits.
T	If the argument is a number it is output as a text character, otherwise it is output as text string

See Also: sprintf

Example 1:

```
format("Cannot open file '[]' - code []", {"/usr/temp/work.dat", 32})
-- "Cannot open file '/usr/temp/work.dat' - code 32"

format("Err-[2], Cannot open file '[1]'", {"/usr/temp/work.dat", 32})
-- "Err-32, Cannot open file '/usr/temp/work.dat'"

format("[4w] [3z:2] [6] [5l] [2z:2], [1:4]", {2009,4,21,"DAY","MONTH","of"})
-- "Day 21 of month 04, 2009"

format("The answer is [:6.2]%", {35.22341})
-- "The answer is  35.22%"

format("The answer is [.6]", {1.2345})
-- "The answer is 1.234500"

format("The answer is [,,.2]", {1234.56})
-- "The answer is 1,234.56"

format("The answer is [,..2]", {1234.56})
-- "The answer is 1.234,56"

format("The answer is [,:.2]", {1234.56})
-- "The answer is 1:234.56"

format("[] [?]", {5, {"cats", "cat"}})
-- "5 cats"

format("[] [?]", {1, {"cats", "cat"}})
-- "1 cat"
```

```
format("[<:4]", {"abcdef"})
-- "abcd"

format("[>:4]", {"abcdef"})
-- "cdef"

format("[>:8]", {"abcdef"})
-- "  abcdef"

format("seq is []", {{1.2, 5, "abcdef", {3}}})
-- `seq is {1.2,5,"abcdef",{3}}`

format("Today is [{day}], the [{date}]", {"date=10/Oct/2012", "day=Wednesday"})
-- "Today is Wednesday, the 10/Oct/2012"

format("'A' is [T]", 65)
-- `'A' is A`
```

get_encoding_properties

gets the table of lowercase and uppercase characters that is used by

Signature:

```
get_encoding_properties()


public function
include text.e
namespace text
```

Returns: A **sequence**, containing three items.
{Encoding_Name, lower case_Set, upper case_Set}

See Also: lower, upper, set_encoding_properties

Example 1:

```
encode_sets = get_encoding_properties()
```

keyvalues

converts a string containing Key/Value pairs into a set of

Signature:

```
keyvalues(sequence source, object pair_delim = ";,",
object kv_delim = ":=", object quotes = "\"'`",
object whitespace = " \t\n\r", integer haskeys = 1)


public function
include text.e
namespace text
```

Arguments:
≡ `source` : a text sequence, containing the representation of the key/values.
≡ `pair_delim` : an object containing a list of elements that delimit one key/value pair from the next. The defaults are semi-colon (;) and comma (,).
≡ `kv_delim` : an object containing a list of elements that delimit the key from its value. The defaults are colon (:) and equal (=).
≡ `quotes` : an object containing a list of elements that can be used to enclose either keys or values that contain delimiters or whitespace. The defaults are double-quote ("), single-quote (') and back-quote (`)
≡ `whitespace` : an object containing a list of elements that are regarded as whitespace characters. The defaults are space, tab, new-line, and carriage-return.
≡ `haskeys` : an integer containing true or false. The default is true. When `true`, the `kv_delim` values are used to separate keys from values, but when `false` it is assumed that each 'pair' is actually just a value.

Returns: A **sequence**, of pairs. Each pair is in the form {key, value}.

Comments: String representations of atoms are not converted, either in the key or value part, but returned as any regular string instead.

If `haskeys` is `true`, but a substring only holds what appears to be a value, the key is synthesized as `p[n]`, where `n` is the number of the pair. See example #2.

By default, pairs can be delimited by either a comma or semi-colon ",;" and a key is delimited from its value by either an equal or a colon "=:". Whitespace between pairs, and between delimiters is ignored.

If you need to have one of the delimiters in the value data, enclose it in quotation marks. You can use any of single, double and back quotes, which also means you can quote quotation marks themselves. See example #3.

It is possible that the value data itself is a nested set of pairs. To do this enclose the value in parentheses. Nested sets can nested to any level. See example #4.

If a sub-list has only data values and not keys, enclose it in either braces or square brackets. See example #5. If you need to have a bracket as the first character in a data value, prefix it with a tilde. Actually a leading tilde will always just be stripped off regardless of what it prefixes. See example #6.

Example 1:

```
s= keyvalues("foo=bar, qwe=1234, asdf='contains space, comma, and equal(=)'")
-- s is
-- {
--   {"foo", "bar"},
--   {"qwe", "1234"},
--   {"asdf", "contains space, comma, and equal(=)"}
--  }
```

Example 2:

```
s = keyvalues("abc fgh=ijk def")
-- s is { {"p[1]", "abc"}, {"fgh", "ijk"}, {"p[3]", "def"} }
```

Example 3:

```
s = keyvalues("abc=`'quoted'`")
-- s is { {"abc", "'quoted'"} }
```

Example 4:

```
s = keyvalues("colors=(a=black, b=blue, c=red)")
-- s is { {"colors", {{"a", "black"}, {"b", "blue"},{"c", "red"}}  } }
s = keyvalues("colors=(black=[0,0,0], blue=[0,0,FF], red=[FF,0,0])")
-- s is
-- { {"colors",
--   {{"black",{"0", "0", "0"}},
--   {"blue",{"0", "0", "FF"}},
--   {"red", {"FF","0","0"}}}} }
```

Example 5:

```
s = keyvalues("colors=[black, blue, red]")
-- s is { {"colors", { "black", "blue", "red"}  } }
```

Example 6:

```
s = keyvalues("colors=~[black, blue, red]")
-- s is { {"colors", "[black, blue, red]"}  } }
-- The following is another way to do the same.
s = keyvalues("colors=`[black, blue, red]`")
-- s is { {"colors", "[black, blue, red]"}  } }
```

lower

converts an atom or sequence to lower case.

Signature:

```
lower(object x)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ x : Any Euphoria object.

Returns: A **sequence**, the lowercase version of x

Comments:

- For *windows* systems this uses the current code page for conversion
- For *unix* systems this only works on ASCII characters. It alters characters in the 'a'..'z' range. If you need to do case conversion with other encodings use the set_encoding_properties first.
- x may be a sequence of any shape, all atoms of which will be acted upon.

Warning:

When using ASCII encoding, this can also affect floating point numbers in the range 65 to 90.

See Also: upper, proper, set_encoding_properties, get_encoding_properties

Example 1:

```
s = lower("Euphoria")
-- s is "euphoria"

a = lower('B')
-- a is 'b'

s = lower({"Euphoria", "Programming"})
-- s is {"euphoria", "programming"}
```

proper

converts a text sequence to capitalized words.

Signature:

```
proper(sequence x)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ x : A text sequence.

Returns: A **sequence**, the Capitalized Version of x

Comments: A text sequence is one in which all elements are either characters or text sequences. This means that if a non-character is found in the input, it is not converted. However this rule only applies to elements on the same level, meaning that sub-sequences could be converted if they are actually text sequences.

