



Open

Euphoria

The programming language

4.0 ver Beta

Euphoria v4.0 svn3379

Table of Contents

0.0.0.1 ?.....	1
Example 1:.....	1
1 Constants.....	1
1.0.0.1 ADD.....	115
1.0.0.2 ADDR_ADDRESS.....	115
2 Routines.....	115
2.0.0.1 abort.....	667
Parameters:.....	title
Subject and Routine Index.....	title



1 Constants

1.0.0.1 ADD

```
include std/map.e  
public enum ADD
```

1.0.0.2 ADDR_ADDRESS

```
include std/net/dns.e  
public enum ADDR_ADDRESS
```

1.0.0.3 ADDR_FAMILY

```
include std/net/dns.e  
public enum ADDR_FAMILY
```

1.0.0.4 ADDR_FLAGS

```
include std/net/dns.e  
public enum ADDR_FLAGS
```

1.0.0.5 ADDR_PROTOCOL

```
include std/net/dns.e  
public enum ADDR_PROTOCOL
```

1.0.0.6 ADDR_TYPE

```
include std/net/dns.e  
public enum ADDR_TYPE
```



1.0.0.7 ADD_APPEND

```
include std/sequence.e
public enum ADD_APPEND
```

1.0.0.8 ADD_PREPEND

```
include std/sequence.e
public enum ADD_PREPEND
```

1.0.0.9 ADD_SORT_DOWN

```
include std/sequence.e
public enum ADD_SORT_DOWN
```

1.0.0.10 ADD_SORT_UP

```
include std/sequence.e
public enum ADD_SORT_UP
```

1.0.0.11 ADLER32

```
include std/map.e
public enum ADLER32
```

1.0.0.12 AF_APPLETALK

```
include std/socket.e
public constant AF_APPLETALK
```

Appletalk

1.0.0.13 AF_BTH

```
include std/socket.e
public constant AF_BTH
```

Bluetooth

1.0.0.14 AF_INET

```
include std/socket.e
public constant AF_INET
```

IPv4 Internet protocols

1.0.0.15 AF_INET6

```
include std/socket.e
public constant AF_INET6
```

IPv6 Internet protocols

1.0.0.16 AF_UNIX

```
include std/socket.e
public constant AF_UNIX
```

Local communications

1.0.0.17 AF_UNSPEC

```
include std/socket.e
public constant AF_UNSPEC
```

Address family is unspecified

1.0.0.18 ANCHORED

```
public constant ANCHORED
```

Forces matches to be only from the first place it is asked to try to make a search. In C, this is called PCRE_ANCHORED. This is passed to all routines including `new`.



1.0.0.19 ANY_UP

```
include std/mouse.e
public integer ANY_UP
```

1.0.0.20 APPEND

```
include std/map.e
public enum APPEND
```

1.0.0.21 ASCENDING

```
include std/sort.e
public constant ASCENDING
```

ascending sort order, always the default.

When a sequence is sorted in `ASCENDING` order, its first element is the smallest as per the sort order and its last element is the largest

1.0.0.22 AT_EXPANSION

```
include std/cmdline.e
public enum AT_EXPANSION
```

Expand arguments that begin with '@' into the command line. (default) For example, @filename will expand the contents of file named 'filename' as if the file's contents were passed in on the command line. Arguments that come after the first extra will not be expanded when `NO_VALIDATION_AFTER_FIRST_EXTRA` is specified.

1.0.0.23 AUTO_CALLOUT

```
public constant AUTO_CALLOUT
```

In C, this is called `PCRE_AUTO_CALLOUT`. To get the functionality of this flag in EUPHORIA, you can use: `find_replace_callback` without passing this option. This is passed to `new`.



1.0.0.24 A_EXECUTE

```
include std/memconst.e
export integer A_EXECUTE
```

1.0.0.25 A_WRITE

```
include std/memconst.e
export integer A_WRITE
```

1.0.0.26 BAD_FILE

```
include std/eds.e
public enum BAD_FILE
```

bad file

1.0.0.27 BAD_RECNO

```
include std/eds.e
public enum BAD_RECNO
```

unknown key_location index was supplied.

1.0.0.28 BAD_SEEK

```
include std/eds.e
public enum BAD_SEEK
```

seek() failed.

1.0.0.29 BINARY_MODE

```
include std/io.e
public enum BINARY_MODE
```



1.0.0.30 BK_LEN

```
include std/sequence.e
public enum BK_LEN
```

Indicates that `size` parameter is maximum length of sub-sequence. See [breakup](#)

1.0.0.31 BK_PIECES

```
include std/sequence.e
public enum BK_PIECES
```

Indicates that `size` parameter is maximum number of sub-sequence. See [breakup](#)

1.0.0.32 BLACK

```
include std/graphcst.e
public constant BLACK
```

1.0.0.33 BLINKING

```
include std/graphcst.e
public constant BLINKING
```

Add to color to get blinking text

1.0.0.34 BLOCK_CURSOR

```
include std/console.e
public constant BLOCK_CURSOR
```

1.0.0.35 BLUE

```
include std/graphcst.e
public constant BLUE
```

1.0.0.36 BMP_INVALID_MODE

```
include std/graphcst.e
public enum BMP_INVALID_MODE
```

1.0.0.37 BMP_OPEN_FAILED

```
include std/graphcst.e
public enum BMP_OPEN_FAILED
```

1.0.0.38 BMP_SUCCESS

```
include std/graphcst.e
public enum BMP_SUCCESS
```

1.0.0.39 BMP_UNEXPECTED_EOF

```
include std/graphcst.e
public enum BMP_UNEXPECTED_EOF
```

1.0.0.40 BMP_UNSUPPORTED_FORMAT

```
include std/graphcst.e
public enum BMP_UNSUPPORTED_FORMAT
```

1.0.0.41 BORDER_SPACE

```
include std/memory.e
export constant BORDER_SPACE
```

1.0.0.42 BORDER_SPACE

```
include std/safe.e
export constant BORDER_SPACE
```



1.0.0.43 BRIGHT_BLUE

```
include std/graphcst.e  
public constant BRIGHT_BLUE
```

1.0.0.44 BRIGHT_CYAN

```
include std/graphcst.e  
public constant BRIGHT_CYAN
```

1.0.0.45 BRIGHT_GREEN

```
include std/graphcst.e  
public constant BRIGHT_GREEN
```

1.0.0.46 BRIGHT_MAGENTA

```
include std/graphcst.e  
public constant BRIGHT_MAGENTA
```

1.0.0.47 BRIGHT_RED

```
include std/graphcst.e  
public constant BRIGHT_RED
```

1.0.0.48 BRIGHT_WHITE

```
include std/graphcst.e  
public constant BRIGHT_WHITE
```

1.0.0.49 BROWN

```
include std/graphcst.e  
public constant BROWN
```

1.0.0.50 BSR_ANYCRLF

```
public constant BSR_ANYCRLF
```

With this option only ASCII new line sequences are recognized as newlines. Other UNICODE newline sequences (encoded as UTF8) are not recognized as an end of line marker. This is passed to all routines including **new**.

1.0.0.51 BSR_UNICODE

```
public constant BSR_UNICODE
```

With this option any UNICODE new line sequence is recognized as a newline. The UNICODE will have to be encoded as UTF8, however. This is passed to all routines including **new**.

1.0.0.52 BYTES_PER_CHAR

```
include std/graphcst.e
public constant BYTES_PER_CHAR
```

1.0.0.53 BYTES_PER_SECTOR

```
include std/filesys.e
public enum BYTES_PER_SECTOR
```

1.0.0.54 CASELESS

```
public constant CASELESS
```

This will make your regular expression matches case insensitive. With this flag for example, [a-z] is the same as [A-Za-z]. This is passed to **new**.

1.0.0.55 CHILD

```
include std/pipeio.e
public enum CHILD
```

Set of pipes that are given to the child - should not be used by the parent

1.0.0.56 CMD_SWITCHES

```
include std/os.e
public constant CMD_SWITCHES
```

1.0.0.57 COMBINE_SORTED

```
include std/sequence.e
public enum COMBINE_SORTED
```

1.0.0.58 COMBINE_UNSORTED

```
include std/sequence.e
public enum COMBINE_UNSORTED
```

1.0.0.59 CONCAT

```
include std/map.e
public enum CONCAT
```

1.0.0.60 COUNT_DIRS

```
include std/filesys.e
public enum COUNT_DIRS
```

1.0.0.61 COUNT_FILES

```
include std/filesys.e
public enum COUNT_FILES
```



1.0.0.62 COUNT_SIZE

```
include std/filesys.e
public enum COUNT_SIZE
```

1.0.0.63 COUNT_TYPES

```
include std/filesys.e
public enum COUNT_TYPES
```

1.0.0.64 CS_FIRST

```
include std/types.e
public enum CS_FIRST
```

Predefined character sets:

1.0.0.65 CYAN

```
include std/graphcst.e
public constant CYAN
```

1.0.0.66 C_BOOL

```
include std/dll.e
public constant C_BOOL
```

bool 32-bits

1.0.0.67 C_BYTE

```
include std/dll.e
public constant C_BYTE
```

byte 8-bits



1.0.0.68 C_CHAR

```
include std/dll.e  
public constant C_CHAR
```

char 8-bits

1.0.0.69 C_DOUBLE

```
include std/dll.e  
public constant C_DOUBLE
```

double 64-bits

1.0.0.70 C_DWORD

```
include std/dll.e  
public constant C_DWORD
```

dword 32-bits

1.0.0.71 C_DWORDLONG

```
include std/dll.e  
public constant C_DWORDLONG
```

dwordlong 64-bits

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

1.0.0.72 C_FLOAT

```
include std/dll.e  
public constant C_FLOAT
```

float 32-bits



1.0.0.73 C_HANDLE

```
include std/dll.e  
public constant C_HANDLE
```

handle 32-bits

1.0.0.74 C_HRESULT

```
include std/dll.e  
public constant C_HRESULT
```

hresult 32-bits

1.0.0.75 C_HWND

```
include std/dll.e  
public constant C_HWND
```

hwnd 32-bits

1.0.0.76 C_INT

```
include std/dll.e  
public constant C_INT
```

int 32-bits

1.0.0.77 C_LONG

```
include std/dll.e  
public constant C_LONG
```

long 32-bits

1.0.0.78 C_LPARAM

```
include std/dll.e  
public constant C_LPARAM
```



lparam 32-bits

1.0.0.79 C_POINTER

```
include std/dll.e  
public constant C_POINTER
```

any valid pointer 32-bits

1.0.0.80 C_SHORT

```
include std/dll.e  
public constant C_SHORT
```

short 16-bits

1.0.0.81 C_SIZE_T

```
include std/dll.e  
public constant C_SIZE_T
```

size_t 32-bits

1.0.0.82 C_UBYTE

```
include std/dll.e  
public constant C_UBYTE
```

ubyte 8-bits

1.0.0.83 C_UCHAR

```
include std/dll.e  
public constant C_UCHAR
```

unsigned char 8-bits



1.0.0.84 C_UINT

```
include std/dll.e
public constant C_UINT
```

unsigned int 32-bits

1.0.0.85 C_ULONG

```
include std/dll.e
public constant C_ULONG
```

unsigned long 32-bits

1.0.0.86 C_USHORT

```
include std/dll.e
public constant C_USHORT
```

unsigned short 16-bits

1.0.0.87 C_WORD

```
include std/dll.e
public constant C_WORD
```

word 16-bits

1.0.0.88 C_WPARAM

```
include std/dll.e
public constant C_WPARAM
```

wparam 32-bits

1.0.0.89 DB_EXISTS_ALREADY

```
include std/eds.e
public constant DB_EXISTS_ALREADY
```

The database could not be created, it already exists.

1.0.0.90 DB_FATAL_FAIL

```
include std/eds.e
public constant DB_FATAL_FAIL
```

A fatal error has occurred.

1.0.0.91 DB_LOCK_EXCLUSIVE

```
include std/eds.e
public enum DB_LOCK_EXCLUSIVE
```

Open the database with read and write access.

1.0.0.92 DB_LOCK_FAIL

```
include std/eds.e
public constant DB_LOCK_FAIL
```

A lock could not be gained on the database.

1.0.0.93 DB_LOCK_NO

```
include std/eds.e
public enum DB_LOCK_NO
```

Do not lock the file.

1.0.0.94 DB_LOCK_SHARED

```
include std/eds.e
public enum DB_LOCK_SHARED
```

Open the database with read-only access.

1.0.0.95 DB_OK

```
include std/eds.e
public constant DB_OK
```

Database is OK, not error has occurred.

1.0.0.96 DB_OPEN_FAIL

```
include std/eds.e
public constant DB_OPEN_FAIL
```

The database could not be opened.

1.0.0.97 DEFAULT

```
public constant DEFAULT
```

This is a value used for not setting any flags at all. This can be passed to all routines including **new**

1.0.0.98 DEGREES_TO_RADIANS

```
include std/mathcons.e
public constant DEGREES_TO_RADIANS
```

Conversion factor: Degrees to Radians = $\text{PI} / 180$

1.0.0.99 DEP_on

```
include std/machine.e
public procedure DEP_on(integer value)
```

1.0.0.100 DEP_really_works

```
include std/memconst.e
export integer DEP_really_works
```

1.0.0.101 DESCENDING

```
include std/sort.e
public constant DESCENDING
```

descending sort order, which is the reverse of ASCENDING.

1.0.0.102 DFA_RESTART

```
public constant DFA_RESTART
```

This is NOT used by any standard library routine.

1.0.0.103 DFA_SHORTEST

```
public constant DFA_SHORTEST
```

This is NOT used by any standard library routine.

1.0.0.104 DISPLAY_ASCII

```
include std/pretty.e
public enum DISPLAY_ASCII
```

1.0.0.105 DIVIDE

```
include std/map.e
public enum DIVIDE
```

1.0.0.106 DNS_QUERY_ACCEPT_TRUNCATED_RESPONSE

```
include std/net/dns.e
public constant DNS_QUERY_ACCEPT_TRUNCATED_RESPONSE
```



1.0.0.107 DNS_QUERY_BYPASS_CACHE

```
include std/net/dns.e
public constant DNS_QUERY_BYPASS_CACHE
```

1.0.0.108 DNS_QUERY_DONT_RESET_TTL_VALUES

```
include std/net/dns.e
public constant DNS_QUERY_DONT_RESET_TTL_VALUES
```

1.0.0.109 DNS_QUERY_NO_HOSTS_FILE

```
include std/net/dns.e
public constant DNS_QUERY_NO_HOSTS_FILE
```

1.0.0.110 DNS_QUERY_NO_LOCAL_NAME

```
include std/net/dns.e
public constant DNS_QUERY_NO_LOCAL_NAME
```

1.0.0.111 DNS_QUERY_NO_NETBT

```
include std/net/dns.e
public constant DNS_QUERY_NO_NETBT
```

1.0.0.112 DNS_QUERY_NO_RECURSION

```
include std/net/dns.e
public constant DNS_QUERY_NO_RECURSION
```

1.0.0.113 DNS_QUERY_NO_WIRE_QUERY

```
include std/net/dns.e
public constant DNS_QUERY_NO_WIRE_QUERY
```

1.0.0.114 DNS_QUERY_RESERVED

```
include std/net/dns.e
public constant DNS_QUERY_RESERVED
```

1.0.0.115 DNS_QUERY_RETURN_MESSAGE

```
include std/net/dns.e
public constant DNS_QUERY_RETURN_MESSAGE
```

1.0.0.116 DNS_QUERY_STANDARD

```
include std/net/dns.e
public constant DNS_QUERY_STANDARD
```

1.0.0.117 DNS_QUERY_TREAT_AS_FQDN

```
include std/net/dns.e
public constant DNS_QUERY_TREAT_AS_FQDN
```

1.0.0.118 DNS_QUERY_USE_TCP_ONLY

```
include std/net/dns.e
public constant DNS_QUERY_USE_TCP_ONLY
```

1.0.0.119 DNS_QUERY_WIRE_ONLY

```
include std/net/dns.e
public constant DNS_QUERY_WIRE_ONLY
```

1.0.0.120 DOLLAR_ENDONLY

```
public constant DOLLAR_ENDONLY
```

If this bit is set, a dollar sign metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar sign also matches immediately before a newline at the end of the string (but not before any other newlines). Thus you must include the newline character in the pattern before the dollar sign if you want to match a line that contains a newline character. The DOLLAR_ENDONLY option is ignored if

MULTILINE is set. There is no way to set this option within a pattern. This is passed to **new**.

1.0.0.121 DOS_TEXT

```
include std/io.e
public enum DOS_TEXT
```

1.0.0.122 DOTALL

```
public constant DOTALL
```

With this option the '.' character also matches a newline sequence. This is passed to **new**.

1.0.0.123 DUPNAMES

```
public constant DUPNAMES
```

Allow duplicate names for named subpatterns. Since there is no way to access named subpatterns this flag has no effect. This is passed to **new**.

1.0.0.124 DUP_TABLE

```
include std/eds.e
public enum DUP_TABLE
```

this table already exists.

1.0.0.125 D_ALTNAME

```
include std/filesys.e
public enum D_ALTNAME
```

1.0.0.126 D_ATTRIBUTES

```
include std/filesys.e
public enum D_ATTRIBUTES
```

1.0.0.127 D_DAY

```
include std/filesys.e
public enum D_DAY
```

1.0.0.128 D_HOUR

```
include std/filesys.e
public enum D_HOUR
```

1.0.0.129 D_MILLISECOND

```
include std/filesys.e
public enum D_MILLISECOND
```

1.0.0.130 D_MINUTE

```
include std/filesys.e
public enum D_MINUTE
```

1.0.0.131 D_MONTH

```
include std/filesys.e
public enum D_MONTH
```

1.0.0.132 D_NAME

```
include std/filesys.e
public enum D_NAME
```

1.0.0.133 D_SECOND

```
include std/filesys.e
public enum D_SECOND
```

1.0.0.134 D_SIZE

```
include std/filesys.e
public enum D_SIZE
```

1.0.0.135 D_YEAR

```
include std/filesys.e
public enum D_YEAR
```

1.0.0.136 E

```
include std/mathcons.e
public constant E
```

Euler (e)The base of the natural logarithm.

1.0.0.137 EOF

```
include std/io.e
public constant EOF
```

End of file

1.0.0.138 EOL

```
public constant EOL
```

All platform's newline character: '\n'. When text lines are read the native platform's EOLSEP string is replaced by a single character EOL.

1.0.0.139 EOLSEP

```
public constant EOLSEP
```

Current platform's newline string: "\n" on *Unix*, else "\r\n".