See Also: lower upper

Example 1:

```
s = proper("euphoria programming language")
-- s is "Euphoria Programming Language"
s = proper("EUPHORIA PROGRAMMING LANGUAGE")
-- s is "Euphoria Programming Language"
s = proper({"EUPHORIA PROGRAMMING", "language", "rapid dEPLOYMENT", "sOfTwArE"})
-- s is {"Euphoria Programming", "Language", "Rapid Deployment", "Software"}
s = proper({'a', 'b', 'c'})
```

```
-- s is {'A', 'b', c'} -- "Abc"
s = proper({'a', 'b', 'c', 3.1472})
-- s is {'a', 'b', c', 3.1472} -- Unchanged because it contains a non-character.
s = proper({"abc", 3.1472})
-- s is {"Abc", 3.1472} -- The embedded text sequence is converted.
```

quote

returns a quoted version of the first argument.

Signature:

```
quote(sequence text_in, object quote_pair = {"\"", "\""},
integer esc = - 1, t_text sp = "")
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `text_in` : The string or set of strings to quote.
≡ `quote_pair` : A sequence of two strings. The first string is the opening quote to use, and the second string is the closing quote to use. The default is {"\"", "\""} which means that the output will be enclosed by double-quotation marks.
≡ `esc` : A single escape character. If this is not negative (the default), then this is used to 'escape' any embedded quote characters and 'esc' characters already in the `text_in` string.
≡ `sp` : A list of zero or more special characters. The `text_in` is only quoted if it contains any of the special characters. The default is "" which means that the `text_in` is always quoted.

Returns: A **sequence**, the quoted version of `text_in`.

See Also: escape

Example 1:

```
-- Using the defaults. Output enclosed in double-quotes, no escapes and no specials.
s = quote("The small man")
-- 's' now contains '"the small man"' including the double-quote characters.
```

Example 2:

```
s = quote("The small man", {"(", ")"} )
-- 's' now contains '(the small man)'
```

Example 3:

```
s = quote("The (small) man", {"(", ")"}, '~' )
-- 's' now contains '(The ~(small~) man)'
```

Example 4:

```
s = quote("The (small) man", {"(", ")"}, '~', "#" )
-- 's' now contains "the (small) man"
-- because the input did not contain a '#' character.
```

Example 5:

```
s = quote("The #1 (small) man", {"(", ")"}, '~', "#" )
-- 's' now contains '(the #1 ~(small~) man)'
-- because the input did contain a '#' character.
```

Example 6:

```
-- input is a set of strings...
s = quote({"a b c", "def", "g hi"},)
-- 's' now contains three quoted strings: '"a b c"', '"def"', and '"g hi"'
```

set_encoding_properties

sets the table of lowercase and uppercase characters that is used by

Signature:

```
set_encoding_properties(sequence en = "", sequence lc = "",
sequence uc = "")
```

```
public procedure
include text.e
namespace text
```

Arguments: ≡ `en` : The name of the encoding represented by these character sets
≡ `lc` : The set of lowercase characters
≡ `uc` : The set of upper case characters

Comments:

- `lc` and `uc` must be the same length.
- If no parameters are given, the default ASCII table is set.

See Also: lower, upper, get_encoding_properties

Example 1:

```
set_encoding_properties( "Elvish", "aeiouy", "AEIOUY")
```

Example 2:

```
set_encoding_properties( "1251") -- Loads a predefined code page.
```

sprint

returns the representation of any Euphoria object as a string of characters.

Signature:

```
sprint(object x)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `x` : Any Euphoria object.

Returns: A **sequence**, a string representation of `x`.

Comments: This is exactly the same as `print(fn, x)`, except that the output is returned as a sequence of characters, rather than being sent to a file or device. x can be any Euphoria object.

The atoms contained within `x` will be displayed to a maximum of ten significant digits, just as with print().

See Also: sprintf, printf

Example 1:

```
s = sprint(12345)
-- s is "12345"
```

Example 2:

```
s = sprint({10,20,30}+5)
-- s is "{15,25,35}"
```

sprintf

returns a string sequence based on the format string with embeded values (analogous to `printf`).

Signature:

```
sprintf(sequence format, object values)
```

```
<built-in> function
```

Arguments: ≡ `format` : a sequence, the text to print. This text may contain format specifiers.
≡ `values` : usually, a sequence of values. It should have as many elements as format specifiers in `format`, as these values will be substituted to the specifiers.

Returns: A **sequence**, of printable characters, representing `format` with the values in `values` spliced in.

Comments: `printf(fn, st, x)` is equivalent to `puts(fn, sprintf(st, x))`.

Some typical uses of `sprintf()` are:

# Converting numbers to strings. # Creating strings to pass to system(). # Creating formatted error messages that can be passed to a common error message handler.

See Also: printf, sprint, format

Example 1:

```
 s = sprintf("%08d", 12345)
 -- s is "00012345"
```

trim

trim all items in the supplied set from both the left end (head,start) and right end (tail,end)

Signature:

```
trim(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `source` : the sequence to trim.
≡ `what` : the set of item to trim from `source` (defaults to " \t\r\n").
≡ `ret_index` : If zero (the default) returns the trimmed sequence, otherwise it returns a 2-element sequence containing the index of the leftmost item and rightmost item **not** in `what`.

Returns: A **sequence**, if `ret_index` is zero, which is the trimmed version of `source`
A **2-element sequence**, if `ret_index` is not zero, in the form {left_index, right_index}.

See Also: trim_head, trim_tail

Example 1:

```
 object s
 s = trim("\r\nSentence read from a file\r\n", "\r\n")
 -- s is "Sentence read from a file"
 s = trim("\r\nSentence read from a file\r\n", "\r\n", TRUE)
 -- s is {3,27}
 s = trim(" This is a sentence.\n")  -- Default is to trim off all " \t\r\n"
 -- s is "This is a sentence."
```

trim_head

trims all items in the supplied set from the leftmost (start or head) of a sequence.

Signature:

```
trim_head(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `source` : the sequence to trim.
≡ `what` : the set of item to trim from `source` (defaults to " \t\r\n").
≡ `ret_index` : If zero (the default) returns the trimmed sequence, otherwise it returns the index of the leftmost item **not** in `what`.

Returns: A **sequence**, if `ret_index` is zero, which is the trimmed version of `source`
A **integer**, if `ret_index` is not zero, which is index of the leftmost element in `source` that is not in `what`.

See Also: trim_tail, trim, pad_head

Example 1:

```
object s
s = trim_head("\r\nSentence read from a file\r\n", "\r\n")
-- s is "Sentence read from a file\r\n"
s = trim_head("\r\nSentence read from a file\r\n", "\r\n", TRUE)
-- s is 3
```

trim_tail

trims all items in the supplied set from the rightmost (end or tail) of a sequence.

Signature:

```
trim_tail(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

```
public function
include text.e
namespace text
```

Arguments: ≡ `source` : the sequence to trim.
≡ `what` : the set of item to trim from `source` (defaults to " \t\r\n").
≡ `ret_index` : If zero (the default) returns the trimmed sequence, otherwise it returns the index of the rightmost item **not** in `what`.

Returns: A **sequence**, if `ret_index` is zero, which is the trimmed version of `source`
A **integer**, if `ret_index` is not zero, which is index of the rightmost element in `source` that is not in `what`.

See Also: trim_head, trim, pad_tail

Example 1:

```
object s
s = trim_tail("\r\nSentence read from a file\r\n", "\r\n")
-- s is "\r\nSentence read from a file"
s = trim_tail("\r\nSentence read from a file\r\n", "\r\n", TRUE)
-- s is 27
```

upper

Convert an atom or sequence to upper case.

Signature:

```
upper(object x)
```

```
public function
```

```
include text.e
namespace text
```

Arguments: ≡ `x` : Any Euphoria object.

Returns: A **sequence**, the uppercase version of `x`

Comments:

- For *windows* systems this uses the current code page for conversion
- For *unix* this only works on ASCII characters. It alters characters in the 'a'..'z' range. If you need to do case conversion with other encodings use the set_encoding_properties first.
- `x` may be a sequence of any shape, all atoms of which will be acted upon.

Warning:

When using ASCII encoding, this can also affects floating point numbers in the range 97 to 122.

See Also: lower, proper, set_encoding_properties, get_encoding_properties

Example 1:

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('b')
-- a is 'B'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

wrap

wraps text to a length limit.

Signature:

```
wrap(sequence content, integer width = 78, sequence wrap_with = "\n",
sequence wrap_at = " \t")


public function
include text.e
namespace text
```

Arguments:

- `content` - sequence content to wrap
- `width` - width to wrap at, defaults to 78
- `wrap_with` - sequence to wrap with, defaults to "\n"
- `wrap_at` - sequence of characters to wrap at, defaults to space and tab

Returns: Sequence containing wrapped text

Example 1:

```
sequence result = wrap("Hello, World")
-- result = "Hello, World"
```

Example 2:

```
sequence msg = "Hello, World. Today we are going to learn about apples."
sequence result = wrap(msg, 40)
-- result =
--    "Hello, World. today we are going to\n"
--    "learn about apples."
```

Example 3:

```
sequence msg = "Hello, World. Today we are going to learn about apples."
sequence result = wrap(msg, 40, "\n    ")
-- result =
--   "Hello, World. today we are going to\n"
--   "    learn about apples."
```

Example 4:

```
        sequence msg = "Hello, World. This, Is, A, Dummy, Sentence, Ok, World?"
        sequence result = wrap(msg, 30, "\n", ",")
        -- result =
        --   "Hello, World. This, Is, A,"
        --   "Dummy, Sentence, Ok, World?"
```

---

types

- OBJ_UNASSIGNED
- OBJ_INTEGER
- OBJ_ATOM
- OBJ_SEQUENCE
- object
- integer
- atom
- sequence
- FALSE
- TRUE

Predefined character sets

- CS_FIRST
- CS_Consonant
- CS_Vowel
- CS_Hexadecimal
- CS_Whitespace
- CS_Punctuation
- CS_Printable
- CS_Displayable
- CS_Lowercase
- CS_Uppercase
- CS_Alphanumeric
- CS_Identifier
- CS_Alphabetic
- CS_ASCII
- CS_Control
- CS_Digit
- CS_Graphic
- CS_Bytes
- CS_SpecWord
- CS_Boolean
- CS_LAST

Support Functions

- char_test
- set_default_charsets
- get_charsets
- set_charsets

Types

- boolean

t_boolean
t_alnum
t_identifier
t_alpha
t_ascii
t_cntrl
t_digit
t_graph
t_specword
t_bytearray
t_lower
t_print
t_display
t_punct
t_space
t_upper
t_xdigit
t_vowel
t_consonant
integer_array
t_text
number_array
sequence_array
ascii_string
string
cstring
INVALID_ROUTINE_ID
NO_ROUTINE_ID

---

types API

CS_ASCII

Signature:

```
CS_ASCII


public enum
include types.e
namespace types
```

CS_Alphabetic

Signature:

```
CS_Alphabetic


public enum
include types.e
namespace types
```

CS_Alphanumeric

Signature:

```
CS_Alphanumeric
```

```
public enum
include types.e
namespace types
```

CS_Boolean

Signature:

```
CS_Boolean
```

```
public enum
include types.e
namespace types
```

CS_Bytes

Signature:

```
CS_Bytes
```

```
public enum
include types.e
namespace types
```

CS_Consonant

Signature:

```
CS_Consonant
```

```
public enum
include types.e
namespace types
```

CS_Control

Signature:

```
CS_Control
```

```
public enum
include types.e
namespace types
```

CS_Digit

Signature:

```
CS_Digit
```

```
public enum
include types.e
namespace types
```

CS_Displayable

Signature:

```
CS_Displayable
```

```
public enum
include types.e
namespace types
```

CS_FIRST

Signature:

```
CS_FIRST
```

```
public enum
include types.e
namespace types
```

CS_Graphic

Signature:

```
CS_Graphic
```

```
public enum
include types.e
namespace types
```

CS_Hexadecimal

Signature:

```
CS_Hexadecimal
```

```
public enum
include types.e
namespace types
```

CS_Identifier

Signature:

```
CS_Identifier
```

```
public enum
include types.e
namespace types
```

CS_LAST

Signature:

```
CS_LAST
```

```
public enum
include types.e
namespace types
```

CS_Lowercase

Signature:

```
CS_Lowercase
```

```
public enum
include types.e
namespace types
```

CS_Printable

Signature:

```
CS_Printable
```

```
public enum
include types.e
namespace types
```

CS_Punctuation

Signature:

```
CS_Punctuation
```

```
public enum
include types.e
namespace types
```

CS_SpecWord

Signature:

```
CS_SpecWord
```

```
public enum
include types.e
namespace types
```

CS_Uppercase

Signature:

```
CS_Uppercase
```

```
public enum
include types.e
namespace types
```

CS_Vowel

Signature:

```
CS_Vowel
```

```
public enum
include types.e
namespace types
```

CS_Whitespace

Signature:

```
CS_Whitespace
```

```
public enum
include types.e
namespace types
```

FALSE

is a FALSE Boolean value.

Signature:

```
FALSE
```

```
public constant
include types.e
namespace types
```

INVALID_ROUTINE_ID

indicates no valid routine_id.

Signature:

```
INVALID_ROUTINE_ID
```

```
public constant
include types.e
namespace types
```

Comments: value returned from routine_id() when the routine does not exist or is out of scope. this is typically seen as -1 in legacy code.

NO_ROUTINE_ID

a flag for no routine_id

Signature:

```
NO_ROUTINE_ID
```

```
public constant
include types.e
namespace types
```

Comments: This constant is to be used as a flag for no routine_id() supplied.

OBJ_ATOM

Object is atom

Signature:

```
OBJ_ATOM
```

```
public constant
include types.e
namespace types
```

OBJ_INTEGER

Object is integer

Signature:

```
OBJ_INTEGER
```

```
public constant
include types.e
namespace types
```

OBJ_SEQUENCE

Object is sequence

Signature:

```
OBJ_SEQUENCE
```

```
public constant
include types.e
namespace types
```

OBJ_UNASSIGNED

Object not assigned

Signature:

```
OBJ_UNASSIGNED
```

```
public constant
include types.e
namespace types
```

TRUE

is a TRUE Boolean value.

Signature:

```
TRUE
```

```
public constant
include types.e
namespace types
```

ascii_string

tests for ASCII characters.

Signature:

```
ascii_string(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more ASCII characters.