1.0.0.140 ERROR_BADCOUNT

```
include std/regex.e
public constant ERROR_BADCOUNT
```

1.0.0.141 ERROR_BADMAGIC

```
include std/regex.e
public constant ERROR_BADMAGIC
```

1.0.0.142 ERROR_BADNEWLINE

```
include std/regex.e
public constant ERROR_BADNEWLINE
```

1.0.0.143 ERROR_BADOPTION

```
include std/regex.e
public constant ERROR_BADOPTION
```

1.0.0.144 ERROR_BADPARTIAL

```
include std/regex.e
public constant ERROR_BADPARTIAL
```

1.0.0.145 ERROR_BADUTF8

```
include std/regex.e
public constant ERROR_BADUTF8
```

1.0.0.146 ERROR_BADUTF8_OFFSET

```
include std/regex.e
public constant ERROR_BADUTF8_OFFSET
```



1.0.0.147 ERROR_CALLOUT

```
include std/regex.e  
public constant ERROR_CALLOUT
```

1.0.0.148 ERROR_DFA_RECURSE

```
include std/regex.e  
public constant ERROR_DFA_RECURSE
```

1.0.0.149 ERROR_DFA_UCOND

```
include std/regex.e  
public constant ERROR_DFA_UCOND
```

1.0.0.150 ERROR_DFA_UITEM

```
include std/regex.e  
public constant ERROR_DFA_UITEM
```

1.0.0.151 ERROR_DFA_UMLIMIT

```
include std/regex.e  
public constant ERROR_DFA_UMLIMIT
```

1.0.0.152 ERROR_DFA_WSSIZE

```
include std/regex.e  
public constant ERROR_DFA_WSSIZE
```

1.0.0.153 ERROR_INTERNAL

```
include std/regex.e  
public constant ERROR_INTERNAL
```

1.0.0.154 ERROR_MATCHLIMIT

```
include std/regex.e
public constant ERROR_MATCHLIMIT
```

1.0.0.155 ERROR_NOMATCH

```
include std/regex.e
public constant ERROR_NOMATCH
```

1.0.0.156 ERROR_NOMEMORY

```
include std/regex.e
public constant ERROR_NOMEMORY
```

1.0.0.157 ERROR_NOSUBSTRING

```
include std/regex.e
public constant ERROR_NOSUBSTRING
```

1.0.0.158 ERROR_NULL

```
include std/regex.e
public constant ERROR_NULL
```

1.0.0.159 ERROR_NULLWSLIMIT

```
include std/regex.e
public constant ERROR_NULLWSLIMIT
```

1.0.0.160 ERROR_PARTIAL

```
include std/regex.e
public constant ERROR_PARTIAL
```



1.0.0.161 ERROR_RECURSIONLIMIT

```
include std/regex.e
public constant ERROR_RECURSIONLIMIT
```

1.0.0.162 ERROR_UNKNOWN_NODE

```
include std/regex.e
public constant ERROR_UNKNOWN_NODE
```

1.0.0.163 ERROR_UNKNOWN_OPCODE

```
include std/regex.e
public constant ERROR_UNKNOWN_OPCODE
```

1.0.0.164 ERR_ACCESS

```
include std/socket.e
public constant ERR_ACCESS
```

Permission has been denied. This can happen when using a `send_to` call on a broadcast address without setting the socket option `SO_BROADCAST`. Another, possibly more common, reason is you have tried to bind an address that is already exclusively bound by another application.

May occur on a Unix Domain Socket when the socket directory or file could not be accessed due to security.

1.0.0.165 ERR_ADDRINUSE

```
include std/socket.e
public constant ERR_ADDRINUSE
```

Address is already in use.

1.0.0.166 ERR_ADDRNOTAVAIL

```
include std/socket.e
public constant ERR_ADDRNOTAVAIL
```

The specified address is not a valid local IP address on this computer.

1.0.0.167 ERR_AFNOSUPPORT

```
include std/socket.e
public constant ERR_AFNOSUPPORT
```

Address family not supported by the protocol family.

1.0.0.168 ERR_AGAIN

```
include std/socket.e
public constant ERR_AGAIN
```

Kernel resources to complete the request are temporarily unavailable.

1.0.0.169 ERR_ALREADY

```
include std/socket.e
public constant ERR_ALREADY
```

Operation is already in progress.

1.0.0.170 ERR_CLOSE_CHAR

```
include tokenize.e
public enum ERR_CLOSE_CHAR
```

1.0.0.171 ERR_CONNABORTED

```
include std/socket.e
public constant ERR_CONNABORTED
```

Software has caused a connection to be aborted.

1.0.0.172 ERR_CONNREFUSED

```
include std/socket.e
public constant ERR_CONNREFUSED
```



Connection was refused.

1.0.0.173 ERR_CONNRESET

```
include std/socket.e
public constant ERR_CONNRESET
```

An incoming connection was supplied however it was terminated by the remote peer.

1.0.0.174 ERR_DECIMAL

```
include tokenize.e
public enum ERR_DECIMAL
```

1.0.0.175 ERR_DESTADDRREQ

```
include std/socket.e
public constant ERR_DESTADDRREQ
```

Destination address required.

1.0.0.176 ERR_EOF

```
include tokenize.e
public enum ERR_EOF
```

1.0.0.177 ERR_EOF_STRING

```
include tokenize.e
public enum ERR_EOF_STRING
```

1.0.0.178 ERR_EOL_CHAR

```
include tokenize.e
public enum ERR_EOL_CHAR
```



1.0.0.179 ERR_EOL_STRING

```
include tokenize.e
public enum ERR_EOL_STRING
```

1.0.0.180 ERR_ESCAPE

```
include tokenize.e
public enum ERR_ESCAPE
```

1.0.0.181 ERR_FAULT

```
include std/socket.e
public constant ERR_FAULT
```

Address creation has failed internally.

1.0.0.182 ERR_HEX

```
include tokenize.e
public enum ERR_HEX
```

1.0.0.183 ERR_HEX_STRING

```
include tokenize.e
public enum ERR_HEX_STRING
```

1.0.0.184 ERR_HOSTUNREACH

```
include std/socket.e
public constant ERR_HOSTUNREACH
```

No route to the host specified could be found.

1.0.0.185 ERR_INPROGRESS

```
include std/socket.e
public constant ERR_INPROGRESS
```

1.0.0.2 ADDR_ADDRESS

A blocking call is in progress.

1.0.0.186 ERR_INTR

```
include std/socket.e
public constant ERR_INTR
```

A blocking call was cancelled or interrupted.

1.0.0.187 ERR_INVALID

```
include std/socket.e
public constant ERR_INVALID
```

An invalid sequence of command calls were made, for instance trying to `accept` before an actual `listen` was called.

1.0.0.188 ERR_IO

```
include std/socket.e
public constant ERR_IO
```

An I/O error occurred while making the directory entry or allocating the inode. (Unix Domain Socket).

1.0.0.189 ERR_ISCONN

```
include std/socket.e
public constant ERR_ISCONN
```

Socket is already connected.

1.0.0.190 ERR_ISDIR

```
include std/socket.e
public constant ERR_ISDIR
```

An empty pathname was specified. (Unix Domain Socket).

1.0.0.191 ERR_LOOP

```
include std/socket.e
public constant ERR_LOOP
```

Too many symbolic links were encountered. (Unix Domain Socket).

1.0.0.192 ERR_MFILE

```
include std/socket.e
public constant ERR_MFILE
```

The queue is not empty upon routine call.

1.0.0.193 ERR_MSGSIZE

```
include std/socket.e
public constant ERR_MSGSIZE
```

Message is too long for buffer size. This would indicate an internal error to Euphoria as Euphoria sets a dynamic buffer size.

1.0.0.194 ERR_NAMETOOLONG

```
include std/socket.e
public constant ERR_NAMETOOLONG
```

Component of the path name exceeded 255 characters or the entire path exceeded 1023 characters. (Unix Domain Socket).

1.0.0.195 ERR_NETDOWN

```
include std/socket.e
public constant ERR_NETDOWN
```

The network subsystem is down or has failed



1.0.0.196 ERR_NETRESET

```
include std/socket.e  
public constant ERR_NETRESET
```

Network has dropped it's connection on reset.

1.0.0.197 ERR_NETUNREACH

```
include std/socket.e  
public constant ERR_NETUNREACH
```

Network is unreachable.

1.0.0.198 ERR_NFILE

```
include std/socket.e  
public constant ERR_NFILE
```

Not a file. (Unix Domain Sockets).

1.0.0.199 ERR_NOBUFS

```
include std/socket.e  
public constant ERR_NOBUFS
```

No buffer space is available.

1.0.0.200 ERR_NOENT

```
include std/socket.e  
public constant ERR_NOENT
```

Named socket does not exist. (Unix Domain Socket).

1.0.0.201 ERR_NOTCONN

```
include std/socket.e  
public constant ERR_NOTCONN
```



Socket is not connected.

1.0.0.202 ERR_NOTDIR

```
include std/socket.e  
public constant ERR_NOTDIR
```

Component of the path prefix is not a directory. (Unix Domain Socket).

1.0.0.203 ERR_NOTINITIALISED

```
include std/socket.e  
public constant ERR_NOTINITIALISED
```

Socket system is not initialized (Windows only)

1.0.0.204 ERR_NOTSOCK

```
include std/socket.e  
public constant ERR_NOTSOCK
```

The descriptor is not a socket.

1.0.0.205 ERR_OPEN

```
include tokenize.e  
public enum ERR_OPEN
```

1.0.0.206 ERR_OPNOTSUPP

```
include std/socket.e  
public constant ERR_OPNOTSUPP
```

Operation is not supported on this type of socket.

1.0.0.207 ERR_PROTONOSUPPORT

```
include std/socket.e
public constant ERR_PROTONOSUPPORT
```

Protocol not supported.

1.0.0.208 ERR_PROTOTYPE

```
include std/socket.e
public constant ERR_PROTOTYPE
```

Protocol is the wrong type for the socket.

1.0.0.209 ERR_ROFS

```
include std/socket.e
public constant ERR_ROFS
```

The name would reside on a read-only file system. (Unix Domain Socket).

1.0.0.210 ERR_SHUTDOWN

```
include std/socket.e
public constant ERR_SHUTDOWN
```

The socket has been shutdown. Possibly a send/receive call after a shutdown took place.

1.0.0.211 ERR_SOCKTNOSUPPORT

```
include std/socket.e
public constant ERR_SOCKTNOSUPPORT
```

Socket type is not supported.

1.0.0.212 ERR_TIMEDOUT

```
include std/socket.e
public constant ERR_TIMEDOUT
```



Connection has timed out.

1.0.0.213 ERR_UNKNOWN

```
include tokenize.e
public enum ERR_UNKNOWN
```

1.0.0.214 ERR_WOULDBLOCK

```
include std/socket.e
public constant ERR_WOULDBLOCK
```

The operation would block on a socket marked as non-blocking.

These values are passed as the `family` and `sock_type` parameters of the `create` function.

1.0.0.215 ET_ERROR

```
include tokenize.e
public constant ET_ERROR
```

1.0.0.216 ET_ERR_COLUMN

```
include tokenize.e
public constant ET_ERR_COLUMN
```

1.0.0.217 ET_ERR_LINE

```
include tokenize.e
public constant ET_ERR_LINE
```

1.0.0.218 ET_TOKENS

```
include tokenize.e
public constant ET_TOKENS
```

1.0.0.219 EULER_GAMMA

```
include std/mathcons.e
public constant EULER_GAMMA
```

Gamma (Euler Gamma)

1.0.0.220 EXTENDED

```
public constant EXTENDED
```

Whitespace and characters beginning with a hash mark to the end of the line in the pattern will be ignored when searching except when the whitespace or hash is escaped or in a character class. This is passed to **new**.

1.0.0.221 EXTRA

```
public constant EXTRA
```

When an alphanumeric follows a backslash(\) has no special meaning an error is generated. This is passed to **new**.

1.0.0.222 EXT_COUNT

```
include std/filesys.e
public enum EXT_COUNT
```

1.0.0.223 EXT_NAME

```
include std/filesys.e
public enum EXT_NAME
```

1.0.0.224 EXT_SIZE

```
include std/filesys.e
public enum EXT_SIZE
```



1.0.0.225 E_ATOM

```
include std/dll.e
public constant E_ATOM
```

atom

1.0.0.226 E_INTEGER

```
include std/dll.e
public constant E_INTEGER
```

integer

1.0.0.227 E_OBJECT

```
include std/dll.e
public constant E_OBJECT
```

object

1.0.0.228 E_SEQUENCE

```
include std/dll.e
public constant E_SEQUENCE
```

sequence

1.0.0.229 FALSE

```
include std/types.e
public constant FALSE
```

Boolean FALSE value



1.0.0.230 FIFO

```
include std/stack.e
public constant FIFO
```

Stack types

- FIFO: like people standing in line: first item in is first item out
 - FILO: like for a stack of plates : first item in is last item out
-
-

1.0.0.231 FILETYPE_DIRECTORY

```
include std/filesys.e
public enum FILETYPE_DIRECTORY
```

1.0.0.232 FILETYPE_FILE

```
include std/filesys.e
public enum FILETYPE_FILE
```

1.0.0.233 FILETYPE_NOT_FOUND

```
include std/filesys.e
public enum FILETYPE_NOT_FOUND
```

1.0.0.234 FILETYPE_UNDEFINED

```
include std/filesys.e
public enum FILETYPE_UNDEFINED
```

1.0.0.235 FIRSTLINE

```
public constant FIRSTLINE
```

If `PCRE_FIRSTLINE` is set, the match must happen before or at the first newline in the subject (though it may continue over the newline). This is passed to `new`.

1.0.0.236 FLETCHER32

```
include std/map.e
public enum FLETCHER32
```

1.0.0.237 FP_FORMAT

```
include std/pretty.e
public enum FP_FORMAT
```

1.0.0.238 FREEBSD

```
include std/os.e
public enum FREEBSD
```

These constants are returned by the `platform` function.

- WIN32 -- Host operating system is Windows
- LINUX -- Host operating system is Linux
- FREEBSD -- Host operating system is FreeBSD
- OSX -- Host operating system is Mac OS X
- SUNOS -- Host operating system is Sun's OpenSolaris
- OPENBSD -- Host operating system is OpenBSD
- NETBSD -- Host operating system is NetBSD

Note:

Via the `platform` call, there is no way to determine if you are on Linux or FreeBSD. This was done to provide a generic UNIX return value for `platform`.

In most situations you are better off to test the host platform by using the `ifdef statement`. It is both more precise and faster.

1.0.0.239 FREE_BYTES

```
include std/filesys.e
public enum FREE_BYTES
```

**1.0.0.240 FREE_RID**

```
include std/memconst.e
export integer FREE_RID public enum A_READ
```

1.0.0.241 GET_EOF

```
include std/get.e
public constant GET_EOF
```

1.0.0.242 GET_FAIL

```
include std/get.e
public constant GET_FAIL
```

1.0.0.243 GET_LONG_ANSWER

```
include std/get.e
public constant GET_LONG_ANSWER
```

1.0.0.244 GET_NOTHING

```
include std/get.e
public constant GET_NOTHING
```

1.0.0.245 GET_SHORT_ANSWER

```
include std/get.e
public constant GET_SHORT_ANSWER
```

1.0.0.246 GET_SUCCESS

```
include std/get.e
public constant GET_SUCCESS
```

**1.0.0.247 GRAY**

```
include std/graphcst.e
public constant GRAY
```

1.0.0.248 GREEN

```
include std/graphcst.e
public constant GREEN
```

1.0.0.249 GetLastError_rid

```
include std/memconst.e
export atom GetLastError_rid
```

1.0.0.250 GetSystemInfo_rid

```
include std/memconst.e
export atom GetSystemInfo_rid
```

1.0.0.251 HALFPI

```
include std/mathcons.e
public constant HALFPI
```

Half of PI

1.0.0.252 HALFSQRT2

```
include std/mathcons.e
public constant HALFSQRT2
```

$\sqrt{2}/2$



1.0.0.253 HALF_BLOCK_CURSOR

```
include std/console.e
public constant HALF_BLOCK_CURSOR
```

1.0.0.254 HAS_CASE

```
include std/cmdline.e
public constant HAS_CASE
```

This option switch is case sensitive. See [cmd_parse](#)

1.0.0.255 HAS_PARAMETER

```
include std/cmdline.e
public constant HAS_PARAMETER
```

This option switch does have a parameter. See [cmd_parse](#)

1.0.0.256 HELP

```
include std/cmdline.e
public constant HELP
```

This option switch triggers the 'help' display. See [cmd_parse](#)

1.0.0.257 HELP_RID

```
include std/cmdline.e
public enum HELP_RID
```

Additional help routine id. See [cmd_parse](#)

1.0.0.258 HOST_ALIASES

```
include std/net/dns.e
public enum HOST_ALIASES
```



1.0.0.259 HOST_IPS

```
include std/net/dns.e
public enum HOST_IPS
```

1.0.0.260 HOST_OFFICIAL_NAME

```
include std/net/dns.e
public enum HOST_OFFICIAL_NAME
```

1.0.0.261 HOST_TYPE

```
include std/net/dns.e
public enum HOST_TYPE
```

1.0.0.262 HSIEH32

```
include std/map.e
public enum HSIEH32
```

1.0.0.263 HTTP_HEADER_ACCEPT

```
include std/net/http.e
public constant HTTP_HEADER_ACCEPT
```

1.0.0.264 HTTP_HEADER_ACCEPTCHARSET

```
include std/net/http.e
public constant HTTP_HEADER_ACCEPTCHARSET
```

1.0.0.265 HTTP_HEADER_ACCEPTENCODING

```
include std/net/http.e
public constant HTTP_HEADER_ACCEPTENCODING
```



1.0.0.266 HTTP_HEADER_ACCEPTLANGUAGE

```
include std/net/http.e
public constant HTTP_HEADER_ACCEPTLANGUAGE
```

1.0.0.267 HTTP_HEADER_ACCEPTRANGES

```
include std/net/http.e
public constant HTTP_HEADER_ACCEPTRANGES
```

1.0.0.268 HTTP_HEADER_AUTHORIZATION

```
include std/net/http.e
public constant HTTP_HEADER_AUTHORIZATION
```

1.0.0.269 HTTP_HEADER_CACHECONTROL

```
include std/net/http.e
public constant HTTP_HEADER_CACHECONTROL
```

1.0.0.270 HTTP_HEADER_CONNECTION

```
include std/net/http.e
public constant HTTP_HEADER_CONNECTION
```

1.0.0.271 HTTP_HEADER_CONTENTLENGTH

```
include std/net/http.e
public constant HTTP_HEADER_CONTENTLENGTH
```

1.0.0.272 HTTP_HEADER_CONTENTTYPE

```
include std/net/http.e
public constant HTTP_HEADER_CONTENTTYPE
```



1.0.0.273 HTTP_HEADER_DATE

```
include std/net/http.e
public constant HTTP_HEADER_DATE
```

1.0.0.274 HTTP_HEADER_FROM

```
include std/net/http.e
public constant HTTP_HEADER_FROM
```

1.0.0.275 HTTP_HEADER_GET

```
include std/net/http.e
public constant HTTP_HEADER_GET
```

1.0.0.276 HTTP_HEADER_HOST

```
include std/net/http.e
public constant HTTP_HEADER_HOST
```

1.0.0.277 HTTP_HEADER_HTTPVERSION

```
include std/net/http.e
public constant HTTP_HEADER_HTTPVERSION
```

1.0.0.278 HTTP_HEADER_IFMODIFIEDSINCE

```
include std/net/http.e
public constant HTTP_HEADER_IFMODIFIEDSINCE
```

1.0.0.279 HTTP_HEADER_KEEPALIVE

```
include std/net/http.e
public constant HTTP_HEADER_KEEPALIVE
```



1.0.0.280 HTTP_HEADER_POST

```
include std/net/http.e
public constant HTTP_HEADER_POST
```

1.0.0.281 HTTP_HEADER_POSTDATA

```
include std/net/http.e
public constant HTTP_HEADER_POSTDATA
```

1.0.0.282 HTTP_HEADER_REFERER

```
include std/net/http.e
public constant HTTP_HEADER_REFERER
```

1.0.0.283 HTTP_HEADER_USERAGENT

```
include std/net/http.e
public constant HTTP_HEADER_USERAGENT
```

1.0.0.284 IDABORT

```
include std/win32/msgbox.e
public constant IDABORT
```

Abort button was selected.

1.0.0.285 IDCANCEL

```
include std/win32/msgbox.e
public constant IDCANCEL
```

Cancel button was selected.

1.0.0.286 IDIGNORE

```
include std/win32/msgbox.e
public constant IDIGNORE
```

1.0.0.2 ADDR_ADDRESS



Ignore button was selected.

1.0.0.287 IDNO

```
include std/win32/msgbox.e  
public constant IDNO
```

No button was selected.

1.0.0.288 IDOK

```
include std/win32/msgbox.e  
public constant IDOK
```

OK button was selected.

1.0.0.289 IDRETRY

```
include std/win32/msgbox.e  
public constant IDRETRY
```

Retry button was selected.

1.0.0.290 IDYES

```
include std/win32/msgbox.e  
public constant IDYES
```

Yes button was selected.

1.0.0.291 INDENT

```
include std/pretty.e  
public enum INDENT
```

1.0.0.292 INSERT_FAILED

```
include std/eds.e
public enum INSERT_FAILED
```

couldn't insert a new record.

1.0.0.293 INT_FORMAT

```
include std/pretty.e
public enum INT_FORMAT
```

1.0.0.294 INVALID_ROUTINE_ID

```
include std/types.e
public constant INVALID_ROUTINE_ID
```

value returned from `routine_id()` when the routine doesn't exist or is out of scope. this is typically seen as -1 in legacy code.

1.0.0.295 INVLN10

```
include std/mathcons.e
public constant INVLN10
```

$1 / \ln(10)$

1.0.0.296 INVLN2

```
include std/mathcons.e
public constant INVLN2
```

$1 / (\ln(2))$

1.0.0.297 INVSQ2PI

```
include std/mathcons.e
public constant INVSQ2PI
```

$1 / (\text{sqrt}(2\text{PI}))$

1.0.0.298 LAST_ERROR_CODE

```
include std/eds.e
public enum LAST_ERROR_CODE
```

last error code

1.0.0.299 LEAVE

```
include std/map.e
public enum LEAVE
```

1.0.0.300 LEFT_DOWN

```
include std/mouse.e
public integer LEFT_DOWN
```

1.0.0.301 LEFT_UP

```
include std/mouse.e
public integer LEFT_UP
```

1.0.0.302 LINE_BREAKS

```
include std/pretty.e
public enum LINE_BREAKS
```

1.0.0.303 LINUX

```
include std/os.e
public enum LINUX
```

1.0.0.304 LN10

```
include std/mathcons.e
public constant LN10
```

$\ln(10) :: 10 = \text{power}(E, \text{LN10})$

1.0.0.305 LN2

```
include std/mathcons.e
public constant LN2
```

$\ln(2) :: 2 = \text{power}(E, \text{LN2})$

1.0.0.306 LOCK_EXCLUSIVE

```
include std/io.e
public enum LOCK_EXCLUSIVE
```

1.0.0.307 LOCK_SHARED

```
include std/io.e
public enum LOCK_SHARED
```

1.0.0.308 MAGENTA

```
include std/graphcst.e
public constant MAGENTA
```

1.0.0.309 MANDATORY

```
include std/cmdline.e
public constant MANDATORY
```

This option switch must be supplied on command line. See [cmd_parse](#)



1.0.0.310 MAP_ANONYMOUS

```
include std/unix/mmap.e  
public constant MAP_ANONYMOUS
```

1.0.0.311 MAP_FILE

```
include std/unix/mmap.e  
public constant MAP_FILE
```

1.0.0.312 MAP_FIXED

```
include std/unix/mmap.e  
public constant MAP_FIXED
```

1.0.0.313 MAP_PRIVATE

```
include std/unix/mmap.e  
public constant MAP_PRIVATE
```

1.0.0.314 MAP_SHARED

```
include std/unix/mmap.e  
public constant MAP_SHARED
```

1.0.0.315 MAP_TYPE

```
include std/unix/mmap.e  
public constant MAP_TYPE
```

1.0.0.316 MAX_ASCII

```
include std/pretty.e  
public enum MAX_ASCII
```



1.0.0.317 MAX_LINES

```
include std/pretty.e
public enum MAX_LINES
```

1.0.0.318 MB_ABORTRETRYIGNORE

```
include std/win32/msgbox.e
public constant MB_ABORTRETRYIGNORE
```

Abort, Retry, Ignore

1.0.0.319 MB_APPLMODAL

```
include std/win32/msgbox.e
public constant MB_APPLMODAL
```

User must respond before doing something else

1.0.0.320 MB_DEFAULT_DESKTOP_ONLY

```
include std/win32/msgbox.e
public constant MB_DEFAULT_DESKTOP_ONLY
```

1.0.0.321 MB_DEFBUTTON1

```
include std/win32/msgbox.e
public constant MB_DEFBUTTON1
```

First button is default button

1.0.0.322 MB_DEFBUTTON2

```
include std/win32/msgbox.e
public constant MB_DEFBUTTON2
```

Second button is default button

1.0.0.323 MB_DEFBUTTON3

```
include std/win32/msgbox.e
public constant MB_DEFBUTTON3
```

Third button is default button

1.0.0.324 MB_DEFBUTTON4

```
include std/win32/msgbox.e
public constant MB_DEFBUTTON4
```

Fourth button is default button

1.0.0.325 MB_HELP

```
include std/win32/msgbox.e
public constant MB_HELP
```

Windows 95: Help button generates help event

1.0.0.326 MB_ICONASTERISK

```
include std/win32/msgbox.e
public constant MB_ICONASTERISK
```

1.0.0.327 MB_ICONERROR

```
include std/win32/msgbox.e
public constant MB_ICONERROR
```

1.0.0.328 MB_ICONEXCLAMATION

```
include std/win32/msgbox.e
public constant MB_ICONEXCLAMATION
```

Exclamation-point appears in the box

1.0.0.329 MB_ICONHAND

```
include std/win32/msgbox.e  
public constant MB_ICONHAND
```

A hand appears

1.0.0.330 MB_ICONINFORMATION

```
include std/win32/msgbox.e  
public constant MB_ICONINFORMATION
```

Lowercase letter i in a circle appears

1.0.0.331 MB_ICONQUESTION

```
include std/win32/msgbox.e  
public constant MB_ICONQUESTION
```

A question-mark icon appears

1.0.0.332 MB_ICONSTOP

```
include std/win32/msgbox.e  
public constant MB_ICONSTOP
```

1.0.0.333 MB_ICONWARNING

```
include std/win32/msgbox.e  
public constant MB_ICONWARNING
```

1.0.0.334 MB_OK

```
include std/win32/msgbox.e  
public constant MB_OK
```

Message box contains one push button: OK



1.0.0.335 MB_OKCANCEL

```
include std/win32/msgbox.e
public constant MB_OKCANCEL
```

Message box contains OK and Cancel

1.0.0.336 MB_RETRYCANCEL

```
include std/win32/msgbox.e
public constant MB_RETRYCANCEL
```

Message box contains Retry and Cancel

1.0.0.337 MB_RIGHT

```
include std/win32/msgbox.e
public constant MB_RIGHT
```

Windows 95: The text is right-justified

1.0.0.338 MB_RTLCREADING

```
include std/win32/msgbox.e
public constant MB_RTLCREADING
```

Windows 95: For Hebrew and Arabic systems

1.0.0.339 MB_SERVICE_NOTIFICATION

```
include std/win32/msgbox.e
public constant MB_SERVICE_NOTIFICATION
```

Windows NT: The caller is a service

1.0.0.340 MB_SETFOREGROUND

```
include std/win32/msgbox.e
public constant MB_SETFOREGROUND
```



Message box becomes the foreground window

1.0.0.341 MB_SYSTEMMODAL

```
include std/win32/msgbox.e
public constant MB_SYSTEMMODAL
```

All applications suspended until user responds

1.0.0.342 MB_TASKMODAL

```
include std/win32/msgbox.e
public constant MB_TASKMODAL
```

Similar to MB_APPLMODAL

1.0.0.343 MB_YESNO

```
include std/win32/msgbox.e
public constant MB_YESNO
```

Message box contains Yes and No

1.0.0.344 MB_YESNOCANCEL

```
include std/win32/msgbox.e
public constant MB_YESNOCANCEL
```

Message box contains Yes, No, and Cancel

possible values returned by MessageBox(). 0 means failure

1.0.0.345 MD5

```
include std/map.e
public enum MD5
```



1.0.0.346 MEM_COMMIT

```
include std/memconst.e
export constant MEM_COMMIT
```

1.0.0.347 MEM_RELEASE

```
include std/memconst.e
export constant MEM_RELEASE
```

1.0.0.348 MEM_RESERVE

```
include std/memconst.e
export constant MEM_RESERVE
```

1.0.0.349 MEM_RESET

```
include std/memconst.e
export constant MEM_RESET
```

1.0.0.350 MIDDLE_DOWN

```
include std/mouse.e
public integer MIDDLE_DOWN
```

1.0.0.351 MIDDLE_UP

```
include std/mouse.e
public integer MIDDLE_UP
```

1.0.0.352 MINF

```
include std/mathcons.e
public constant MINF
```

Negative Infinity

1.0.0.353 MIN_ASCII

```
include std/pretty.e
public enum MIN_ASCII
```

1.0.0.354 MISSING_END

```
include std/eds.e
public enum MISSING_END
```

Missing 0 terminator

1.0.0.355 MOVE

```
include std/mouse.e
public integer MOVE
```

1.0.0.356 MSG_CONFIRM

```
include std/socket.e
public constant MSG_CONFIRM
```

Tell the link layer that forward progress happened: you got a successful reply from the other side. If the link layer doesn't get this it will regularly reprobe the neighbor (e.g., via a unicast ARP). Only valid on **SOCK_DGRAM** and **SOCK_RAW** sockets and currently only implemented for IPv4 and IPv6.

1.0.0.357 MSG_CTRUNC

```
include std/socket.e
public constant MSG_CTRUNC
```

indicates that some control data were discarded due to lack of space in the buffer for ancillary data.

1.0.0.358 MSG_DONTROUTE

```
include std/socket.e
public constant MSG_DONTROUTE
```

Don't use a gateway to send out the packet, only send to hosts on directly connected networks. This is usually used only by diagnostic or routing programs. This is only defined for protocol families that route; packet



sockets don't.

1.0.0.359 MSG_DONTWAIT

```
include std/socket.e
public constant MSG_DONTWAIT
```

Enables non-blocking operation; if the operation would block, EAGAIN or EWOULDBLOCK is returned.

1.0.0.360 MSG_EOR

```
include std/socket.e
public constant MSG_EOR
```

Terminates a record (when this notion is supported, as for sockets of type **SOCK_SEQPACKET**).

1.0.0.361 MSG_ERRQUEUE

```
include std/socket.e
public constant MSG_ERRQUEUE
```

indicates that no data was received but an extended error from the socket error queue.

1.0.0.362 MSG_FIN

```
include std/socket.e
public constant MSG_FIN
```

1.0.0.363 MSG_MORE

```
include std/socket.e
public constant MSG_MORE
```

The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the TCP_CORK socket option, with the difference that this flag can be set on a per-call basis.

1.0.0.364 MSG_NOSIGNAL

```
include std/socket.e
public constant MSG_NOSIGNAL
```

Requests not to send SIGPIPE on errors on stream oriented sockets when the other end breaks the connection. The EPIPE error is still returned.

1.0.0.365 MSG_OOB

```
include std/socket.e
public constant MSG_OOB
```

Sends out-of-band data on sockets that support this notion (e.g., of type **SOCK_STREAM**); the underlying protocol must also support out-of-band data.

1.0.0.366 MSG_PEEK

```
include std/socket.e
public constant MSG_PEEK
```

This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

1.0.0.367 MSG_PROXY

```
include std/socket.e
public constant MSG_PROXY
```

1.0.0.368 MSG_RST

```
include std/socket.e
public constant MSG_RST
```

1.0.0.369 MSG_SYN

```
include std/socket.e
public constant MSG_SYN
```

1.0.0.370 MSG_TRUNC

```
include std/socket.e
public constant MSG_TRUNC
```

indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

1.0.0.371 MSG_TRYHARD

```
include std/socket.e
public constant MSG_TRYHARD
```

1.0.0.372 MSG_WAITALL

```
include std/socket.e
public constant MSG_WAITALL
```

This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

1.0.0.373 MULTILINE

```
public constant MULTILINE
```

When MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is passed to [new](#).

1.0.0.374 MULTIPLE

```
include std/cmdline.e
public constant MULTIPLE
```

This option switch may occur multiple times on a command line. See [cmd_parse](#)

1.0.0.375 MULTIPLY

```
include std/map.e
public enum MULTIPLY
```

1.0.0.376 M_ALLOC

```
include std/memconst.e
export constant M_ALLOC
```

1.0.0.377 M_FREE

```
include std/memconst.e
export constant M_FREE
```

1.0.0.378 NESTED_ALL

```
include std/search.e
public constant NESTED_ALL
```

1.0.0.379 NESTED_ANY

```
include std/search.e
public constant NESTED_ANY
```

1.0.0.380 NESTED_BACKWARD

```
include std/search.e
public constant NESTED_BACKWARD
```

1.0.0.381 NESTED_INDEX

```
include std/search.e
public constant NESTED_INDEX
```

1.0.0.382 NETBSD

```
include std/os.e
public enum NETBSD
```

1.0.0.383 NEWLINE_ANY

```
public constant NEWLINE_ANY
```

Sets ANY newline sequence as the NEWLINE sequence including those from UNICODE when UTF8 is also set. The string will have to be encoded as UTF8, however. The NEWLINE sequence will match \$ when MULTILINE is set. This is passed to all routines including **new**.

1.0.0.384 NEWLINE_ANYCRLF

```
public constant NEWLINE_ANYCRLF
```

Sets ANY newline sequence from ASCII. The NEWLINE sequence will match \$ when MULTILINE is set. This is passed to all routines including **new**.

1.0.0.385 NEWLINE_CR

```
public constant NEWLINE_CR
```

Sets CR as the NEWLINE sequence. The NEWLINE sequence will match \$ when MULTILINE is set. This is passed to all routines including **new**.

1.0.0.386 NEWLINE_CRLF

```
public constant NEWLINE_CRLF
```

Sets CRLF as the NEWLINE sequence. The NEWLINE sequence will match \$ when MULTILINE is set. This is passed to all routines including **new**.

1.0.0.387 NEWLINE_LF

```
public constant NEWLINE_LF
```

Sets LF as the NEWLINE sequence. The NEWLINE sequence will match \$ when MULTILINE is set. This is passed to all routines including `new`.

1.0.0.388 NORMAL_ORDER

```
include std/sort.e
public constant NORMAL_ORDER
```

The normal sort order used by the custom comparison routine.

1.0.0.389 NOTBOL

```
public constant NOTBOL
```

This indicates that beginning of the passed string does **NOT** start at the **B**eginning **O**f a **L**ine (NOTBOL), so a carrot symbol (^) in the original pattern will not match the beginning of the string. This is used by routines other than `new`.

1.0.0.390 NOTEMPTY

```
public constant NOTEMPTY
```

Here matches of empty strings will not be allowed. In C, this is PCRE_NOTEMPTY. The pattern: ``A*a*`` will match "AAAA", "aaaa", and "Aaaa" but not "". This is used by routines other than `new`.

1.0.0.391 NOTEOL

```
public constant NOTEOL
```

This indicates that end of the passed string does **NOT** end at the **E**nd **O**f a **L**ine (NOTEOL), so a dollar sign (\$) in the original pattern will not match the end of the string. This is used by routines other than `new`.

1.0.0.392 NO_AT_EXPANSION

```
include std/cmdline.e
public enum NO_AT_EXPANSION
```

Do not expand arguments that begin with '@' into the command line. Normally `@filename` will expand the file names contents as if the file's contents were passed in on the command line. This option supresses this

behavior.

1.0.0.393 NO_AUTO_CAPTURE

```
public constant NO_AUTO_CAPTURE
```

Disables capturing subpatterns except when the subpatterns are named. This is passed to [new](#).

1.0.0.394 NO_CASE

```
include std/cmdline.e  
public constant NO_CASE
```

This option switch is not case sensitive. See [cmd_parse](#)

1.0.0.395 NO_CURSOR

```
include std/console.e  
public constant NO_CURSOR
```

1.0.0.396 NO_DATABASE

```
include std/eds.e  
public enum NO_DATABASE
```

current_db is not set

1.0.0.397 NO_HELP

```
include std/cmdline.e  
public constant NO_HELP
```

1.0.0.398 NO_PARAMETER

```
include std/cmdline.e  
public constant NO_PARAMETER
```

This option switch does not have a parameter. See [cmd_parse](#)

1.0.0.399 NO_ROUTINE_ID

```
include std/types.e
public constant NO_ROUTINE_ID
```

to be used as a flag for no `routine_id()` supplied.

1.0.0.400 NO_TABLE

```
include std/eds.e
public enum NO_TABLE
```

no table was found.

1.0.0.401 NO_UTF8_CHECK

```
public constant NO_UTF8_CHECK
```

Turn off checking for the validity of your UTF string. Use this with caution. An invalid utf8 string with this option could **crash** your program. Only use this if you know the string is a valid utf8 string. See `unicode:validate`. This is passed to all routines including `new`.

1.0.0.402 NO_VALIDATION

```
include std/cmdline.e
public enum NO_VALIDATION
```

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

1.0.0.403 NO_VALIDATION_AFTER_FIRST_EXTRA

```
include std/cmdline.e
public enum NO_VALIDATION_AFTER_FIRST_EXTRA
```

Do not cause an error for an invalid parameter after the first extra item has been found. This can be helpful for processes such as the Interpreter itself that must deal with command line parameters that it is not meant to handle. At expansions after the first extra are also disabled.