Example 1:

```
 ascii_string(-1)             -- FALSE (not a sequence)
 ascii_string("abc")          -- TRUE (all single ASCII characters)
 ascii_string({1, 2, "abc"}) -- FALSE (contains a sequence)
 ascii_string({1, 2, 9.7})    -- FALSE (contains a non-integer)
 ascii_string({1, 2, 'a'})    -- TRUE
 ascii_string({1, -2, 'a'})   -- FALSE (contains a negative integer)
 ascii_string({})             -- TRUE
```

atom

tests if the supplied argument `x` to an atom.

Signature:

```
atom(object x)
```

```
<built-in> type
```

Returns: # An integer.
…… ♦ 1 if `x` is an atom.
…… ♦ 0 if `x` is not an atom.

See Also: sequence(), object(), integer()

Example 1:

```
 ? atom(1) --> 1
 ? atom(1.1) --> 1
 ? atom("1") --> 0
```

boolean

the Boolean datatype.

Signature:

```
boolean(object test_data)
```

```
public type
```

```
include types.e
namespace types
```

Returns: Returns TRUE if argument is 1 or 0

Returns FALSE if the argument is anything else other than 1 or 0.

Example 1:

```
boolean(-1)             -- FALSE
boolean(0)              -- TRUE
boolean(1)              -- TRUE
boolean(1.234)          -- FALSE
boolean('A')            -- FALSE
boolean('9')            -- FALSE
boolean('?')            -- FALSE
boolean("abc")          -- FALSE
boolean("ab3")          -- FALSE
boolean({1,2,"abc"})    -- FALSE
boolean({1, 2, 9.7)     -- FALSE
boolean({})             -- FALSE (empty sequence)
```

char_test

determines whether one or more characters are in a given character set.

Signature:

```
char_test(object test_data, sequence char_set)
```

```
public function
include types.e
namespace types
```

Arguments: ≡ `test_data` : an object to test, either a character or a string
≡ `char_set` : a sequence, either a list of allowable characters, or a list of pairs representing allowable ranges.

Returns: An **integer**, 1 if all characters are allowed, else 0.

Comments: `pCharset` is either a simple sequence of characters eg. "qwertyuiop[]
" or a sequence of character pairs, which represent allowable ranges of characters. eg. Alphabetic is defined as {{'a','z'}, {'A', 'Z'}}

To add an isolated character to a character set which is defined using ranges, present it as a range of length 1, like in `{%,%}`.

Example 1:

```
char_test("ABCD", {{'A', 'D'}})
-- TRUE, every char is in the range 'A' to 'D'

char_test("ABCD", {{'A', 'C'}})
-- FALSE, not every char is in the range 'A' to 'C'

char_test("Harry", {{'a', 'z'}, {'D', 'J'}})
-- TRUE, every char is either in the range 'a' to 'z',
--       or in the range 'D' to 'J'

char_test("Potter", "novel")
-- FALSE, not every character is in the set 'n', 'o', 'v', 'e', 'l'
```

cstring

tests for string sequences (disallows null character).

Signature:

```
cstring(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more non-null byte characters.

Example 1:

```
cstring(-1)             -- FALSE (not a sequence)
cstring("abc'6")        -- TRUE (all single byte characters)
cstring({1, 2, "abc'6"}) -- FALSE (contains a sequence)
cstring({1, 2, 9.7})     -- FALSE (contains a non-integer)
cstring({1, 2, 'a'})     -- TRUE
cstring({1, 2, 'a', 0}) -- FALSE (contains a null byte)
cstring({1, -2, 'a'})    -- FALSE (contains a negative integer)
cstring({})             -- TRUE
```

get_charsets

gets the definition for each of the defined character sets.

Signature:

```
get_charsets()
```

```
public function
include types.e
namespace types
```

Returns: A **sequence**, of pairs. The first element of each pair is the character set id , eg. CS_Whitespace, and the second is the definition of that character set.

Comments: This is the same format required for the set_charsets() routine.

See Also: set_charsets, set_default_charsets

Example 1:

```
sequence sets
sets = get_charsets()
```

integer

tests if the supplied argument x to see if it is an integer.

Signature:

```
integer(object x)
```

```
<built-in> type
```

Returns: # An integer.
…… ♦ 1 if x is an integer.
…… ♦ 0 if x is not an integer.

See Also: sequence(), object(), atom()

Example 1:

```
? integer(1) --> 1
? integer(1.1) --> 0
? integer("1") --> 0
```

integer_array

tests if elements are integers.

Signature:

```
integer_array(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more integers.

Example 1:

```
integer_array(-1)            -- FALSE (not a sequence)
integer_array("abc")         -- TRUE (all single characters)
integer_array({1, 2, "abc"}) -- FALSE (contains a sequence)
integer_array({1, 2, 9.7})    -- FALSE (contains a non-integer)
integer_array({1, 2, 'a'})    -- TRUE
integer_array({})            -- TRUE
```

number_array

tests elements if they are atoms.

Signature:

```
number_array(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more numbers.

Example 1:

```
number_array(-1)            -- FALSE (not a sequence)
number_array("abc")         -- TRUE (all single characters)
number_array({1, 2, "abc"}) -- FALSE (contains a sequence)
number_array(1, 2, 9.7})    -- TRUE
number_array(1, 2, 'a'})    -- TRUE
number_array({})            -- TRUE
```

object

returns information about the object type of the supplied argument x.

Signature:

```
object(object x)
```

```
<built-in> type
```

Returns: # An integer.
…… ♦ OBJ_UNASSIGNED if x has not been assigned anything yet.
…… ♦ OBJ_INTEGER if x holds an integer value.
…… ♦ OBJ_ATOM if x holds a number that is not an integer.
…… ♦ OBJ_SEQUENCE if x holds a sequence value.

See Also: sequence(), integer(), atom()

Example 1:

```
? object(1) --> OBJ_INTEGER
? object(1.1) --> OBJ_ATOM
? object("1") --> OBJ_SEQUENCE
object x
? object(x) --> OBJ_UNASSIGNED
```

sequence

tests if the supplied argument x if it is a sequence.

Signature:

```
sequence( object x)
```

```
<built-in> type
```

Returns: # An integer.
…… ♦ 1 if x is a sequence.
…… ♦ 0 if x is not an sequence.

See Also: integer(), object(), atom()

Example 1:

```
 ? sequence(1) --> 0
 ? sequence({1}) --> 1
 ? sequence("1") --> 1
```

sequence_array

tests elements if they are sequences.

Signature:

```
sequence_array(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more sequences.

Example 1:

```
 sequence_array(-1)            -- FALSE (not a sequence)
 sequence_array("abc")         -- FALSE (all single characters)
 sequence_array({1, 2, "abc"}) -- FALSE (contains some atoms)
 sequence_array({1, 2, 9.7})   -- FALSE
 sequence_array({1, 2, 'a'})   -- FALSE
 sequence_array({"abc", {3.4, 99182.78737}}) -- TRUE
 sequence_array({})            -- TRUE
```

set_charsets

sets the definition for one or more defined character sets.

Signature:

```
set_charsets(sequence charset_list)
```

```
public procedure
include types.e
namespace types
```

Arguments: ≡ `charset_list` : a sequence of zero or more character set definitions.

Comments: `charset_list` must be a sequence of pairs. The first element of each pair is the character set id (for example: CS_Whitespace) and the second is the definition of that character set.

This is the same format returned by the get_charsets() routine.

You cannot create new character sets using this routine.

See Also: get_charsets

Example 1:

```
set_charsets({{CS_Whitespace, " \t"}})
t_space('\n') --> FALSE

t_specword('$') --> FALSE
set_charsets({{CS_SpecWord, "_-#$%"}})
t_specword('$') --> TRUE
```

set_default_charsets

sets all the defined character sets to their default definitions.

Signature:

```
set_default_charsets()
```

```
public procedure
include types.e
namespace types
```

Example 1:

```
set_default_charsets()
```

string

tests for string sequences (allows null character).

Signature:

```
string(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more byte characters.

Example 1:

```
string(-1)            -- FALSE (not a sequence)
string("abc'6")       -- TRUE (all single byte characters)
string({1, 2, "abc'6"}) -- FALSE (contains a sequence)
string({1, 2, 9.7})    -- FALSE (contains a non-integer)
string({1, 2, 'a'})    -- TRUE
string({1, 2, 'a', 0}) -- TRUE (even though it contains a null byte)
string({1, -2, 'a'})   -- FALSE (contains a negative integer)
string({})            -- TRUE
```

t_alnum

tests for alphanumeric values.

Signature:

```
t_alnum(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an alphanumeric character or if every element of the argument is an alphanumeric character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphanumeric elements

Example 1:

```
  t_alnum(-1)             -- FALSE
  t_alnum(0)              -- FALSE
  t_alnum(1)              -- FALSE
  t_alnum(1.234)          -- FALSE
  t_alnum('A')            -- TRUE
  t_alnum('9')            -- TRUE
  t_alnum('?')            -- FALSE
  t_alnum("abc")          -- TRUE (every element is alphabetic or a digit)
  t_alnum("ab3")          -- TRUE
  t_alnum({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_alnum({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_alnum({})             -- FALSE (empty sequence)
```

t_alpha

tests for alphabetic characters.

Signature:

```
t_alpha(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an alphabetic character or if every element of the argument is an alphabetic character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphabetic elements

Example 1:

```
  t_alpha(-1)             -- FALSE
  t_alpha(0)              -- FALSE
  t_alpha(1)              -- FALSE
  t_alpha(1.234)          -- FALSE
  t_alpha('A')            -- TRUE
  t_alpha('9')            -- FALSE
  t_alpha('?')            -- FALSE
  t_alpha("abc")          -- TRUE (every element is alphabetic)
  t_alpha("ab3")          -- FALSE
  t_alpha({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_alpha({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_alpha({})             -- FALSE (empty sequence)
```

t_ascii

tests for ASCII values.

Signature:

```
t_ascii(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an ASCII character or if every element of the argument is an ASCII character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-ASCII elements

Example 1:

```
  t_ascii(-1)             -- FALSE
  t_ascii(0)              -- TRUE
  t_ascii(1)              -- TRUE
  t_ascii(1.234)          -- FALSE
  t_ascii('A')            -- TRUE
  t_ascii('9')            -- TRUE
  t_ascii('?')            -- TRUE
  t_ascii("abc")          -- TRUE (every element is ascii)
  t_ascii("ab3")          -- TRUE
  t_ascii({1, 2, "abc"})  -- FALSE (contains a sequence)
  t_ascii({1, 2, 9.7})     -- FALSE (contains a non-integer)
  t_ascii({})             -- FALSE (empty sequence)
```

t_boolean

tests for boolean data.

Signature:

```
t_boolean(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is boolean (1 or 0) or if every element of the argument is boolean.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-boolean elements

Example 1:

```
  t_boolean(-1)             -- FALSE
  t_boolean(0)              -- TRUE
  t_boolean(1)              -- TRUE
  t_boolean({1, 1, 0})      -- TRUE
  t_boolean({1, 1, 9.7})     -- FALSE
  t_boolean({})             -- FALSE (empty sequence)
```

t_bytearray

tests for valid bytes.

Signature:

```
t_bytearray(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a byte or if every element of the argument is a byte. (Integers from 0 to 255)

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-byte

Example

1:

```
t_bytearray(-1)             -- FALSE (contains value less than zero)
t_bytearray(0)              -- TRUE
t_bytearray(1)              -- TRUE
t_bytearray(10)             -- TRUE
t_bytearray(100)            -- TRUE
t_bytearray(1000)           -- FALSE (greater than 255)
t_bytearray(1.234)          -- FALSE (contains a floating number)
t_bytearray('A')            -- TRUE
t_bytearray('9')            -- TRUE
t_bytearray('?')            -- TRUE
t_bytearray(' ')            -- TRUE
t_bytearray("abc")          -- TRUE
t_bytearray("ab3")          -- TRUE
t_bytearray("123")          -- TRUE
t_bytearray({1, 2, "abc"}) -- FALSE (contains a sequence)
t_bytearray({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_bytearray({1, 2, 'a'})     -- TRUE
t_bytearray({})             -- FALSE (empty sequence)
```

t_cntrl

Returns TRUE if argument is an Control character or if every element of

Signature:

```
t_cntrl(object test_data)
```

```
public type
include types.e
namespace types
```

Example 1:

```
t_cntrl(-1)             -- FALSE
t_cntrl(0)              -- TRUE
t_cntrl(1)              -- TRUE
t_cntrl(1.234)          -- FALSE
t_cntrl('A')            -- FALSE
t_cntrl('9')            -- FALSE
t_cntrl('?')            -- FALSE
t_cntrl("abc")          -- FALSE (every element is ascii)
t_cntrl("ab3")          -- FALSE
t_cntrl({1, 2, "abc"}) -- FALSE (contains a sequence)
t_cntrl({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_cntrl({1, 2, 'a'})     -- FALSE (contains a non-control)
t_cntrl({})             -- FALSE (empty sequence)
```

t_consonant

tests for consonant characters.