For instance:

```
eui -D TEST greet.ex -name John -greeting Bye -D TEST
```

is meant for eui, but -name and -greeting options are meant for greet.ex. See [cmd_parse](#)

```
eui @euopts.txt greet.ex @hotmail.com here 'hotmail.com'
```

is not expanded into the command line but 'euopts.txt' is.

1.0.0.404 NS_C_ANY

```
include std/net/dns.e
public constant NS_C_ANY
```

1.0.0.405 NS_C_IN

```
include std/net/dns.e
public constant NS_C_IN
```

1.0.0.406 NS_KT_DH

```
include std/net/dns.e
public constant NS_KT_DH
```

1.0.0.407 NS_KT_DSA

```
include std/net/dns.e
public constant NS_KT_DSA
```

1.0.0.408 NS_KT_PRIVATE

```
include std/net/dns.e
public constant NS_KT_PRIVATE
```

1.0.0.409 NS_KT_RSA

```
include std/net/dns.e  
public constant NS_KT_RSA
```

1.0.0.410 NS_T_A

```
include std/net/dns.e  
public constant NS_T_A
```

1.0.0.411 NS_T_A6

```
include std/net/dns.e  
public constant NS_T_A6
```

1.0.0.412 NS_T_AAAA

```
include std/net/dns.e  
public constant NS_T_AAAA
```

1.0.0.413 NS_T_ANY

```
include std/net/dns.e  
public constant NS_T_ANY
```

1.0.0.414 NS_T_MX

```
include std/net/dns.e  
public constant NS_T_MX
```

1.0.0.415 NS_T_NS

```
include std/net/dns.e  
public constant NS_T_NS
```

1.0.0.416 NS_T_PTR

```
include std/net/dns.e
public constant NS_T_PTR
```

1.0.0.417 NULL

```
include std/dll.e
public constant NULL
```

C's NULL pointer

1.0.0.418 NULLDEVICE

```
public constant NULLDEVICE
```

Current platform's null device path: `/dev/null` on *Unix*, else `NUL:.`

1.0.0.419 NUMBER_OF_FREE_CLUSTERS

```
include std/filesys.e
public enum NUMBER_OF_FREE_CLUSTERS
```

1.0.0.420 NUM_ENTRIES

```
include std/map.e
public enum NUM_ENTRIES
```

Retrieves characteristics of a map.

Parameters:

1. `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])
```

1.0.0.421 OBJ_ATOM

```
include std/types.e
public constant OBJ_ATOM
```

Object is atom

1.0.0.422 OBJ_INTEGER

```
include std/types.e
public constant OBJ_INTEGER
```

Object is integer

1.0.0.423 OBJ_SEQUENCE

```
include std/types.e
public constant OBJ_SEQUENCE
```

Object is sequence

1.0.0.424 OBJ_UNASSIGNED

```
include std/types.e
public constant OBJ_UNASSIGNED
```

Object not assigned



1.0.0.425 OK

```
include std/socket.e
public constant OK
```

No error occurred.

1.0.0.426 ONCE

```
include std/cmdline.e
public constant ONCE
```

This option switch must only occur once on the command line. See [cmd_parse](#)

1.0.0.427 OPENBSD

```
include std/os.e
public enum OPENBSD
```

1.0.0.428 OPTIONAL

```
include std/cmdline.e
public constant OPTIONAL
```

This option switch does not have to be on command line. See [cmd_parse](#)

1.0.0.429 OPT_CNT

```
include std/cmdline.e
public enum OPT_CNT
```

The number of times that the routine has been called by `cmd_parse` for this option. See [cmd_parse](#)

1.0.0.430 OPT_IDX

```
include std/cmdline.e
public enum OPT_IDX
```

An index into the `opts` list. See [cmd_parse](#)

1.0.0.431 OPT_REV

```
include std/cmdline.e
public enum OPT_REV
```

The value 1 if the command line indicates that this option is to remove any earlier occurrences of it. See [cmd_parse](#)

1.0.0.432 OPT_VAL

```
include std/cmdline.e
public enum OPT_VAL
```

The option's value as found on the command line. See [cmd_parse](#)

1.0.0.433 OSX

```
include std/os.e
public enum OSX
```

1.0.0.434 PAGE_EXECUTE

```
include std/memconst.e
public constant PAGE_EXECUTE
```

You may run the data in this page

1.0.0.435 PAGE_EXECUTE_READ

```
include std/memconst.e
public constant PAGE_EXECUTE_READ
```

You may read or run the data



1.0.0.436 PAGE_EXECUTE_READWRITE

```
include std/memconst.e
public constant PAGE_EXECUTE_READWRITE
```

You may run, read or write in this page

1.0.0.437 PAGE_EXECUTE_WRITECOPY

```
include std/memconst.e
public constant PAGE_EXECUTE_WRITECOPY
```

You may run or write in this page

1.0.0.438 PAGE_NOACCESS

```
include std/memconst.e
public constant PAGE_NOACCESS
```

You have no access to this page

1.0.0.439 PAGE_NONE

```
include std/memconst.e
public constant PAGE_NONE
```

You have no access to this page An alias to PAGE_NOACCESS

1.0.0.440 PAGE_READ

```
include std/memconst.e
public constant PAGE_READ
```

You may only read to this page An alias to PAGE_READONLY

1.0.0.441 PAGE_READONLY

```
include std/memconst.e
public constant PAGE_READONLY
```

1.0.0.2 ADDR_ADDRESS

You may only read data in this page

1.0.0.442 PAGE_READWRITE

```
include std/memconst.e
public constant PAGE_READWRITE
```

You may read or write in this page.

1.0.0.443 PAGE_READ_EXECUTE

```
include std/memconst.e
public constant PAGE_READ_EXECUTE
```

You may read or run the data An alias to PAGE_EXECUTE_READ

1.0.0.444 PAGE_READ_WRITE

```
include std/memconst.e
public constant PAGE_READ_WRITE
```

You may read or write to this page An alias to PAGE_READWRITE

1.0.0.445 PAGE_READ_WRITE_EXECUTE

```
include std/memconst.e
public constant PAGE_READ_WRITE_EXECUTE
```

You may run, read or write in this page An alias to PAGE_EXECUTE_READWRITE

1.0.0.446 PAGE_SIZE

```
include std/machine.e
public constant PAGE_SIZE
```

1.0.0.447 PAGE_WRITECOPY

```
include std/memconst.e
public constant PAGE_WRITECOPY
```

You may write to this page.

1.0.0.448 PAGE_WRITE_COPY

```
include std/memconst.e
public constant PAGE_WRITE_COPY
```

You may write to this page. Data will copied for use with other processes when you first write to it.

1.0.0.449 PAGE_WRITE_EXECUTE_COPY

```
include std/memconst.e
public constant PAGE_WRITE_EXECUTE_COPY
```

You may run or write to this page. Data will copied for use with other processes when you first write to it.

1.0.0.450 PARENT

```
include std/pipeio.e
public enum PARENT
```

Set of pipes that are for the use of the parent

1.0.0.451 PARTIAL

```
public constant PARTIAL
```

This option has no effect with these routines. Refer to the C documentation for what it does in C. In C, this constant is called PCRE_PARTIAL. This is used by routines other than *new*.

1.0.0.452 PATHSEP

```
public constant PATHSEP
```

Current platform's path separator character: `:` on *Unix*, else `;`.

1.0.0.453 PATH_BASENAME

```
include std/filesys.e
public enum PATH_BASENAME
```

1.0.0.454 PATH_DIR

```
include std/filesys.e
public enum PATH_DIR
```

1.0.0.455 PATH_DRIVEID

```
include std/filesys.e
public enum PATH_DRIVEID
```

1.0.0.456 PATH_FILEEXT

```
include std/filesys.e
public enum PATH_FILEEXT
```

1.0.0.457 PATH_FILENAME

```
include std/filesys.e
public enum PATH_FILENAME
```

1.0.0.458 PAUSE_MSG

```
include std/cmdline.e
public enum PAUSE_MSG
```

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

1.0.0.459 PHI

```
include std/mathcons.e
public constant PHI
```

phi => Golden Ratio = $(1 + \sqrt{5}) / 2$

1.0.0.460 PI

```
include std/mathcons.e
public constant PI
```

PI is the ratio of a circle's circumference to it's diameter.

$PI = C / D :: C = PI * D :: C = PI * 2 * R(\text{radius})$

1.0.0.461 PID

```
include std/pipeio.e
public enum PID
```

Process ID

1.0.0.462 PINF

```
include std/mathcons.e
public constant PINF
```

Positive Infinity

1.0.0.463 PISQR

```
include std/mathcons.e
public constant PISQR
```

PI^2

1.0.0.464 PRETTY_DEFAULT

```
include std/pretty.e
public constant PRETTY_DEFAULT
```

1.0.0.465 PROT_EXEC

```
include std/unix/mmap.e
public constant PROT_EXEC
```

1.0.0.466 PROT_NONE

```
include std/unix/mmap.e
public constant PROT_NONE
```

1.0.0.467 PROT_READ

```
include std/unix/mmap.e
public constant PROT_READ
```

1.0.0.468 PROT_WRITE

```
include std/unix/mmap.e
public constant PROT_WRITE
```

1.0.0.469 PUT

```
include std/map.e
public enum PUT
```

1.0.0.470 QUARTPI

```
include std/mathcons.e
public constant QUARTPI
```

Quarter of PI

1.0.0.471 RADIANS_TO_DEGREES

```
include std/mathcons.e
public constant RADIANS_TO_DEGREES
```

Conversion factor: Radians to Degrees = $180 / \text{PI}$

1.0.0.472 RD_INPLACE

```
include std/sequence.e
public enum RD_INPLACE
```

These are used with the `remove_dups()` function.

- ♦ 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.
-

1.0.0.473 RED

```
include std/graphcst.e
public constant RED
```

1.0.0.474 REVERSE_ORDER

```
include std/sort.e
public constant REVERSE_ORDER
```

Reverses the sense of the order returned by a custom comparison routine.

1.0.0.475 RIGHT_DOWN

```
include std/mouse.e
public integer RIGHT_DOWN
```

1.0.0.476 RIGHT_UP

```
include std/mouse.e
public integer RIGHT_UP
```

1.0.0.477 ROTATE_LEFT

```
include std/sequence.e
public constant ROTATE_LEFT
```

1.0.0.478 ROTATE_RIGHT

```
include std/sequence.e
public constant ROTATE_RIGHT
```

1.0.0.479 SCREEN

```
include std/io.e
public constant SCREEN
```

Screen (Standard Out)

1.0.0.480 SD_BOTH

```
include std/socket.e
public constant SD_BOTH
```

Shutdown both send and receive operations.

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.



1.0.0.481 SD_RECEIVE

```
include std/socket.e
public constant SD_RECEIVE
```

Shutdown the receive operations.

1.0.0.482 SD_SEND

```
include std/socket.e
public constant SD_SEND
```

Shutdown the send operations.

1.0.0.483 SECTORS_PER_CLUSTER

```
include std/filesys.e
public enum SECTORS_PER_CLUSTER
```

1.0.0.484 SELECT_IS_ERROR

```
include std/socket.e
public enum SELECT_IS_ERROR
```

Boolean (1/0) value indicating the error state.

Pass one of the following to the `method` parameter of `shutdown`.

1.0.0.485 SELECT_IS_READABLE

```
include std/socket.e
public enum SELECT_IS_READABLE
```

Boolean (1/0) value indicating the readability.

1.0.0.486 SELECT_IS_WRITABLE

```
include std/socket.e
public enum SELECT_IS_WRITABLE
```

Boolean (1/0) value indicating the writeability.

1.0.0.487 SELECT_SOCKET

```
include std/socket.e
public enum SELECT_SOCKET
```

The socket

1.0.0.488 SEQ_NOALT

```
include std/sequence.e
public constant SEQ_NOALT
```

Indicates that `remove_subseq()` must not replace removed sub-sequences with an alternative value.

1.0.0.489 SHA256

```
include std/map.e
public enum SHA256
```

1.0.0.490 SHARED_LIB_EXT

```
public constant SHARED_LIB_EXT
```

Current platform's shared library extension. For instance it can be `dll`, `so` or `dylib` depending on the platform.

1.0.0.491 SHOW_ONLY_OPTIONS

```
include std/cmdline.e
public enum SHOW_ONLY_OPTIONS
```

Only display the option list in `show_help`. Do not display other information such as program name, options, etc... See `cmd_parse`

1.0.0.492 SIDE_NONE

```
include std/sets.e
public enum SIDE_NONE
```

The following constants denote orientation of distributivity or unitarity:

- SIDE_NONE -- no units, or no distributivity
 - SIDE_LEFT -- property is requested or verified on the left side
 - SIDE_RIGHT -- property is requested or verified on the right side
 - SIDE_BOTH -- property is requested or verified on both sides.
-
-

1.0.0.493 SLASH

```
public constant SLASH
```

Current platform's path separator character

Comments:

When on *Windows*, '\\'. When on *Unix*, '/'.

1.0.0.494 SLASHES

```
public constant SLASHES
```

Current platform's possible path separators. This is slightly different in that on *Windows* the path separators variable contains \\ as well as : and / as newer *Windows* versions support / as a path separator. On *Unix* systems, it only contains /.

1.0.0.495 SMALLMAP

```
include std/map.e
public constant SMALLMAP
```

1.0.0.496 SM_TEXT

```
include std/map.e
public enum SM_TEXT
```

Saves a map to a file.

Parameters:

1. `m` : a map.
2. `file_name_p` : Either a sequence, the name of the file to save to, or an open file handle as returned by `open()`.
3. `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.

The `SM_RAW` type saves the map in an efficient manner. It is generally smaller than the text format and is faster to process, but it is not human readable and standard text editors can not be used to edit it. In this format, the file will

contain three serialized sequences:

1. Header sequence: {integer:format version, string: date and time of save (YYMMDDhhmmss), sequence: euphoria version {major, minor, revision, patch}}
2. Keys. A list of all the keys
3. Values. A list of the corresponding values for the keys.

Example 1:

```
map AppOptions
  if save_map(AppOptions, "c:\myapp\options.txt") = -1
    Error("Failed to save application options")
  end if
  if save_map(AppOptions, "c:\myapp\options.dat", SM_RAW) = -1
    Error("Failed to save application options")
  end if
```

See Also:

[load_map](#)

1.0.0.497 SND_ASTERISK

```
include std/win32/sounds.e
public constant SND_ASTERISK
```

1.0.0.498 SND_DEFAULT

```
include std/win32/sounds.e
public constant SND_DEFAULT
```

1.0.0.499 SND_EXCLAMATION

```
include std/win32/sounds.e
public constant SND_EXCLAMATION
```

1.0.0.500 SND_QUESTION

```
include std/win32/sounds.e
public constant SND_QUESTION
```

1.0.0.501 SND_STOP

```
include std/win32/sounds.e
public constant SND_STOP
```

1.0.0.502 SOCKET_SOCKETADDR_IN

```
include std/socket.e
export enum SOCKET_SOCKETADDR_IN
```

Accessor index for the sockaddr_in pointer of a socket type

1.0.0.503 SOCKET_SOCKET

```
include std/socket.e
export enum SOCKET_SOCKET
```

Accessor index for socket handle of a socket type

1.0.0.504 SOCK_DGRAM

```
include std/socket.e
public constant SOCK_DGRAM
```

Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

1.0.0.505 SOCK_RAW

```
include std/socket.e
public constant SOCK_RAW
```

Provides raw network protocol access.

1.0.0.506 SOCK_RDM

```
include std/socket.e
public constant SOCK_RDM
```

Provides a reliable datagram layer that does not guarantee ordering.

1.0.0.507 SOCK_SEQPACKET

```
include std/socket.e
public constant SOCK_SEQPACKET
```

Obsolete and should not be used in new programs

Use with the result of `select`.

1.0.0.508 SOCK_STREAM

```
include std/socket.e
public constant SOCK_STREAM
```

Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

1.0.0.509 SOL_SOCKET

```
include std/socket.e
public constant SOL_SOCKET
```

1.0.0.510 SO_ACCEPTCONN

```
include std/socket.e
public constant SO_ACCEPTCONN
```

1.0.0.511 SO_BINDTODEVICE

```
include std/socket.e
public constant SO_BINDTODEVICE
```

Pass to the `flags` parameter of `send` and `receive`



1.0.0.512 SO_BROADCAST

```
include std/socket.e  
public constant SO_BROADCAST
```

1.0.0.513 SO_CONNDATA

```
include std/socket.e  
public constant SO_CONNDATA
```

1.0.0.514 SO_CONNDATALEN

```
include std/socket.e  
public constant SO_CONNDATALEN
```

1.0.0.515 SO_CONNOPT

```
include std/socket.e  
public constant SO_CONNOPT
```

1.0.0.516 SO_CONNOPTLEN

```
include std/socket.e  
public constant SO_CONNOPTLEN
```

1.0.0.517 SO_DEBUG

```
include std/socket.e  
public constant SO_DEBUG
```

1.0.0.518 SO_DISCDATA

```
include std/socket.e  
public constant SO_DISCDATA
```



1.0.0.519 SO_DISCDATALEN

```
include std/socket.e
public constant SO_DISCDATALEN
```

1.0.0.520 SO_DISCOPT

```
include std/socket.e
public constant SO_DISCOPT
```

1.0.0.521 SO_DISCOPTLEN

```
include std/socket.e
public constant SO_DISCOPTLEN
```

1.0.0.522 SO_DONTLINGER

```
include std/socket.e
public constant SO_DONTLINGER
```

1.0.0.523 SO_DONTROUTE

```
include std/socket.e
public constant SO_DONTROUTE
```

1.0.0.524 SO_ERROR

```
include std/socket.e
public constant SO_ERROR
```

1.0.0.525 SO_KEEPALIVE

```
include std/socket.e
public constant SO_KEEPALIVE
```



1.0.0.526 SO_LINGER

```
include std/socket.e  
public constant SO_LINGER
```

1.0.0.527 SO_MAXDG

```
include std/socket.e  
public constant SO_MAXDG
```

1.0.0.528 SO_MAXPATHDG

```
include std/socket.e  
public constant SO_MAXPATHDG
```

1.0.0.529 SO_OOINLINE

```
include std/socket.e  
public constant SO_OOINLINE
```

1.0.0.530 SO_OPENTYPE

```
include std/socket.e  
public constant SO_OPENTYPE
```

1.0.0.531 SO_PASSCRED

```
include std/socket.e  
public constant SO_PASSCRED
```

1.0.0.532 SO_PEERCREC

```
include std/socket.e  
public constant SO_PEERCREC
```

**1.0.0.533 SO_RCVBUF**