Signature:

```
t_consonant(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a consonant character or if every element of the argument is an consonant character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-consonant character.

Example 1:

```
  t_consonant(-1)            -- FALSE
  t_consonant(0)             -- FALSE
  t_consonant(1)             -- FALSE
  t_consonant(1.234)         -- FALSE
  t_consonant('A')           -- FALSE
  t_consonant('9')           -- FALSE
  t_consonant('?')           -- FALSE
  t_consonant("abc")         -- FALSE
  t_consonant("rTfM")        -- TRUE
  t_consonant("123")         -- FALSE
  t_consonant({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_consonant({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_consonant({1, 2, 'a'})    -- FALSE (contains a non-digit)
  t_consonant({})            -- FALSE (empty sequence)
```

t_digit

tests for numerical digits.

Signature:

```
t_digit(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an digit character or if every element of the argument is an digit character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-digits

Example 1:

```
  t_digit(-1)            -- FALSE
  t_digit(0)             -- FALSE
  t_digit(1)             -- FALSE
  t_digit(1.234)         -- FALSE
  t_digit('A')           -- FALSE
  t_digit('9')           -- TRUE
  t_digit('?')           -- FALSE
  t_digit("abc")         -- FALSE
  t_digit("ab3")         -- FALSE
  t_digit("123")         -- TRUE
  t_digit({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_digit({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_digit({1, 2, 'a'})    -- FALSE (contains a non-digit)
  t_digit({})            -- FALSE (empty sequence)
```

t_display

tests if characters can be displayed.

Signature:

```
t_display(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a character that can be displayed or if every element of the argument is a character that can be displayed.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains characters that cannot be displayed.

Example 1:

```
  t_display(-1)            -- FALSE
  t_display(0)             -- FALSE
  t_display(1)             -- FALSE
  t_display(1.234)         -- FALSE
  t_display('A')           -- TRUE
  t_display('9')           -- TRUE
  t_display('?')           -- TRUE
  t_display("abc")         -- TRUE
  t_display("ab3")         -- TRUE
  t_display("123")         -- TRUE
  t_display("123 ")        -- TRUE
  t_display("123\n")       -- TRUE
  t_display({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_display({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_display({1, 2, 'a'})    -- FALSE
  t_display({})            -- FALSE (empty sequence)
```

t_graph

tests for printable characters.

Signature:

```
t_graph(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a glyph character or if every element of the argument is a glyph character. (One that is visible when displayed)

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-glyph

Example 1:

```
  t_graph(-1)            -- FALSE
  t_graph(0)             -- FALSE
  t_graph(1)             -- FALSE
  t_graph(1.234)         -- FALSE
  t_graph('A')           -- TRUE
  t_graph('9')           -- TRUE
  t_graph('?')           -- TRUE
  t_graph(' ')           -- FALSE
  t_graph("abc")         -- TRUE
  t_graph("ab3")         -- TRUE
  t_graph("123")         -- TRUE
  t_graph({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_graph({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_graph({1, 2, 'a'})    -- FALSE (control chars (1,2) don't have glyphs)
  t_graph({})            -- FALSE (empty sequence)
```

t_identifier

tests for valid identifier text strings.

Signature:

```
t_identifier(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an alphanumeric character or if every element of the argument is an alphanumeric character and that the first character is not numeric and the whole group of characters are not all numeric.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-alphanumeric elements

Example 1:

```
  t_identifier(-1)            -- FALSE
  t_identifier(0)             -- FALSE
  t_identifier(1)             -- FALSE
  t_identifier(1.234)         -- FALSE
  t_identifier('A')           -- TRUE
  t_identifier('9')           -- FALSE
  t_identifier('?')           -- FALSE
  t_identifier("abc")         -- TRUE (every element is alphabetic or a digit)
  t_identifier("ab3")         -- TRUE
  t_identifier("ab_3")        -- TRUE (underscore is allowed)
  t_identifier("1abc")        -- FALSE (identifier cannot start with a number)
  t_identifier("102")         -- FALSE (identifier cannot be all numeric)
  t_identifier({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_identifier({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_identifier({})            -- FALSE (empty sequence)
```

t_lower

tests for lowercase characters.

Signature:

```
t_lower(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a lowercase character or if every element of the argument is an lowercase character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-lowercase

Example 1:

```
  t_lower(-1)            -- FALSE
  t_lower(0)             -- FALSE
  t_lower(1)             -- FALSE
  t_lower(1.234)         -- FALSE
  t_lower('A')           -- FALSE
  t_lower('9')           -- FALSE
  t_lower('?')           -- FALSE
  t_lower("abc")         -- TRUE
  t_lower("ab3")         -- FALSE
  t_lower("123")         -- TRUE
  t_lower({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_lower({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_lower({1, 2, 'a'})    -- FALSE (contains a non-digit)
  t_lower({})            -- FALSE (empty sequence)
```

t_print

tests for printable characters.

Signature:

```
t_print(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a character that has an ASCII glyph or if every element of the argument is a character that has an ASCII glyph.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains characters that do not have an ASCII glyph.

Example 1:

```
  t_print(-1)            -- FALSE
  t_print(0)             -- FALSE
  t_print(1)             -- FALSE
  t_print(1.234)         -- FALSE
  t_print('A')           -- TRUE
  t_print('9')           -- TRUE
  t_print('?')           -- TRUE
  t_print("abc")         -- TRUE
  t_print("ab3")         -- TRUE
  t_print("123")         -- TRUE
  t_print("123 ")        -- FALSE (contains a space)
  t_print("123\n")       -- FALSE (contains a new-line)
  t_print({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_print({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_print({1, 2, 'a'})    -- FALSE
  t_print({})            -- FALSE (empty sequence)
```

t_punct

tests for punctuation characters.

Signature:

```
t_punct(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an punctuation character or if every element of the argument is an punctuation character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-punctuation symbols.

Example 1:

```
  t_punct(-1)            -- FALSE
  t_punct(0)             -- FALSE
  t_punct(1)             -- FALSE
  t_punct(1.234)         -- FALSE
  t_punct('A')           -- FALSE
  t_punct('9')           -- FALSE
  t_punct('?')           -- TRUE
  t_punct("abc")         -- FALSE
  t_punct("(-)")         -- TRUE
  t_punct("123")         -- TRUE
  t_punct({1, 2, "abc"}) -- FALSE (contains a sequence)
  t_punct({1, 2, 9.7})    -- FALSE (contains a non-integer)
  t_punct({1, 2, 'a'})    -- FALSE (contains a non-digit)
  t_punct({})            -- FALSE (empty sequence)
```

t_space

tests for whitespace characters.

Signature:

```
t_space(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a whitespace character or if every element of the argument is an whitespace character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-whitespace character.

Example 1:

```
t_space(-1)            -- FALSE
t_space(0)             -- FALSE
t_space(1)             -- FALSE
t_space(1.234)         -- FALSE
t_space('A')           -- FALSE
t_space('9')           -- FALSE
t_space('\t')          -- TRUE
t_space("abc")         -- FALSE
t_space("123")         -- FALSE
t_space({1, 2, "abc"}) -- FALSE (contains a sequence)
t_space({1, 2, 9.7})    -- FALSE (contains a non-integer)
t_space({1, 2, 'a'})    -- FALSE (contains a non-digit)
t_space({})            -- FALSE (empty sequence)
```

t_specword

tests for a special word character.

Signature:

```
t_specword(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a special word character or if every element of the argument is a special word character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-special-word characters.

Comments: A *special word character* is any character that is not normally part of a word but in certain cases may be considered. This is most commonly used when looking for words in programming source code which allows an underscore as a word character.

Example 1:

```
t_specword(-1)            -- FALSE
t_specword(0)             -- FALSE
t_specword(1)             -- FALSE
t_specword(1.234)         -- FALSE
t_specword('A')           -- FALSE
t_specword('9')           -- FALSE
t_specword('?')           -- FALSE
t_specword('_')           -- TRUE
t_specword("abc")         -- FALSE
t_specword("ab3")         -- FALSE
t_specword("123")         -- FALSE
t_specword({1, 2, "abc"}) -- FALSE (contains a sequence)
t_specword({1, 2, 9.7})    -- FALSE (contains a non-integer)
t_specword({1, 2, 'a'})    -- FALSE (control chars (1,2) don't have glyphs)
t_specword({})            -- FALSE (empty sequence)
```

t_text

tests for characters.

Signature:

```
t_text(object x)
```

```
public type
include types.e
namespace types
```

Returns: TRUE if argument is a sequence that only contains zero or more characters.

Example 1:

```
t_text(-1)              -- FALSE (not a sequence)
t_text("abc")           -- TRUE (all single characters)
t_text({1, 2, "abc"}) -- FALSE (contains a sequence)
t_text({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_text({1, 2, 'a'})     -- TRUE
t_text({1, -2, 'a'})    -- FALSE (contains a negative integer)
t_text({})              -- TRUE
```

t_upper

tests for uppercase characters.

Signature:

```
t_upper(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an uppercase character or if every element of the argument is an uppercase character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-uppercase characters.

Example 1:

```
t_upper(-1)             -- FALSE
t_upper(0)              -- FALSE
t_upper(1)              -- FALSE
t_upper(1.234)          -- FALSE
t_upper('A')            -- TRUE
t_upper('9')            -- FALSE
t_upper('?')            -- FALSE
t_upper("abc")          -- FALSE
t_upper("ABC")          -- TRUE
t_upper("123")          -- FALSE
t_upper({1, 2, "abc"}) -- FALSE (contains a sequence)
t_upper({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_upper({1, 2, 'a'})     -- FALSE (contains a non-digit)
t_upper({})             -- FALSE (empty sequence)
```

t_vowel

tests for vowel characters.

Signature:

```
t_vowel(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is a vowel or if every element of the argument is a vowel character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-vowels

Example 1:

```
t_vowel(-1)             -- FALSE
t_vowel(0)              -- FALSE
t_vowel(1)              -- FALSE
t_vowel(1.234)          -- FALSE
t_vowel('A')            -- TRUE
t_vowel('9')            -- FALSE
t_vowel('?')            -- FALSE
t_vowel("abc")          -- FALSE
t_vowel("aiu")          -- TRUE
t_vowel("123")          -- FALSE
t_vowel({1, 2, "abc"}) -- FALSE (contains a sequence)
t_vowel({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_vowel({1, 2, 'a'})     -- FALSE (contains a non-digit)
t_vowel({})             -- FALSE (empty sequence)
```

t_xdigit

tests for hexadecimal digits.

Signature:

```
t_xdigit(object test_data)
```

```
public type
include types.e
namespace types
```

Returns: Returns TRUE if argument is an hexadecimal digit character or if every element of the argument is an hexadecimal digit character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-hexadecimal character.

Example 1:

```
t_xdigit(-1)             -- FALSE
t_xdigit(0)              -- FALSE
t_xdigit(1)              -- FALSE
t_xdigit(1.234)          -- FALSE
t_xdigit('A')            -- TRUE
t_xdigit('9')            -- TRUE
t_xdigit('?')            -- FALSE
t_xdigit("abc")          -- TRUE
t_xdigit("fgh")          -- FALSE
t_xdigit("123")          -- TRUE
t_xdigit({1, 2, "abc"}) -- FALSE (contains a sequence)
t_xdigit({1, 2, 9.7})     -- FALSE (contains a non-integer)
t_xdigit({1, 2, 'a'})     -- FALSE (contains a non-digit)
t_xdigit({})             -- FALSE (empty sequence)
```

unittest

Background Unit testing is the process of assuring that the smallest programming units are actually delivering functionality that complies with their specification. The units in question are usually individual routines rather than whole programs or applications.

The theory is that if the components of a system are working correctly, then there is a high probability that a system using those components can be made to work correctly.

In Euphoria terms, this framework provides the tools to make testing and reporting on functions and procedures easy and standardized. It gives us a simple way to write a test case and to report on the findings.
Example:

```
include std/unittest.e

test_equal( "Power function test #1", 4, power(2, 2))
test_equal( "Power function test #2", 4, power(16, 0.5))

test_report()
```

Name your test file in the special manner, `t_NAME.e` and then simply run `eutest` in that directory.

```
C:\Euphoria> eutest
t_math.e:
failed: Bad math, expected: 100 but got: 8
2 tests run, 1 passed, 1 failed, 50.0% success

===== Test failure summary:
FAIL: t_math.e

2 file(s) run 1 file(s) failed, 50.0% success--
```

In this example, we use the `test_equal` function to record the result of a test. The first parameter is the name of the test, which can be anything and is displayed if the test fails. The second parameter is the expected result -- what we expect the function being tested to return. The third parameter is the actual result returned by the function being tested. This is usually written as a call to the function itself.

It is typical to provide as many test cases as would be required to give us confidence that the function is being truly exercised. This includes calling it with typical values and edge-case or exceptional values. It is also useful to test the function's error handling by calling it with bad parameters.

When a test fails, the framework displays a message, showing the test's name, the expected result and the actual result. You can configure the framework to display each test run, regardless of whether it fails or not.

After running a series of tests, you can get a summary displayed by calling the `test_report`() procedure. To get a better feel for unit testing, have a look at the provided test cases for the standard library in the *tests* directory.

When included in your program, unittest.e sets a crash handler to log a crash as a failure.

---

unittest API

TEST_QUIET

Signature:

```
TEST_QUIET
```

```
public enum
include unittest.e
namespace unittest
```

TEST_SHOW_ALL

Signature:

```
TEST_SHOW_ALL
```

```
public enum
include unittest.e
namespace unittest
```

TEST_SHOW_FAILED_ONLY

Signature:

```
TEST_SHOW_FAILED_ONLY
```

```
public enum
include unittest.e
namespace unittest
```

assert

records whether a test passes. If it fails, the program also fails.

Signature:

```
assert(object name, object outcome)
```

```
public procedure
include unittest.e
```

```
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test
≡ `outcome` : an object, some actual value that should not be zero.

Comments: This is identical to `test_true()` except that if the test fails, the program will also be forced to fail at this point.

See Also: test_equal, test_not_equal, test_false, test_pass, test_fail

set_accumulate_summary

requests the test report to save run stats in "unittest.dat" before exiting.

Signature:

```
set_accumulate_summary(integer accumulate)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `accumulate` : an integer, zero not to accumulate, nonzero to accumulate.