```
include std/socket.e
public constant SO_RCVBUF
```

1.0.0.534 SO_RCVLOWAT

```
include std/socket.e
public constant SO_RCVLOWAT
```

1.0.0.535 SO_RCVTIMEO

```
include std/socket.e
public constant SO_RCVTIMEO
```

1.0.0.536 SO_REUSEADDR

```
include std/socket.e
public constant SO_REUSEADDR
```

1.0.0.537 SO_REUSEPORT

```
include std/socket.e
public constant SO_REUSEPORT
```

1.0.0.538 SO_SECURITY_AUTHENTICATION

```
include std/socket.e
public constant SO_SECURITY_AUTHENTICATION
```

1.0.0.539 SO_SECURITY_ENCRYPTION_NETWORK

```
include std/socket.e
public constant SO_SECURITY_ENCRYPTION_NETWORK
```



1.0.0.540 SO_SECURITY_ENCRYPTION_TRANSPORT

```
include std/socket.e
public constant SO_SECURITY_ENCRYPTION_TRANSPORT
```

1.0.0.541 SO_SNDBUF

```
include std/socket.e
public constant SO_SNDBUF
```

1.0.0.542 SO_SNDLOWAT

```
include std/socket.e
public constant SO_SNDLOWAT
```

1.0.0.543 SO_SNDTIMEO

```
include std/socket.e
public constant SO_SNDTIMEO
```

1.0.0.544 SO_SYNCHRONOUS_ALERT

```
include std/socket.e
public constant SO_SYNCHRONOUS_ALERT
```

1.0.0.545 SO_SYNCHRONOUS_NONALERT

```
include std/socket.e
public constant SO_SYNCHRONOUS_NONALERT
```

1.0.0.546 SO_TYPE

```
include std/socket.e
public constant SO_TYPE
```



1.0.0.547 SO_USELOOPBACK

```
include std/socket.e
public constant SO_USELOOPBACK
```

1.0.0.548 SQRT2

```
include std/mathcons.e
public constant SQRT2
```

sqrt(2)

1.0.0.549 SQRT3

```
include std/mathcons.e
public constant SQRT3
```

Square root of 3

1.0.0.550 SQRT5

```
include std/mathcons.e
public constant SQRT5
```

sqrt(5)

1.0.0.551 SQRTE

```
include std/mathcons.e
public constant SQRTE
```

sqrt(e)

1.0.0.552 START_COLUMN

```
include std/pretty.e
public enum START_COLUMN
```

1.0.0.553 STDERR

```
include std/io.e
public constant STDERR
```

Standard Error

1.0.0.554 STDERR

```
include std/pipeio.e
public enum STDERR
```

Child processes standard error

1.0.0.555 STDFLTR_ALPHA

```
public constant STDFLTR_ALPHA
```

Predefined routine_id for use with `filter()`.

Comments:

Used to filter out non-alphabetic characters from a string.

Example:

```
-- Collect only the alphabetic characters from 'text'
result = filter(text, STDFLTR_ALPHA)
```

1.0.0.556 STDIN

```
include std/io.e
public constant STDIN
```

Standard Input

1.0.0.557 STDIN

```
include std/pipeio.e
public enum STDIN
```

Child processes standard input

1.0.0.558 STDOUT

```
include std/io.e
public constant STDOUT
```

Standard Output

1.0.0.559 STDOUT

```
include std/pipeio.e
public enum STDOUT
```

Child processes standard output

1.0.0.560 STRING_OFFSETS

```
public constant STRING_OFFSETS
```

This is used by [matches](#) and [all_matches](#).

1.0.0.561 ST_ALLNUM

```
include std/stats.e
public enum ST_ALLNUM
```

The supplied data consists of only atoms.



1.0.0.562 ST_FULLPOP

```
include std/stats.e
public enum ST_FULLPOP
```

The supplied data is the entire population.

1.0.0.563 ST_IGNSTR

```
include std/stats.e
public enum ST_IGNSTR
```

Any sub-sequences (eg. strings) in the supplied data are ignored.

1.0.0.564 ST_SAMPLE

```
include std/stats.e
public enum ST_SAMPLE
```

The supplied data is only a random sample of the population.

1.0.0.565 ST_ZEROSTR

```
include std/stats.e
public enum ST_ZEROSTR
```

Any sub-sequences (eg. strings) in the supplied data are assumed to have the value zero.

1.0.0.566 SUBTRACT

```
include std/map.e
public enum SUBTRACT
```

1.0.0.567 SUNOS

```
include std/os.e
public enum SUNOS
```

1.0.0.568 SyntaxColor

```
include syncolor.e
public function SyntaxColor(sequence pline)
```

1.0.0.569 TDATA

```
include tokenize.e
public enum TDATA
```

1.0.0.570 TEST_QUIET

```
include std/unittest.e
public enum TEST_QUIET
```

1.0.0.571 TEST_SHOW_ALL

```
include std/unittest.e
public enum TEST_SHOW_ALL
```

1.0.0.572 TEST_SHOW_FAILED_ONLY

```
include std/unittest.e
public enum TEST_SHOW_FAILED_ONLY
```

1.0.0.573 TEXT_MODE

```
include std/io.e
public enum TEXT_MODE
```

**1.0.0.574 TFORM**

```
include tokenize.e
public enum TFORM
```

1.0.0.575 TF_ATOM

```
include tokenize.e
public constant TF_ATOM
```

1.0.0.576 TF_HEX

```
include tokenize.e
public constant TF_HEX
```

1.0.0.577 TF_INT

```
include tokenize.e
public constant TF_INT
```

1.0.0.578 THICK_UNDERLINE_CURSOR

```
include std/console.e
public constant THICK_UNDERLINE_CURSOR
```

1.0.0.579 TLNUM

```
include tokenize.e
public enum TLNUM
```

1.0.0.580 TLPOS

```
include tokenize.e
public enum TLPOS
```



1.0.0.581 TOTAL_BYTES

```
include std/filesys.e
public enum TOTAL_BYTES
```

1.0.0.582 TOTAL_NUMBER_OF_CLUSTERS

```
include std/filesys.e
public enum TOTAL_NUMBER_OF_CLUSTERS
```

1.0.0.583 TRUE

```
include std/types.e
public constant TRUE
```

Boolean TRUE value

1.0.0.584 TTYPE

```
include tokenize.e
public enum TTYPE
```

1.0.0.585 TWOPI

```
include std/mathcons.e
public constant TWOPI
```

Two times PI

1.0.0.586 T_BLANK

```
include tokenize.e
public constant T_BLANK
```

1.0.0.587 T_CHAR

```
include tokenize.e
public constant T_CHAR
```

quoted character

1.0.0.588 T_COLON

```
include tokenize.e  
public constant T_COLON
```

1.0.0.589 T_COMMA

```
include tokenize.e  
public constant T_COMMA
```

1.0.0.590 T_COMMENT

```
include tokenize.e  
public constant T_COMMENT
```

1.0.0.591 T_CONCAT

```
include tokenize.e  
public constant T_CONCAT
```

1.0.0.592 T_CONCATEQ

```
include tokenize.e  
public constant T_CONCATEQ
```

1.0.0.593 T_DELIMITER

```
include tokenize.e  
public constant T_DELIMITER
```



1.0.0.594 T_DIVIDE

```
include tokenize.e  
public constant T_DIVIDE
```

1.0.0.595 T_DIVIDEEQ

```
include tokenize.e  
public constant T_DIVIDEEQ
```

1.0.0.596 T_DOLLAR

```
include tokenize.e  
public constant T_DOLLAR
```

1.0.0.597 T_DOUBLE_OPS

```
include tokenize.e  
public constant T_DOUBLE_OPS
```

1.0.0.598 T_EOF

```
include tokenize.e  
public constant T_EOF
```

1.0.0.599 T_EQ

```
include tokenize.e  
public constant T_EQ
```

**1.0.0.600 T_GT**

```
include tokenize.e  
public constant T_GT
```

1.0.0.601 T_GTEQ

```
include tokenize.e  
public constant T_GTEQ
```

1.0.0.602 T_IDENTIFIER

```
include tokenize.e  
public constant T_IDENTIFIER
```

1.0.0.603 T_KEYWORD

```
include tokenize.e  
public constant T_KEYWORD
```

1.0.0.604 T_LBRACE

```
include tokenize.e  
public constant T_LBRACE
```

1.0.0.605 T_LBRACKET

```
include tokenize.e  
public constant T_LBRACKET
```

1.0.0.606 T_LPAREN

```
include tokenize.e  
public constant T_LPAREN
```



1.0.0.607 T_LT

```
include tokenize.e  
public constant T_LT
```

1.0.0.608 T_LTEQ

```
include tokenize.e  
public constant T_LTEQ
```

1.0.0.609 T_MINUS

```
include tokenize.e  
public constant T_MINUS
```

1.0.0.610 T_MINUSEQ

```
include tokenize.e  
public constant T_MINUSEQ
```

1.0.0.611 T_MULTIPLY

```
include tokenize.e  
public constant T_MULTIPLY
```

1.0.0.612 T_MULTIPLYEQ

```
include tokenize.e  
public constant T_MULTIPLYEQ
```

1.0.0.613 T_NOT

```
include tokenize.e  
public constant T_NOT
```

1.0.0.614 T_NOTEQ

```
include tokenize.e
public constant T_NOTEQ
```

1.0.0.615 T_NULL

```
include tokenize.e
public constant T_NULL
```

1.0.0.616 T_NUMBER

```
include tokenize.e
public constant T_NUMBER
```

1.0.0.617 T_PERIOD

```
include tokenize.e
public constant T_PERIOD
```

1.0.0.618 T_PLUS

```
include tokenize.e
public constant T_PLUS
```

1.0.0.619 T_PLUSEQ

```
include tokenize.e
public constant T_PLUSEQ
```

1.0.0.620 T_QPRINT

```
include tokenize.e
public constant T_QPRINT
```

```
quick print ( ? x )
```

**1.0.0.621 T_RBRACE**

```
include tokenize.e  
public constant T_RBRACE
```

1.0.0.622 T_RBRACKET

```
include tokenize.e  
public constant T_RBRACKET
```

1.0.0.623 T_RPAREN

```
include tokenize.e  
public constant T_RPAREN
```

1.0.0.624 T_SHBANG

```
include tokenize.e  
public constant T_SHBANG
```

1.0.0.625 T_SINGLE_OPS

```
include tokenize.e  
public constant T_SINGLE_OPS
```

1.0.0.626 T_SLICE

```
include tokenize.e  
public constant T_SLICE
```

1.0.0.627 T_STRING

```
include tokenize.e  
public constant T_STRING
```

string

1.0.0.628 UNDERLINE_CURSOR

```
include std/console.e
public constant UNDERLINE_CURSOR
```

1.0.0.629 UNGREEDY

```
public constant UNGREEDY
```

This modifier sets the pattern such that quantifiers are not greedy by default, but become greedy if followed by a question mark.

This is passed to **new**.

1.0.0.630 UNIX_TEXT

```
include std/io.e
public enum UNIX_TEXT
```

1.0.0.631 URL_ENTIRE

```
include std/net/common.e
public constant URL_ENTIRE
```

1.0.0.632 URL_HOSTNAME

```
include std/net/url.e
public enum URL_HOSTNAME
```

1.0.0.633 URL_HTTP_DOMAIN

```
include std/net/common.e
public constant URL_HTTP_DOMAIN
```



1.0.0.634 URL_HTTP_PATH

```
include std/net/common.e  
public constant URL_HTTP_PATH
```

1.0.0.635 URL_HTTP_QUERY

```
include std/net/common.e  
public constant URL_HTTP_QUERY
```

1.0.0.636 URL_MAIL_ADDRESS

```
include std/net/common.e  
public constant URL_MAIL_ADDRESS
```

1.0.0.637 URL_MAIL_DOMAIN

```
include std/net/common.e  
public constant URL_MAIL_DOMAIN
```

1.0.0.638 URL_MAIL_QUERY

```
include std/net/common.e  
public constant URL_MAIL_QUERY
```

1.0.0.639 URL_MAIL_USER

```
include std/net/common.e  
public constant URL_MAIL_USER
```

1.0.0.640 URL_PASSWORD

```
include std/net/url.e  
public enum URL_PASSWORD
```

1.0.0.641 URL_PATH

```
include std/net/url.e
public enum URL_PATH
```

1.0.0.642 URL_PORT

```
include std/net/url.e
public enum URL_PORT
```

1.0.0.643 URL_PROTOCOL

```
include std/net/common.e
public constant URL_PROTOCOL
```

1.0.0.644 URL_PROTOCOL

```
include std/net/url.e
public enum URL_PROTOCOL
```

1.0.0.645 URL_QUERY_STRING

```
include std/net/url.e
public enum URL_QUERY_STRING
```

1.0.0.646 URL_USER

```
include std/net/url.e
public enum URL_USER
```

1.0.0.647 USED_BYTES

```
include std/filesys.e
public enum USED_BYTES
```

1.0.0.648 UTF8

```
public constant UTF8
```

Makes strings passed in to be interpreted as a UTF8 encoded string. This is passed to [new](#).

1.0.0.649 VALIDATE_ALL

```
include std/cmdline.e  
public enum VALIDATE_ALL
```

Validate all parameters (default). See [cmd_parse](#)

1.0.0.650 VC_COLOR

```
include std/graphcst.e  
public enum VC_COLOR
```

1.0.0.651 VC_COLUMNS

```
include std/graphcst.e  
public enum VC_COLUMNS
```

1.0.0.652 VC_LINES

```
include std/graphcst.e  
public enum VC_LINES
```

1.0.0.653 VC_MODE

```
include std/graphcst.e  
public enum VC_MODE
```



1.0.0.654 VC_NCOLORS

```
include std/graphcst.e
public enum VC_NCOLORS
```

1.0.0.655 VC_PAGES

```
include std/graphcst.e
public enum VC_PAGES
```

1.0.0.656 VC_SCRNCOLS

```
include std/graphcst.e
public enum VC_SCRNCOLS
```

1.0.0.657 Colors

1.0.0.658 VC_SCRNLINES

```
include std/graphcst.e
public enum VC_SCRNLINES
```

1.0.0.659 VC_XPIXELS

```
include std/graphcst.e
public enum VC_XPIXELS
```

1.0.0.660 VC_YPIXELS

```
include std/graphcst.e
public enum VC_YPIXELS
```

1.0.0.661 VirtualAlloc_rid

```
include std/memconst.e
export atom VirtualAlloc_rid
```

1.0.0.662 VirtualFree_rid

```
include std/memory.e
export atom VirtualFree_rid
```

1.0.0.663 VirtualFree_rid

```
include std/safe.e
export atom VirtualFree_rid
```

1.0.0.664 VirtualLock_rid

```
include std/memconst.e
export atom VirtualLock_rid
```

1.0.0.665 VirtualProtect_rid

```
include std/memconst.e
export atom VirtualProtect_rid
```

1.0.0.666 VirtualUnlock_rid

```
include std/memconst.e
export atom VirtualUnlock_rid
```

1.0.0.667 WHITE

```
include std/graphcst.e
public constant WHITE
```

1.0.0.668 WIN32

```
include std/os.e
public enum WIN32
```

1.0.0.669 WRAP

```
include std/pretty.e
public enum WRAP
```

1.0.0.670 W_BAD_PATH

```
include std/filesys.e
public constant W_BAD_PATH
```

Bad path error code. See [walk_dir](#)

1.0.0.671 YEAR

```
include std/datetime.e
public enum YEAR
```

Accessors

- YEAR
 - MONTH
 - DAY
 - HOUR
 - MINUTE
 - SECOND
-

1.0.0.672 YEARS

```
include std/datetime.e
public enum YEARS
```

Intervals

- YEARS
 - MONTHS
 - WEEKS
 - DAYS
 - HOURS
 - MINUTES
 - SECONDS
 - DATE
-
-

1.0.0.673 YELLOW

```
include std/graphcst.e  
public constant YELLOW
```

2 Routines

2.0.0.1 abort

```
<built-in> procedure abort (atom error)
```

Abort execution of the program.

Parameters:

1. `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 don't 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.

Example 1:

```
if x = 0 then
----
    abort(1)
else
    z = y / x
end if
```

See Also:

[crash_message](#), [system_exec](#)

2.0.0.2 abs

```
include std/math.e
public function abs(object a)
```

Returns the absolute value of numbers.

Parameters:

1. `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

Example 1:

```
x = abs({10.5, -12, 3})
-- x is {10.5, 12, 3}

i = abs(-4)
-- i is 4
```

See Also:

[sign](#)

2.0.0.3 absolute_path

```
include std/filesys.e
public function absolute_path(sequence filename)
```

Determine if the supplied string is an absolute path or a relative path.

Parameters:

1. `filename` : a sequence, the name of the file path

Returns:

An **integer**, 0 if filename is a relative path or 1 otherwise.

Comment:

A *relative* path is one which is relative to the current directory and an *absolute* path is one that doesn't need to know the current directory to find the file.

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")
```

2.0.0.4 accept

```
include std/socket.e
public function accept(socket sock)
```

Produces a new socket for an incoming connection.

Parameters:

1. 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().

2.0.0.5 add

```
include std/datetime.e
public function add(datetime dt, object qty, integer interval)
```

Add a number of *intervals* to a datetime.

Parameters:

1. `dt` : the base datetime
2. `qty` : the number of *intervals* to add. It should be positive.
3. `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/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, etc...

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.

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
```

See Also:

[subtract](#), [diff](#)

2.0.0.6 add_item

```
include std/sequence.e
public function add_item(object needle, sequence haystack, integer pOrder = 1)
```

Adds an item to the sequence if its not already there. If it already exists in the list, the list is returned unchanged.

Parameters:

1. `needle` : object to add.
2. `haystack` : sequence to add it to.
3. `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.
```

2.0.0.7 add_to

```
include std/sets.e
public function add_to(object x, set S)
```

Add an object to a set.

Parameters:

1. x : the object to add
2. S : the set to augment

Returns:

A **set**, which is a **copy** of S, with the addition of x if it was not there already.

Example 1:

```
set s0 = {1,3,5,7}
s0=add_to(2,s) -- s0 is now {1,2,3,5,7}
```

See Also:

[remove_from](#), [belongs_to](#), [union](#)

2.0.0.8 all_copyrights

```
include info.e
public function all_copyrights()
```

Get all copyrights associated with this version of Euphoria.

Returns:

A **sequence**, of product names and copyright messages.

```
{
  { ProductName, CopyrightMessage },
  { ProductName, CopyrightMessage },
  ...
}
```

2.0.0.9 all_left_units

```
include std/sets.e
public function all_left_units(operation f)
```

Finds all left units for an operation.

Parameters:

1. f : the operation to test.

Returns:

A possibly empty **sequence**, listing all x such that $f(x, .)$ is the identity map.

Example 1:

```
operation f = {{{1,2,3}, {1,2,3}, {3,1,2}}, {3,3,3}}
sequence s = all_left_units(f)
s is now {1,2}.
```

See Also:

[all_right_units](#), [is_unit](#), [has_unit](#)

2.0.0.10 all_matches

```
include std/regex.e
public function all_matches(regex re, string haystack, integer from = 1, option_spec options =
```

Get the text of all matches

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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`.

Example 1:

```
include std/regex.e as re
constant re_name = re:new("[A-Z][a-z]+ ([A-Z][a-z]+)")

object matches = re:match_all(re_name, "John Doe and Jane Doe")
-- matches is:
-- {
--   {
--     "John Doe", -- first match
--     "John",     -- full match data
--     "John",     -- first group
--     "Doe"       -- second group
--   },
--   {
--     "Jane Doe", -- second match
--     "Jane Doe", -- full match data
```

Parameters:

```

--      "Jane",      -- first group
--      "Doe"       -- second group
--    }
--  }

matches = re:match_all(re_name, "John Doe and Jane Doe", re:STRING_OFFSETS)
-- matches is:
-- {
--   {
--     { "John Doe", 1, 8 }, -- full match data
--     { "John",    1, 4 }, -- first group
--     { "Doe",     6, 8 } -- second group
--   },
--   {
--     { "Jane Doe", 14, 21 }, -- full match data
--     { "Jane",    14, 17 }, -- first group
--     { "Doe",     19, 21 } -- second group
--   }
-- }

```

See Also:[matches](#)**2.0.0.11 all_right_units**

```

include std/sets.e
public function all_right_units(operation f)

```

Finds all right units for an operation.

Parameters:

1. f : the operation to test.

Returns:

A possibly empty **sequence**, of all y such that $f(\cdot, y)$ is the identity map..

Example 1:

```

operation f = {{{1,2,3}, {1,2,3}, {3,1,2}}, {3,3,3}}
sequence s = all_right_units(f)
s is now empty.

```

See Also:

[all_left_units](#), [is_unit](#), [has_unit](#)

2.0.0.12 allocate

```
include std/machine.e
public function allocate(positive_int n, boolean cleanup = 0)
```

Allocate a contiguous block of data memory.

Parameters:

1. `n` : a positive integer, the size of the requested block.
2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to [delete](#).

Return:

An [atom](#), the address of the allocated memory or 0 if the memory can't be allocated.

Comments:

Since `allocate_string()` allocates memory, you are responsible to [free\(\)](#) the block when done with it if `cleanup` is zero. If `cleanup` is non-zero, then the memory can be freed by calling [delete](#), or when the pointer's reference count drops to zero. When you are finished using the block, you should pass the address of the block to `free()` if `cleanup` is zero. If `cleanup` is non-zero, then the memory can be freed by calling [delete](#), or when the pointer's reference count drops to zero. This will free the block and make the memory available for other purposes. When your program terminates, the operating system will reclaim all memory for use with other programs. An address returned by this function shouldn't be passed to `call()`. For that purpose you may use `allocate_code()` instead.

The address returned will be at least 4-byte aligned.

Example 1:

```
buffer = allocate(100)
for i = 0 to 99 do
    poke(buffer+i, 0)
end for
```

See Also:

[free](#), [peek](#), [poke](#), [mem_set](#), [allocate_code](#)

2.0.0.13 allocate_code

```
include std/machine.e
public function allocate_code(object data, valid_wordsize wordsize = 1)
```

Allocates and copies data into executable memory.

Parameters:

1. `a_sequence_of_machine_code` : is the machine code to be put into memory to be later called with `call()`
2. the word `length` : of the said code. You can specify your code as 1-byte, 2-byte or 4-byte chunks if you wish. If your machine code is byte code specify 1. The default is 1.

Return Value:

An **address**, The function returns the address in memory of the code, that can be safely executed whether DEP is enabled or not or 0 if it fails. On the other hand, if you try to execute a code address returned by `allocate()` with DEP enabled the program will receive a machine exception.

Comments:

Use this for the machine code you want to run in memory. The copying is done for you and when the routine returns the memory may not be readable or writeable but it is guaranteed to be executable. If you want to also write to this memory **after the machine code has been copied** you should use `allocate_protect()` instead and you should read about having memory executable and writeable at the same time is a bad idea. You mustn't use `free()` on memory returned from this function. You may instead use `free_code()` but since you will probably need the code throughout the life of your program's process this normally is not necessary. If you want to put only data in the memory to be read and written use `allocate`.

See Also:

[allocate](#), [free_code](#), [allocate_protect](#)

2.0.0.14 allocate_data

```
include std/machine.e
public function allocate_data(positive_int n, boolean cleanup = 0)
```

Parameters:

Allocate n bytes of memory and return the address. Free the memory using free() below.

2.0.0.15 allocate_pointer_array

```
include std/machine.e
public function allocate_pointer_array(sequence pointers, boolean cleanup = 0)
```

Allocate a NULL terminated pointer array.

Parameters:

1. `pointers` : a sequence of pointers to add to the pointer array.
2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to [delete](#)

Comments:

This function adds the NULL terminator.

Example 1:

```
atom pa = allocate_pointer_array({ allocate_string("1"), allocate_string("2") })
```

See Also:

[allocate_string_pointer_array](#), [free_pointer_array](#)

2.0.0.16 allocate_protect

```
include std/machine.e
public function allocate_protect(object data, valid_wordsize wordsize = 1, valid_memory_protect
```

Allocates and copies data into memory and gives it protection using [Standard Library Memory Protection Constants](#) or [Microsoft Windows Memory Protection Constants](#). The user may only pass in one of these constants. If you only wish to execute a sequence as machine code use `allocate_code()`. If you only want to read and write data into memory use `allocate()`.

See [MSDN: Microsoft's Memory Protection Constants](#)

Parameters:

1. `data` : is the machine code to be put into memory.
2. `wordsize` : is the size each element of data will take in memory. Are they 1-byte, 2-bytes or 4-bytes long? Specify here. The default is 1.
3. `protection` : is the particular Windows protection.

Returns:

An **address**, The function returns the address to the required memory or 0 if it fails. This function is guaranteed to return memory on the 4 byte boundary. It also guarantees that the memory returned with at least the protection given (but you may get more).

If you want to call `allocate_protect(data, PAGE_READWRITE)`, you can use `allocate` instead. It is more efficient and simpler.

If you want to call `allocate_protect(data, PAGE_EXECUTE)`, you can use `allocate_code()` instead. It is simpler.

You mustn't use `free()` on memory returned from this function, instead use `free_code()`.

2.0.0.17 `allocate_string`

```
include std/machine.e
public function allocate_string(sequence s, boolean cleanup = 0)
```

Allocate a C-style null-terminated string in memory

Parameters:

1. `s` : a sequence, the string to store in RAM.
2. `cleanup` : an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`.

Returns:

An **atom**, the address of the memory block where the string was stored, or 0 on failure.

Comments:

Only the 8 lowest bits of each atom in `s` is stored. Use `allocate_wstring()` for storing double byte encoded strings.

There is no `allocate_string_low()` function. However, you could easily craft one by adapting the code for `allocate_string`.

Since `allocate_string()` allocates memory, you are responsible to `free()` the block when done with it if `cleanup` is zero. If `cleanup` is non-zero, then the memory can be freed by calling `delete`, or when the pointer's reference count drops to zero.

Example 1:

```
atom title
title = allocate_string("The Wizard of Oz")
```

See Also:

[allocate](#), [allocate_wstring](#)

2.0.0.18 `allocate_string_pointer_array`

```
include std/machine.e
public function allocate_string_pointer_array(object string_list, boolean cleanup = 0)
```

Allocate a C-style null-terminated array of strings in memory

Parameters:

1. `string_list`: sequence of strings to store in RAM.
2. `cleanup`: an integer, if non-zero, then the returned pointer will be automatically freed when its reference count drops to zero, or when passed as a parameter to `delete`

Returns:

An `atom`, the address of the memory block where the string pointer array was stored.

Example 1:

```
atom p = allocate_string_pointer_array({ "One", "Two", "Three" })
-- Same as C: char *p = { "One", "Two", "Three", NULL };
```

See Also:

[free_pointer_array](#)

During the development of your application, you can define the word `SAFE` to cause `machine.e` to use alternative memory functions. These functions are slower but help in the debugging stages. In general, `SAFE` mode should not be enabled during production phases but only for development phases.

To define the word `SAFE` run your application with the `-D SAFE` command line option, or add to the top of your main file with `define SAFE`.

`Data Execute` mode makes data that will be returned from `allocate()` executable. On some systems `allocate()` will return memory that is not executable unless this mode has been enabled. When writing software you should use `allocate_code()` or `allocate_protect()` to get memory for execution. This is more efficient and more secure than using `Data Execute` mode. However, since on many systems executing memory returned from `allocate()` will work much software will be written 4.0 and yet use `allocate()` for executable memory instead of the afore mentioned routines. Therefore, you may use this switch when you find that your are getting `Data Execute Exceptions` running some software. `SAFE` mode will help you discover what memory should be changed to what access level. `Data Execute` mode will only work when the EUPHORIA program uses `std/machine.e` not `machine.e`.

2.0.0.19 `allocate_wstring`

```
include std/machine.e
public function allocate_wstring(sequence s, boolean cleanup = 0)
```

Create a C-style null-terminated `wchar_t` string in memory

Parameters:

1. `s` : a unicode (utf16) string

Returns:

An **atom**, the address of the allocated string, or 0 on failure.

See Also:

[allocate_string](#)

2.0.0.20 `allocations`

```
include std/safe.e
public function allocations()
```

Parameters:

2.0.0.21 allow_break

```
include std/console.e
public procedure allow_break(boolean b)
```

Set behavior of CTRL+C/CTRL+Break

Parameters:

1. `b` : a boolean, TRUE (`!= 0`) to enable the trapping of Ctrl-C/Ctrl-Break, FALSE (`0`) to disable it.

Comments:

When `b` is 1 (true), CTRL+C and CTRL+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 CTRL+C or CTRL+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\(\)](#).

Example 1:

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

See Also:

[check_break](#)

2.0.0.22 amalgamated_sum

```
include std/sets.e
public function amalgamated_sum(set first, set second, set base, map base_to_1, map base_to_2)
```

Returns all pairs in a product that come from applying two maps to the same element in a base set.

Parameters:

1. `first` : one of the sets to involved in the sum
2. `second` : the other set

Parameters:

3. `base` : the base set
4. `base_to_1` : the map from base to first
5. `base_to_2` : the map from base to second

Returns:

A **set**, of pairs obtained by applying `f01Xf02` to `s0`.

Example 1:

```
set s0,s1,s2
  s0={1,2,3} s1={5,7,9,11} s2={13,17,19}
  map f01,f02
  f01={2,4,1,3,4} f02={2,2,1,3,3}
  set s = amalgamated_product(s1,s2,s0,f01,f02)
  -- s is now {{7,17},{11,17},{5,13}}.
```

See Also:

[product](#), [product_map](#), [fiber_product](#)

2.0.0.23 `ampm`

```
include std/datetime.e
public sequence ampm
```

AM/PM

2.0.0.24 `and_bits`

```
<built-in> function and_bits(object a, object b)
```

Perform the logical AND operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are 1.

Parameters:

1. `a` : one of the objects involved
2. `b` : the second object

Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by logical 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.

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 logical operation is interpreted
-- as a signed 32-bit number, so it's negative.
```

See Also:

[or_bits](#), [xor_bits](#), [not_bits](#), [int_to_bits](#)

2.0.0.25 any_key

```
include std/console.e
public procedure any_key(sequence prompt = "Press Any Key to continue...", integer con = 1)
```

Parameters:

Display a prompt to the user and wait for any key.

Parameters:

1. `prompt` : Prompt to display, defaults to "Press Any Key to continue..."
2. `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.

Example 1:

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

Example 2:

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

See Also:

[wait_key](#)

2.0.0.26 append

```
<built-in> function append(sequence target, object x)
```

Adds an object as the last element of a sequence.

Parameters:

1. `source` : the sequence to add to
2. `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 `target` itself is `append()`ed to (as in Example 1 below) is highly optimized.

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"}}
```

See Also:

[prepend, &](#)

2.0.0.27 append_lines

```
include std/io.e
public function append_lines(sequence file, sequence lines)
```

Append a sequence of lines to a file.

Parameters:

1. `file` : an object, either a file path or the handle to an open file.
2. `lines` : the sequence of lines to write

Returns:

An **integer**, 1 on success, -1 on failure.

Errors:

If `puts()` cannot write some line of text, a runtime error will occur.

Comments:

file is opened, written to and then closed.

Example 1:

```
if append_lines("data.txt", {"This is important data", "Goodbye"}) != -1 then
    puts(STDERR, "Failed to append data\n")
end if
```

See Also:

[write_lines](#), [puts](#)

2.0.0.28 apply

```
include std/sequence.e
public function apply(sequence source, integer rid, object userdata = {})
```

Apply a function to every element of a sequence returning a new sequence of the same size.

Parameters:

- `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 parameters. The type of the first parameter must be compatible with all the elements in `source`. The second parameter is an object containing `userdata`.

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!"}
```

See Also:

[filter](#)

2.0.0.29 approx

```
include std/math.e
public function approx(object p, object q, atom epsilon = 0.005)
```

Compares two (sets of) numbers based on approximate equality.

Parameters:

1. `p` : an object, one of the sets to consider
2. `q` : an object, the other set.
3. `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 $p > (q + \text{epsilon})$: P is definitely greater than q.
- -1 when $p < (q - \text{epsilon})$: P is definitely less than q.
- 0 when $p \geq (q - \text{epsilon})$ and $p \leq (q + \text{epsilon})$: 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.
```

2.0.0.30 arccos

```
include std/math.e
public function arccos(trig_range x)
```

Return an angle given its cosine.

Parameters:

1. `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`.

Errors:

If any atom in `value` is not in the `-1..1` range, it cannot be the cosine of a real number, and an error occurs.

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()`.

Example 1:

```
s = arccos({-1,0,1})
-- s is {3.141592654, 1.570796327, 0}
```

See Also:

`cos`, `PI`, `arctan`

2.0.0.31 arccosh

```
include std/math.e
public function arccosh(not_below_1 a)
```

Computes the reverse hyperbolic cosine of an object.

Parameters:

1. `x` : the object to process.

Returns:

An **object**, the same shape as `x`, each atom of which was acted upon.

Errors:

Since `cosh` only takes values not below 1, an argument below 1 causes an error.

Comments:

The hyperbolic cosine grows like the logarithm function.

Example 1:

```
? arccosh(1) -- prints out 0
```

See Also:

[arccos](#), [arcsinh](#), [cosh](#)

2.0.0.32 arcsin

```
include std/math.e
public function arcsin(trig_range x)
```

Return an angle given its sine.

Parameters:

1. `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`.

Errors:

If any atom in `value` is not in the $-1..1$ range, it cannot be the sine of a real number, and an error occurs.

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\(\)](#).

Example 1:

```
s = arcsin({-1,0,1})
s is {-1.570796327, 0, 1.570796327}
```

See Also:

[arccos](#), [arccos](#), [sin](#)

2.0.0.33 arcsinh

```
include std/math.e
public function arcsinh(object a)
```

Computes the reverse hyperbolic sine of an object.

Parameters:

1. `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.

Example 1:

```
? arcsinh(1) -- prints out 0,4812118250596034
```

See Also:

[arccosh](#), [arcsin](#), [sinh](#)

2.0.0.34 arctan

```
<built-in> function arctan(object tangent)
```

Return an angle with given tangent.

Parameters:

1. `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()`.

Example 1:

```
s = arctan({1,2,3})
-- s is {0.785398, 1.10715, 1.24905}
```

See Also:

[arcsin](#), [arccos](#), [tan](#), [flatten](#)

2.0.0.35 arctanh

```
include std/math.e
public function arctanh(abs_below_1 a)
```

Computes the reverse hyperbolic tangent of an object.

Parameters:

1. `x` : the object to process.

Returns:

An **object**, the same shape as `x`, each atom of which was acted upon.

Errors:

Since `tanh` only takes values between -1 and $+1$ excluded, an out of range argument causes an error.

Comments:

The hyperbolic cosine grows like the logarithm function.

Example 1:

```
? arctanh(1/2) -- prints out 0,5493061443340548456976
```

See Also:

[arccos](#), [arcsinh](#), [cosh](#)

2.0.0.36 ascii_string

```
include std/types.e
public type ascii_string(object x)
```

Returns:

TRUE if argument is a sequence that only contains zero or more ASCII characters.

Comment:

An ASCII 'character' is defined as a integer in the range [0 to 127].

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
```

2.0.0.37 assert

```
include std/unittest.e
public procedure assert(object name, object outcome)
```

Records whether a test passes. If it fails, the program also fails.

Parameters:

1. `name` : a string, the name of the test
2. `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](#)

2.0.0.38 at

```
include std/stack.e
public function at(stack sk, integer idx = 1)
```

Fetch a value from the stack without removing it from the stack.

Parameters:

1. `sk` : the stack being queried
2. `idx` : an integer, the place to inspect. The default is 1 (top item).

Returns:

An **object**, the `idx`-th item of the stack.

Errors:

If the supplied value of `idx` does not correspond to an existing element, an error occurs.

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.

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"
```

See Also:

[size](#), [top](#), [peek_top](#), [peek_end](#)

2.0.0.39 atan2

```
include std/math.e
public function atan2(atom y, atom x)
```

Calculate the arctangent of a ratio.

Parameters:

1. *y* : an atom, the numerator of the ratio
2. *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.

Example 1:

```
a = atan2(10.5, 3.1)
-- a is 1.283713958
```

See Also:

[arctan](#)

2.0.0.40 atom

```
<built-in> function atom(object x)
```

Tests the supplied argument *x* to see if it is an atom or not.

Returns:

1. An integer.
 - ◆ 1 if *x* is an atom.
 - ◆ 0 if *x* is not an atom.

Example 1:

```
? atom(1) --> 1
? atom(1.1) --> 1
? atom("1") --> 0
```

See Also:

[sequence\(\)](#), [object\(\)](#), [integer\(\)](#)

2.0.0.41 atom_to_float32

```
include std/convert.e
public function atom_to_float32(atom a)
```

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

Parameters:

1. *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.

Example 1:

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

See Also:

[float32_to_atom](#), [int_to_bytes](#), [atom_to_float64](#)

2.0.0.42 atom_to_float64

```
include std/convert.e
public function atom_to_float64(atom a)
```

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

Parameters:

1. 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.

Example:

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

See Also:

[float64_to_atom](#), [int_to_bytes](#), [atom_to_float32](#)

2.0.0.43 attr_to_colors

```
include std/console.e
public function attr_to_colors(integer attr_code)
```

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

Parameters:

1. `attr_code` : integer, an attribute code.