Comments: The file "unittest.dat" is appended to with {t,f}
: where :: *t* is total number of tests run :: *f* is the total number of tests that failed

set_test_abort

sets the behavior on test failure, and return previous value.

Signature:

```
set_test_abort(integer abort_test)
```

```
public function
include unittest.e
namespace unittest
```

Arguments: ≡ `abort_test` : an integer, the new value for this setting.

Returns: An **integer**, the previous value for the setting.

Comments: By default, the tests go on even if a file crashed.

set_test_verbosity

sets the amount of information that is returned about passed and failed tests.

Signature:

```
set_test_verbosity(atom verbosity)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `verbosity` : an atom which takes predefined values for verbosity levels.

Comments: The following values are allowable for `verbosity`:

- `TEST_QUIET` -- 0,
- `TEST_SHOW_FAILED_ONLY` -- 1
- `TEST_SHOW_ALL` -- 2

However, anything less than `TEST_SHOW_FAILED_ONLY` is treated as `TEST_QUIET`, and everything above `TEST_SHOW_ALL` is treated as `TEST_SHOW_ALL`.

• At the lowest verbosity level, only the score is shown, ie the ratio passed tests/total tests.
• At the medium level, in addition, failed tests display their name, the expected outcome and the outcome they got. This is the initial setting.
• At the highest level of verbosity, each test is reported as passed or failed.

If a file crashes when it should not, this event is reported no matter the verbosity level.

The command line switch "-failed" causes verbosity to be set to medium at startup. The command line switch "-all" causes verbosity to be set to high at startup.

See Also: test_report

set_wait_on_summary

requests the test report to pause before exiting.

Signature:

```
set_wait_on_summary(integer to_wait)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `to_wait` : an integer, zero not to wait, nonzero to wait.

Comments: Depending on the environment, the test results may be invisible if `set_wait_on_summary(1)` was not called prior, as this is not the default. The command line switch "-wait" performs this call.

See Also: test_report

test_equal

records whether a test passes by comparing two values.

Signature:

```
test_equal(sequence name, object expected, object outcome)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test
≡ `expected` : an object, the expected outcome of some action
≡ `outcome` : an object, some actual value that should equal the reference `expected`.

Comments:

• For floating point numbers, a fuzz of 1e-9 is used to assess equality.

A test is recorded as passed if equality holds between `expected` and `outcome`. The latter is typically a function call, or a variable that was set by some prior action.

While `expected` and `outcome` are processed symmetrically, they are not recorded symmetrically, so be careful to pass `expected` before `outcome` for better test failure reports.

See Also: test_not_equal, test_true, test_false, test_pass, test_fail

test_fail

records that a test failed.

Signature:

```
test_fail(sequence name)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test

See Also: test_equal, test_not_equal, test_true, test_false, test_pass

test_false

records whether a test passes by comparing two values.

Signature:

```
test_false(sequence name, object outcome)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test
≡ `outcome` : an object, some actual value that should be zero

Comments: This assumes an expected value of 0. No fuzz is applied when checking whether an atom is zero or not. Use test_equal() instead in this case.

See Also: test_equal, test_not_equal, test_true, test_pass, test_fail

test_not_equal

records whether a test passes by comparing two values.

Signature:

```
test_not_equal(sequence name, object a, object b)
```

```
public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test
≡ `expected` : an object, the expected outcome of some action
≡ `outcome` : an object, some actual value that should equal the reference `expected`.

Comments:

- For atoms, a fuzz of 1e-9 is used to assess equality.
- For sequences, no such fuzz is implemented.

A test is recorded as passed if equality does not hold between `expected` and `outcome`. The latter is typically a function call, or a variable that was set by some prior action.

See Also: test_equal, test_true, test_false, test_pass, test_fail

test_pass

records that a test passed.

Signature:

```
test_pass(sequence name)


public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test

See Also: test_equal, test_not_equal,test_true, test_false, test_fail

test_report

outputs the test report.

Signature:

```
test_report()


public procedure
include unittest.e
namespace unittest
```

Comments: The report components are described in the comments section for set_test_verbosity. Everything prints on the standard error device.

See Also: set_test_verbosity

test_true

records whether a test passes.

Signature:

```
test_true(sequence name, object outcome)


public procedure
include unittest.e
namespace unittest
```

Arguments: ≡ `name` : a string, the name of the test
≡ `outcome` : an object, some actual value that should not be zero.

Comments: This assumes an expected value different from 0. No fuzz is applied when checking whether an atom is zero or not. Use test_equal() instead in this case.

See Also: test_equal, test_not_equal, test_false, test_pass, test_fail

utils

Routines
iif

utils API

iif

embeds a conditional test inside an expression; `iif` stands for inline if or immediate if.

Signature:

```
iif(atom test, object ifTrue, object ifFalse)
```

```
public function
include utils.e
namespace utils
```

Arguments:
≡ `test` : an atom, the result of a boolean expression
≡ `ifTrue` : an object, returned if `test` is **non-zero**
≡ `ifFalse` : an object, returned if `test` is zero

Returns: An object. Either `ifTrue` or `ifFalse` is returned depending on the value of `test`.

Warning:

You must take care when using this function because (just like all other Euphoria routines) it does not do any *lazy evaluation*.

All argument expressions are evaluated **before** the function is called, thus, it cannot be used when one of the parameters could fail to evaluate correctly.

Example 1:

The reason for this is that both `var[1]` and `var` will be evaluated. Therefore if `var` happens to be an atom, the `var[1]` statement will fail.

```
first = iif(sequence(var), var[1], var)
```

Example 2:

```
if sequence(var) then
     first = var[1]
  else
     first = var
  end if
```

Example 3:

```
msg = sprintf("%s: %s", {
    iif(ErrType = 'E', "Fatal error", "Warning"),
    errortext
})
```

wildcard

Routines

[is_match]

wildcard API

is_match

tests if a string matches a pattern.

Signature:

```
is_match(sequence pattern, sequence string)
```

```
public function
include wildcard.e
namespace wildcard
```

Arguments: ≡ `pattern` : a string, the pattern to match.
≡ `string` : the string to be matched against.

Returns: An **integer**, TRUE if `string` matches `pattern`, else FALSE.

Comments: The pattern may contain * and ? wildcards, but there is currently no way search for literal * or ? characters in a pattern.

Character comparisons are case sensitive. If you want case insensitive comparisons then convert (using either `lower` or `upper`) the arguments (`pattern` and `string`) to lower or upper case .

If you want to detect a pattern anywhere within a string, add * to each end of the pattern:

i = is_match('*' & pattern & '*', string)

See Also: upper, lower

Example 1:

```
i = is_match("A?B*", "AQBXXYY")
-- i is 1 (TRUE)
```

Example 2:

```
i = is_match("*xyz*", "AAAbbbxyz")
-- i is 1 (TRUE)
```

Example 3:

```
i = is_match("A*B*C", "a111b222c")
-- i is 0 (FALSE) because upper/lower case doesn't match
```

Example 4:

```
/demo/search
```