Returns:

A sequence of two elements - {fgcolor, bgcolor}

Example 1:

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

See Also:

[get_screen_char](#), [colors_to_attr](#)

2.0.0.44 `avedev`

```
include std/stats.e
public function avedev(sequence data_set, object subseq_opt = ST_ALLNUM, integer population_type)
```

Returns the average of the absolute deviations of data points from their mean.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want the mean of the absolute deviations.
2. `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.
3. `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:

$$\text{avedev}(X) ==> \text{SUM}(\text{ABS}(X\{1..N\} - \text{MEAN}(X))) / N$$

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_FULLLPOP ) -- 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_FULLLPOP ) -- Ans: 2.0859375
```

See also:

[average](#), [stdev](#)

2.0.0.45 average

```
include std/stats.e
public function average(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the average (mean) of the data points.

Parameters:

1. `data_set` : A list of 1 or more numbers for which you want the mean.
2. `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.

The equation for average is:

$$\text{average}(X) \implies \text{SUM}(X\{1..N\}) / N$$

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:

```
? average( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,"text"}, ST_IGNSTR ) -- Ans: 5.13333333
```

See also:

[geomean](#), [harmean](#), [movavg](#), [emovavg](#)

2.0.0.46 begins

```
include std/search.e
public function begins(object sub_text, sequence full_text)
```

Test whether a sequence is the head of another one.

Parameters:

1. `sub_text` : an object to be looked for
2. `full_text` : a sequence, the head of which is being inspected.

Returns:

An **integer**, 1 if `sub_text` begins `full_text`, else 0.

Example 1:

```
s = begins("abc", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

See Also:

[ends](#), [head](#)

2.0.0.47 belongs_to

```
include std/sets.e
public function belongs_to(object x, set s)
```

Parameters:

Decide whether an object is in a set.

Parameters:

1. `x` : the object inquired about
2. `S` : the set being queried

Returns:

An **integer**, 1 if `x` is in `S`, else 0.

Example 1:

```
set s0 = {1,3,5,7}
?belongs_to(2,s)  -- prints out 0
```

See Also:

[is_subset](#) , [intersection](#), [difference](#)

2.0.0.48 `binary_search`

```
include std/search.e
public function binary_search(object needle, sequence haystack, integer start_point = 1, integer end_point = length(haystack) - 1)
```

Finds a "needle" in an ordered "haystack". Start and end point can be given for the search.

Parameters:

1. `needle` : an object to look for
2. `haystack` : a sequence to search in
3. `start_point` : an integer, the index at which to start searching. Defaults to 1.
4. `end_point` : an integer, the end point of the search. Defaults to 0, ie search to end.

Returns:

An **integer**, either:

1. a positive integer `i`, which means `haystack[i]` equals `needle`.
2. 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.

3. 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](#)

2.0.0.49 bind

```
include std/socket.e
public function bind(socket sock, sequence address, integer port = - 1)
```

Joins a socket to a specific local internet address and port so later calls only need to provide the socket.

Parameters:

1. `sock` : the socket
2. `address` : the address to bind the socket to
3. `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)
```

Parameters:

2.0.0.50 bits_to_int

```
include std/convert.e
public function bits_to_int(sequence bits)
```

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

Parameters:

1. `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.

Example 1:

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

See Also:

[bytes_to_int](#), [int_to_bits](#), [operations on sequences](#)

2.0.0.51 bk_color

```
include std/graphics.e
public procedure bk_color(color c)
```

Set the background color to one of the 16 standard colors.

Parameters:

1. `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, e.g. `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.

Example:

```
bk_color(BLACK)
```

See Also:

[text_color](#)

2.0.0.52 boolean

```
include std/types.e
public type boolean(object test_data)
```

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)
```

2.0.0.53 bordered_address

```
include std/memory.e
export type bordered_address(atom addr)
```

Euphoria objects are automatically garbage collected when they are no longer referenced anywhere. Euphoria also provides the ability to manage resources associated with euphoria objects. These resources could be open file handles, allocated memory, or other euphoria objects. There are two built-in routines for managing these external resources.

2.0.0.54 bordered_address

```
include std/safe.e
export type bordered_address(ext_addr addr)
```

2.0.0.55 breakup

```
include std/sequence.e
public function breakup(sequence source, object size, integer style = BK_LEN)
```

Breaks up a sequence into multiple sequences of a given length.

Parameters:

1. *source* : the sequence to be broken up into sub-sequences.
2. *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.
3. *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`...

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`.

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"}
```

See Also:

[split flatten](#)

2.0.0.56 build_commandline

```
include std/cmdline.e
public function build_commandline(sequence cmds)
```

Returns a text string based on the set of supplied strings. Typically, this is used to ensure that arguments on a command line are properly formed before submitting it to the shell.

Parameters:

1. `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 globbing `*`, `?`. And it is not specified here what happens if you pass redirection or piping characters.

Example 1:

```
s = build_commandline( { "-d", "/usr/my docs/" } )
-- s now contains '-d "/usr/my docs/'
```

Example 2:

You can use this to run things that might be difficult to quote out:

Suppose you want to run a program that requires quotes on its command line? Use this function to pass quotation marks:

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

See Also:

[parse_commandline](#), [system](#), [system_exec](#), [command_line](#)

2.0.0.57 build_list

```
include std/sequence.e
public function build_list(sequence source, object transformer, integer singleton = 1, object u
```

Implements "List Comprehension" or building a list based on the contents of another list.

Parameters:

Parameters:

1. `source` : A sequence. The list of items to base the new list upon.
2. `transformer` : One or more `routine_ids`. These are **routine ids** 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`.
3. `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).
4. `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)
  (x < 0) then
    {0} -- noretupnt
  else
    {1,x} -- repurtx'
  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}
```

2.0.0.58 builtins

```
include keywords.e
public constant builtins
```

Sequence of Euphoria's built-in function names

Syntax Color Break Euphoria statements into words with multiple colors. The editor and pretty printer (eprint.ex) both use this file.

2.0.0.59 byte_range

```
include std/io.e
public type byte_range(sequence r)
```

Byte Range Type

2.0.0.60 bytes_to_int

```
include std/convert.e
public function bytes_to_int(sequence s)
```

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

Parameters:

1. *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.

Example 1:

```
atom int32

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

See Also:

[bits_to_int](#), [float64_to_atom](#), [int_to_bytes](#), [peek](#), [peek4s](#), [peek4u](#), [poke4](#)

2.0.0.61 c_func

```
<built-in> function c_func(integer rid, sequence args={})
```

Call a C function, or machine code function, or translated/compiled Euphoria function by routine id.

Parameters:

1. `rid`: an integer, the `routine_id` of the external function being called.
2. `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()`.

Errors:

If `rid` is not a valid routine id, or the arguments do not match the prototype of the routine being called, an error occurs.

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).

Example 1:

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

```
-- open user32.dll - it contains the BeginPaint C function
```

Parameters:

```

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(user32, "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})

```

See Also:

[c_proc](#), [define_c_proc](#), [open_dll](#), [Platform-Specific Issues](#)

2.0.0.62 c_func

```

include std/safe.e
override function c_func(integer

```

2.0.0.63 c_proc

```
<built-in> procedure c_proc(integer rid, sequence args={})
```

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

Parameters:

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

Errors:

If `rid` is not a valid routine id, or the arguments do not match the prototype of the routine being called, an error occurs.

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. e.g. 5.9 will be passed as 5, -5.9 will be passed as -5.

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})
```

See Also:

[c_func](#), [define_c_func](#), [open_dll](#), [Platform-Specific Issues](#)

2.0.0.64 c_proc

```
include std/safe.e
override procedure c_proc(integer
```

2.0.0.65 calc_hash

```
include std/map.e
public function calc_hash(object key_p, integer max_hash_p)
```

Calculate a Hashing value from the supplied data.

Parameters:

1. pData : The data for which you want a hash value calculated.
2. 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 )
```

2.0.0.66 calc_primes

```
include std/primes.e
public function calc_primes(integer max_p, atom time_limit_p = 10)
```

Returns all the prime numbers below some threshold, with a cap on computation time.

Parameters:

1. `max_p` : an integer, the last prime returned is the next prime after or on this value.
2. `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 next prime number that falls on or after the value of `max_p`.

Comments:

- The returned sequence contains all the prime numbers less than its last element.
- If the function times out, it may not hold all primes below `max_p`, but only the largest ones will be absent. If the last element returned is less than `max_p` then the function timed out.
- To disable the timeout, simply give it a negative value.

Example 1:

```
? calc_primes(1000, 5)
-- 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 997.
-- This call will never take longer than 5 seconds.
```

Parameters:

See Also:

[next_prime](#) [prime_list](#)

2.0.0.67 call

```
<built-in> procedure call(atom addr)
```

Call a machine language routine which was stored in memory prior.

Parameters:

1. `addr` : an atom, the address at which to transfer execution control.

Comments:

The machine code routine must execute a RET instruction #C3 to return control to Euphoria. The routine should save and restore any registers that it uses.

You can allocate a block of memory for the routine and then poke in the bytes of machine code using `allocate_code()`. You might allocate other blocks of memory for data and parameters that the machine code can operate on using `allocate()`. The addresses of these blocks could be part of the machine code.

If your machine code uses the stack, use `c_proc()` instead of `call()`.

Example 1:

```
demo/callmach.ex
```

See Also:

[allocate_code](#), [free_code](#), [c_proc](#), [define_c_proc](#)

2.0.0.68 call

```
include std/safe.e
override procedure call(machine_addr)
```

2.0.0.69 call_back

```
include std/dll.e
public function call_back(object id)
```

Get a machine address for an Euphoria procedure.

Parameters:

1. `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-like shared library (.so), as a 32-bit "call-back" address for calling your Euphoria routine.

Errors:

The length of `name` should not exceed 1,024 characters.

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 Linux/FreeBSD `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.

Example 1:

See: demo\win32>window.exw, demo\linux\qsort.ex

See Also:

[routine_id](#)

2.0.0.70 call_func

```
<built-in> function call_func(integer id, sequence args={})
```

Call the user-defined Euphoria function by routine id.

Parameters:

1. `id` : an integer, the routine id of the function to call
2. `args` : a sequence, the parameters to pass to the function.

Returns:

The **value**, the called function returns.

Errors:

If `id` is negative or otherwise unknown, an error occurs.

If the length of `args` is not the number of parameters the function takes, an error occurs.

Comments:

`id` must be a valid routine id returned by [routine_id\(\)](#).

`args` must be a sequence of argument values of length `n`, where `n` is the number of arguments required by the called function. Defaulted parameters currently cannot be synthesized while making a indirect call.

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

Example 1:

Take a look at the sample program called `demo/csort.ex`

See Also:

[call_proc](#), [routine_id](#), [c_func](#)

2.0.0.71 call_proc

```
<built-in> procedure call_proc(integer id, sequence args={})
```

Call a user-defined Euphoria procedure by routine id.

Parameters:

1. `id`: an integer, the routine id of the procedure to call
2. `args`: a sequence, the parameters to pass to the function.

Errors:

If `id` is negative or otherwise unknown, an error occurs.

If the length of `args` is not the number of parameters the function takes, an error occurs.

Comments:

`id` must be a valid routine id returned by [routine_id\(\)](#).

`args` must be a sequence of argument values of length `n`, where `n` is the number of arguments required by the called procedure. Defaulted parameters currently cannot be synthesized while making an indirect call.

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

Example 1:

```
public integer foo_id

procedure x()
    call_proc(foo_id, {1, "Hello World\n"})
end procedure

procedure foo(integer a, sequence s)
    puts(a, s)
```

Parameters:

```
end procedure

foo_id = routine_id("foo")

x()
```

See Also:

[call_func](#), [routine_id](#), [c_proc](#)

2.0.0.72 can_add

```
include std/sequence.e
public function can_add(object a, object b)
```

Checks whether two objects can be legally added together.

Parameters:

1. a : one of the objects to test for compatible shape
2. b : the other object

Returns:

An **integer**, 1 if an addition (or any of the [Relational operators](#)) are possible between a and b, else 0.

Example 1:

```
i = can_add({1,2,3},{4,5})
-- i is 0

i = can_add({1,2,3},4)
-- i is 1

i = can_add({1,2,3},{4,{5,6},7})
-- i is 1
```

See Also:

[linear](#)

2.0.0.73 canon2win

```
include std/localeconv.e
public function canon2win(sequence new_locale)
```

TODO: document

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](#)
-

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 they 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.

2.0.0.74 Compile Time and Match Time

When a regular expression object is created via `new` we also say it gets "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 be 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.

2.0.0.75 canonical

```
include std/localeconv.e
public function canonical(sequence new_locale)
```

Get canonical name for a locale.

Parameters:

1. `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](#)

2.0.0.76 canonical_path

```
include std/filesys.e
public function canonical_path(sequence path_in, integer directory_given = 0, integer no_case =
```

Returns the full path and file name of the supplied file name.

Parameters:

1. `path_in`: A sequence. This is the file name whose full path you want.
2. `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.

3. `no_case` : An integer. Only applies to the Windows platform. If zero (the default) the path name is returned using the same case as supplied, otherwise the returned value is all in lowercase.

Returns:

A **sequence**, the full path and file name.

Comment:

- In non-Unix systems, the result is always in lowercase.
- The supplied file/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"
```

2.0.0.77 cardinal

```
include std/sets.e  
public function cardinal(set S)
```

Return the cardinal of a set

Parameters:

1. `S` : the set being queried.

Returns:

An **integer**, the count of elements in `S`.

See Also:

[set](#)

2.0.0.78 ceil

```
include std/math.e
public function ceil(object a)
```

Computes the next integer equal or greater than the argument.

Parameters:

1. `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, $X = \text{floor}(X) = \text{ceil}(X)$.

Example 1:

```
sequence nums
nums = {8, -5, 3.14, 4.89, -7.62, -4.3}
nums = ceil(nums) -- {8, -5, 4, 5, -7, -4}
```

See Also:

[floor](#), [round](#)

2.0.0.79 central_moment

```
include std/stats.e
public function central_moment(sequence data_set, object datum, integer order_mag = 1, object s
```

Returns the distance between a supplied value and the mean, to some supplied order of magnitude. This is used to get a measure of the *shape* of a data set.

Parameters:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `datum`: either a single value or a list of values for which you require the central moments.
3. `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
4. `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
```

See also:

[average](#)

2.0.0.80 chance

```
include std/rand.e
public function chance(atom my_limit, atom top_limit = 100)
```

Simulates the probability of a desired outcome.

Parameters:

1. `my_limit`: an atom. The desired chance of something happening.
2. `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 want something to happen with a probability 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)` #.

Example 1:

```
-- 65% of the days are sunny, so ...
if chance(65) then
    puts(1, "Today will be a sunny day")
elseif 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
```

See Also:

[rnd](#), [roll](#)

2.0.0.81 change_target

```
include std/sets.e
public function change_target(map f, set old_target, set new_target)
```

Converts a map by changing its output set.

Parameters:

Parameters:

1. `f` : the map to retarget
2. `old_target` : the initial target set for `f`
3. `new_target` : the new target set.

Returns:

A **map**, which agrees with `f` and has values in `new_target` instead of `old_target`, or "" if `f` hits something outside `new_target`.

Example 1:

```
set s1,s2
  s1={1,3,5,7,9,11} s2={1,3,7,11,17,19,23}
  map f = {2,1,4,6,2,6,6,6}
  map f0 = change_target(f,s1,s2)
  f0 is now: {2,1,3,4,2,4,6,7}
```

See Also:

[restrict](#), [direct_map](#)

2.0.0.82 char_test

```
include std/types.e
public function char_test(object test_data, sequence char_set)
```

Determine whether one or more characters are in a given character set.

Parameters:

1. `test_data` : an object to test, either a character or a string
2. `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'
```

2.0.0.83 chdir

```
include std/filesys.e
public function chdir(sequence newdir)
```

Set a new value for the current directory

Parameters:

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 *WIN32* 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 won't affect the current directory of its

parent process.

Example 1:

```
if chdir("c:\\euphoria") then
    f = open("readme.doc", "r")
else
    puts(STDERR, "Error: No euphoria directory?\n")
end if
```

See Also:

[current_dir](#), [dir](#)

2.0.0.84 check_all_blocks

```
include std/memory.e
public procedure check_all_blocks()
```

2.0.0.85 check_all_blocks

```
include std/safe.e
public procedure check_all_blocks()
```

2.0.0.86 check_break

```
include std/console.e
public function check_break()
```

Returns the number of Control-C/Control-BREAK key presses.

Returns:

An **integer**, the number of times that CTRL+C or CTRL+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 CTRL+C or CTRL+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.

Parameters:

Neither CTRL+C or CTRL+Break will be returned as input characters when you read the keyboard. You can only detect them by calling `check_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
```

See Also:

[allow_break](#)

2.0.0.87 check_calls

```
include std/memory.e
public integer check_calls
```

2.0.0.88 check_calls

```
include std/safe.e
public integer check_calls
```

Define block checking policy.

Comments:

If this integer is 1, (the default), check all blocks for edge corruption after each `call()`, `c_proc()` or `c_func()`. To save time, your program can turn off this checking by setting `check_calls` to 0.

2.0.0.89 check_free_list

```
include std/eds.e
public procedure check_free_list()
```

Detects corruption of the free list in a Euphoria database.

Parameters:

**Comments:**

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

2.0.0.90 checksum

```
include std/filesys.e
global function checksum(sequence filename, integer size = 4)
```

Returns a checksum value for the specified file.

Parameters:

1. `filename` : A sequence. The name of the file whose checksum you want.
2. `size` : An integer. The number of atoms to return. Default is 4

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 two files are different then use higher values for `size`. For example, `##size = 8` gives you 256 bits of file signature.

Example 1:

```
? checksum("myfile", 1) --> {92837498}
? checksum("myfile", 2) --> {1238176, 87192873}
? checksum("myfile", 4) --> {23448, 239807, 79283749, 427370}
? checksum("myfile")    --> {23448, 239807, 79283749, 427370} -- default
```

2.0.0.91 clear

```
include std/map.e
public procedure clear(map the_map_p)
```

Remove all entries in a map.

Parameters:

1. `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](#)

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
```

See Also:

[remove](#), [has](#)

2.0.0.92 clear

```
include std/stack.e
public procedure clear(stack sk)
```

Wipe out a stack.

Parameters:

1. `sk` : the stack to clear.

Side effect:

The stack contents is emptied.

See Also:

[new](#), [is_empty](#)

The sets.e module defines a type for sets and provides basic tools for handling them.

Other modules may be built upon them, for instance graph handling or simple topology, finite groups etc.

Notes:

- A *set* is an ordered sequence in ascending order, not more, not less
 - A *map* from setA to setB is a sequence the length of setA whose elements are indexes into setB, followed by {length(setA),length(setB)}.
 - An *operation* of $E \times F \implies G$ is a two dimensional sequence of elements of G, indexed by $E \times F$, and the triple {card(E),card(F),card(G)}.
-
-

2.0.0.93 clear_directory

```
include std/filesys.e
public function clear_directory(sequence path, integer recurse = 1)
```

Clear (delete) a directory of all files, but retaining sub-directories.

Parameters:

1. *name* : a sequence, the name of the directory whose files you want to remove.
2. *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.

Comment:

This never removes a directory. It only ever removes files. It is used to clear a directory structure of all existing files, leaving the structure intact.

Example 1:

```
integer cnt = clear_directory("the_old_folder")
if cnt = 0 then
    ("Filesystem problem - could not remove one or more of the files.")
end if
printf(1, "Number of files removed: %d\n", cnt - 1)
```

See Also:

[remove_directory](#), [delete_file](#)

2.0.0.94 clear_screen

```
<built-in> procedure clear_screen()
```

Clear the screen using the current background color (may be set by [bk_color\(\)](#)).

See Also:

[bk_color](#)

2.0.0.95 close

```
<built-in> procedure close(atom fn)
```

Close a file or device and flush out any still-buffered characters.

Parameters:

1. `fn` : an integer, the handle to the file or device to query.

Errors:

The target file or device must be open.

Comments:

Any still-open files will be closed automatically when your program terminates.

2.0.0.96 close

```
include std/pipeio.e
public function close(atom fd)
```

Close handle fd

Returns:

An **integer**, 0 on success, -1 on failure

Example 1:

```
integer status = pipeio:close(p[STDIN])
```

2.0.0.97 close

```
include std/socket.e
public function close(socket sock)
```

Closes a socket.

Parameters:

1. `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.

2.0.0.98 cmd_parse

```
include std/cmdline.e
public function cmd_parse(sequence opts, object parse_options = {}, sequence cmds = command_line())
```

Parse command line options, and optionally call procedures that relate to these options

Parameters:

1. `opts` : a sequence of valid option records: See Comments: section for details
2. `parse_options` : an optional sequence of parse options: See Parse Options section for details
3. `cmds` : an optional sequence of command line arguments. If omitted the output from `command_line()` is used.

Returns:

A **map**, containing the options set. The returned map has one special key named "extras" which are values passed on the command line that are not part of any option, for instance a list of files `myprog -verbose file1.txt file2.txt`. If any command element begins with an @ symbol then that file will be opened and its contents used to add to the command line.

Parse Options:

`parse_options` can be a sequence of options that will affect the parsing of the command line options. Options can be:

1. `VALIDATE_ALL` -- The default. All options will be validated for all possible errors.
2. `NO_VALIDATION` -- Do not validate any parameter.
3. `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.
4. `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; 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.
5. `NO_AT_EXPANSION` -- Do not expand arguments that begin with '@'.
6. `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

Parameters:

'@', 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:

1. a single '-'. Simply added to the 'extras' list
2. 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.
3. -shortName. The option will be looked up in the short name field of `opts`.
4. /shortName. Same as -shortName.
5. -!shortName. If the 'shortName' has already been found the option is removed.
6. /!shortName. Same as -!shortName
7. --longName. The option will be looked up in the long name field of `opts`.
8. --!longName. If the 'longName' has already been found the option is removed.
9. 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.

Option records have the following structure:

1. a sequence representing the (short name) text that will follow the "-" option format. Use an atom if not relevant
2. a sequence representing the (long name) text that will follow the "--" option format. Use an atom if not relevant
3. a sequence, text that describes the option's purpose. Usually short as it is displayed when "-h"/"--help" is on the command line. Use an atom if not required.
4. An object ...
 - ◆ If an **atom** then it can be either `HAS_PARAMETER` or anything else if there is no parameter for this option. This format also implies that the option is optional, case-sensitive and can only occur once.

- ◆ If a **sequence**, it can containing zero or more processing flags in any order ...
 - ◊ MANDATORY to indicate that the option must always be supplied.
 - ◊ HAS_PARAMETER to indicate that the option must have a parameter following it. You can optionally have a name for the parameter immediately follow the HAS_PARAMETER flag. If one isn't there, the help text will show "x" otherwise it shows the supplied name.
 - ◊ NO_CASE to indicate that the case of the supplied option is not significant.
 - ◊ ONCE to indicate that the option must only occur once on the command line.
 - ◊ MULTIPLE to indicate that the option can occur any number of times on the command line.
 - ◆ If both ONCE and MULTIPLE are omitted then switches that also have HAS_PARAMETER are only allowed once but switches without HAS_PARAMETER can have multiple occurrences but only one is recorded in the output map.
5. an integer; a **routine_id**. This function will be called when the option is located on the command line and before it updates the map.
- Use -1 if cmd_parse is not to invoke a function for this option.
- The user defined function must accept a single sequence parameter containing four values. If the function returns 1 then the command option does not update the map. You can use the predefined index values OPT_IDX, OPT_CNT, OPT_VAL, OPT_REV when referencing the function's parameter elements.
1. An index into the `opts` list.
 2. The number of times that the routine has been called by `cmd_parse` for this option
 3. The option's value as found on the command line
 4. 1 if the command line indicates that this option is to remove any earlier occurrences of it.

When assigning a value to the resulting map, the key is the long name if present, otherwise it uses the short name. For options, you must supply a short name, a long name or both.

If you want `cmd_parse()` to call a user routine for the extra command line values, you need to specify an Option Record that has neither a short name or a long name, in which case only the `routine_id` field is used.

For more details on how the command line is being pre-parsed, see [command_line](#).

Example:

```
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
end function
```

```

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

function opt_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])
    lreturn
end function

option_definition = {
    { "v", "verbose", "Verbose output",{NO_PARAMETER}, routine_id("opt_verbose")},
    { "h", "hash", "Calculate 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 },
    { 0, 0, 0, 0, routine_id("opt_extras")}
}

map:map opts = cmd_parse(option_definition)

-- 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"}

```

See Also:

[show_help, command_line](#)

2.0.0.99 color

```

include std/graphcst.e
public type color(integer x)

```

2.0.0.100 colors_to_attr

```

include std/console.e
public function colors_to_attr(object fgbg, integer bg = 0)

```

Parameters:

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

Parameters:

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

Returns:

An integer attribute code.

Example 1:

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

See Also:

[get_screen_char](#), [put_screen_char](#), [attr_to_colors](#)

2.0.0.101 columnize

```
include std/sequence.e
public function columnize(sequence source, object cols = {}, object defval = 0)
```

Converts a set of sub sequences into a set of 'columns'.

Parameters:

1. `source` : sequence containing the sub-sequences
2. `cols` : either a specific column number or a set of column numbers. Default is 0, which returns the maximum number of columns.
3. `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 1-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" }
```

2.0.0.102 combine

```
include std/sequence.e
public function combine(sequence source_data, integer proc_option = COMBINE_SORTED)
```

Combines all the sub-sequences into a single, optionally sorted, list

Parameters:

1. `source_data` : A sequence that contains sub-sequences to be combined.
2. `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) --> "aaacdeffghhiilllmnoorsstwww"
? combine(s, COMBINE_UNSORTED) --> "catdogfishwhalewolfsnailworm"
```

2.0.0.103 combine_maps

```
include std/sets.e
public function combine_maps(map f1, set source1, set target1, map f2, set source2, set target2)
```

Combines two maps into one defined from the union of source sets to the union of target sets.

Parameters:

1. f1 : the first map
2. source1 : its source set
3. target1 : its target set
4. f2 : the second map
5. source2 : its source set
6. target2 : its target set

Returns:

A **map**, from union(source1, source2) to union(target1, target2) which agrees with f1 and f2, or "" if f1 and f2 disagree at any point of intersection (s11, s21).

Errors:

If f1 and f2 are both defined for some point, they must have the same value at this point..

Example 1:

```
set s11,s12,s21,s22
  s11={2,3,5,7,11,13,17,19} s21={7,13,19,23,29}
  s12={-1,0,1,4} s22={-2,0,1,2,6}
map f1,f2
f1={2,1,3,3,2,3,1,2,8,4} f2={3,3,2,4,5,5,5}
map f = combine_maps(f1,s11,s12,f2,s21,s22)
-- f is now: {3,2,4,4,3,4,2,3,5,7,10,7}.
```

See Also:

[restrict](#), [direct_map](#)

2.0.0.104 command_line

```
<built-in> function command_line()
```

A **sequence**, of strings, where each string is a word from the command-line that started your program.

Returns:

1. The `path`, to either the Euphoria executable, (`eui`, `eui.exe`, `euid.exe` `euiw.exe`) or to your bound executable file.
2. The `next word`, is either the name of your Euphoria main file, or (again) the path to your bound executable file.
3. 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).

Example 1:

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

cmd = command_line()

-- cmd will be:
    {"C:\EUPHORIA\BIN\EUI.EXE",
     "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:
    {"C:\MYFILES\MYPROG.EXE",
     "C:\MYFILES\MYPROG.EXE", -- place holder
     "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.
```

See Also:

[build_commandline](#), [option_switches](#), [getenv](#), [cmd_parse](#), [show_help](#)

2.0.0.105 compare

<built-in> `function compare(object compared, object reference)`

Compare two items returning less than, equal or greater than.

Parameters:

1. compared : the compared object
2. 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. Sequences are compared alphabetically starting with the first element until a difference is found or one of the sequences is exhausted. Atoms are compared as ordinary reals.

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
```

See Also:

[equal](#), [relational operators](#), [operations on sequences](#), [sort](#)

2.0.0.106 compare

```
include std/map.e
public function compare(map map_1_p, map map_2_p, integer scope_p = 'd')
```

Compares two maps to test equality.

Parameters:

1. `map_1_p` : A map
2. `map_2_p` : A map
3. `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
    ... -- two maps have the same keys
```

2.0.0.107 compose_map

```
include std/sets.e
public function compose_map(map f1, map f2)
```

Creates a new map using elements from `f2`, mapped against `f1`

Parameters:

1. `f1` : the map containing indexes into `f2`
2. `f2` : the map containing elements used to build the resulting map.

Returns:

A **map**, f defined by $f(x) = f2(f1(x))$ for all x

Comments:

Each element in `f1` is an index into the elements of `f2`. So if `f1` contains `{3,2,1}` the result map contains the 3rd, 2nd and 1st element from `f2` in that order.

Errors:

Every element of `f1` must be a valid index into `f2`.

Example 1:

```
map f1, f2, f
  f1={2,3,1,1,2,5,3}
  f2={4,8,1,2,6,7,6,9}
  f=compose_map(f1, f2)
  -- f is now: {8,1,4,4,8,5,9}
```

See Also:

[diagram_commutes](#)

2.0.0.108 connect

```
include std/socket.e
public function connect(socket sock, sequence address, integer port = - 1)
```

Establish an outgoing connection to a remote computer. Only works with TCP sockets.

Parameters:

1. `sock` : the socket
2. `address` : ip address to connect, optionally with `:PORT` at the end
3. `port` : port number

Returns:

An **integer**, 0 for success and -1 on failure.

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
```

2.0.0.109 copy

```
include std/map.e
public function copy(map source_map, object dest_map = 0, integer put_operation = PUT)
```

Duplicates a map.

Parameters:

1. `source_map` : map to copy from
2. `dest_map` : optional, map to copy to
3. `put_operation` : optional, operation to use when `destmap##` 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.

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} }
```

See Also:

[put](#)

2.0.0.110 copy_file

```
include std/filesys.e
public function copy_file(sequence src, sequence dest, integer overwrite = 0)
```

Copy a file.

Parameters:

1. `src` : a sequence, the name of the file or directory to copy
2. `dest` : a sequence, the new name or location of the file
3. `overwrite` : an integer; 0 (the default) will prevent an existing destination file from being overwritten. Non-zero will overwrite the destination file.

Parameters:

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](#)

2.0.0.111 cos

```
<built-in> function cos(object angle)
```

Return the cosine of an angle expressed in radians

Parameters:

1. `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.

Example 1:

```
x = cos({.5, .6, .7})  
-- x is {0.8775826, 0.8253356, 0.7648422}
```

See Also:

[sin](#), [tan](#), [arccos](#), [PI](#), [deg2rad](#)

2.0.0.112 cosh

```
include std/math.e
public function cosh(object a)
```

Computes the hyperbolic cosine of an object.

Parameters:

1. 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, $\text{power}(\cosh(x), 2) - \text{power}(\sinh(x), 2) = 1$. Compare with ordinary trigonometry.

Example 1:

```
? cosh(LN2) -- prints out 1.25
```

See Also:

[cos](#), [sinh](#), [arccosh](#)

2.0.0.113 count

```
include std/stats.e
public function count(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the count of all the atoms in an object.

Parameters:

1. `data_set` : either an atom or a list.
2. `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.

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.

Returns:

An **integer**, the number of atoms in the set. When `data_set` is an atom, 1 is returned.

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
```

See also:

[average](#), [sum](#)

2.0.0.114 crash

```
include std/error.e
public procedure crash(sequence fmt, object data = {})
```

Crash running program, displaying a formatted error message the way `printf()` does.

Parameters:

1. `fmt` : a sequence representing the message text. It may have format specifiers in it
2. `data` : an object, defaulted to {}.

Comments:

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.

Example 1:

```
if PI = 3 then
    crash("The whole structure of universe just changed - please 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
```

See Also:

[crash_file](#), [crash_message](#), [printf](#)

2.0.0.115 crash_file

```
include std/error.e
public procedure crash_file(sequence file_path)
```

Specify a file path name in place of "ex.err" where you want any diagnostic information to be written.

Parameters:

1. `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](#)

2.0.0.116 crash_message

```
include std/error.e
public procedure crash_message(sequence msg)
```

Specify a final message to display for your user, in the event that Euphoria has to shut down your program due to an error.

Parameters:

1. `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.

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".
```

See Also:

[crash](#), [crash_file](#)

2.0.0.117 crash_routine

```
include std/error.e
public procedure crash_routine(integer func)
```

Specify a function to be called when an error takes place at run time.

Parameters:

1. `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.

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"))
```

See Also:

[crash_file](#), [routine_id](#), [Debugging and Profiling](#)

One use is to emulate PBR, such as Euphoria's `map` and `stack` types.

2.0.0.118 create

```
include std/pipeio.e
public function create()
```

Create pipes for inter-process communication

Returns:

A **handle**, process handles { {parent side pipes},{child side pipes} }

Example 1:

```
object p = exec("dir", create())
```

2.0.0.119 create

```
include std/socket.e  
public function create(integer family, integer sock_type, integer protocol)
```

Create a new socket

Parameters:

1. family: an integer
2. sock_type: an integer, the type of socket to create
3. 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**, -1 on failure, else a supposedly valid socket id.

Example 1:

```
socket = create(AF_INET, SOCK_STREAM, 0)
```

2.0.0.120 create_directory

```
include std/filesys.e
public function create_directory(sequence name, integer mode = 448, integer mkparent = 1)
```

Create a new directory.

Parameters:

1. name : a sequence, the name of the new directory to create
2. mode : on *Unix* systems, permissions for the new directory. Default is 448 (all rights for owner, none for others).
3. 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.

Example 1:

```
if not create_directory("the_new_folder") then
    ("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
    ("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
    ("Filesystem problem - could not create the new folder")
end if
```

See Also:

[remove_directory](#), [chdir](#)

2.0.0.121 create_file

```
include std/filesys.e
public function create_file(sequence name)
```

Create a new file.

Parameters:

1. 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.

Example 1:

```
if not create_file("the_new_file") then
    ("Filesystem problem - could not create the new file")
end if
```

See Also:

[create_directory](#)

2.0.0.122 curdir

```
include std/filesys.e
public function curdir(integer drive_id = 0)
```

Returns the current directory, with a trailing SLASH

**Parameters:**

1. `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.

Comment:

Windows maintain a current directory for each disk drive. You would use this routine if you wanted the current directory for a drive that may not be the current drive.

For Unix systems, this is simply ignored because there is only one current directory at any time on Unix.

Note:

This always ensures that the returned value has a trailing SLASH character.

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\  

```

2.0.0.123 current_dir

```
include std/filesys.e  
public function current_dir()
```

Return the name of the current working directory.

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:\

Parameters:

Example 1:

```
sequence s
s = current_dir()
-- s would have "C:\EUPHORIA\DOC" if you were in that directory
```

See Also:

[dir](#), [chdir](#)

2.0.0.124 cursor

```
include std/console.e
public procedure cursor(integer style)
```

Select a style of cursor.

Parameters:

1. `style`: an integer defining the cursor shape.

Platform:

Not *Unix*

Comments:

In pixel-graphics modes no cursor is displayed.

Example 1:

```
cursor(BLOCK_CURSOR)
```

Cursor Type Constants:

- `NO_CURSOR`
- `UNDERLINE_CURSOR`
- `THICK_UNDERLINE_CURSOR`
- `HALF_BLOCK_CURSOR`
- `BLOCK_CURSOR`

See Also:

[graphics_mode](#), [text_rows](#)

2.0.0.125 custom_sort

```
include std/sort.e
public function custom_sort(integer custom_compare, sequence x, object data = {}, integer order)
```

Sort the elements of a sequence according to a user-defined order.

Parameters:

1. `custom_compare` : an integer, the routine-id of the user defined routine that compares two items which appear in the sequence to sort.
2. `x` : the sequence of items to be sorted.
3. `data` : an object, either {} (no custom data, the default), an atom or a non-empty sequence.
4. `order` : an integer, either `NORMAL_ORDER` (the default) or `REVERSE_ORDER`.

Returns:

A **sequence**, a copy of the original sequence in sorted order

Errors:

If the user defined routine does not return according to the specifications in the *Comments* section below, an error will occur.

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 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", i.e. elements that are considered equal might change position relative to each other.

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}
}

```

See Also:

[compare](#), [sort](#)

2.0.0.126 date

```
<built-in> function date()
```

Return a sequence with information on the current date.

Returns:

A **sequence** of length 8, laid out as follows:

1. year -- since 1900
2. month -- January = 1
3. day -- day of month, starting at 1
4. hour -- 0 to 23
5. minute -- 0 to 59
6. second -- 0 to 59
7. day of the week -- Sunday = 1
8. 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.

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
```

See Also:

[time](#), [now](#)

2.0.0.127 datetime

```
include std/datetime.e
public type datetime(object o)
```

datetime type

Parameters:

Parameters:

1. `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.

Note:

All components must be integers except seconds, as those can also be floating point values.

2.0.0.128 datetime

```
include std/locale.e
public function datetime(sequence fmt, dt :datetime dtm)
```

Formats a date according to current locale.

Parameters:

1. `fmt` : A format string, as described in [datetime:format](#)
2. `dtm` : the datetime to write out.

Returns:

A **sequence**, representing the formatted date.

Example 1:

```
? datetime("Today is a %A",dt:now())
```

See Also:

[datetime:format](#)

Win32 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

2.0.0.129 day_abbrs

```
include std/datetime.e
public sequence day_abbrs
```

Abbreviations of day names

2.0.0.130 day_names

```
include std/datetime.e
public sequence day_names
```

Names of the days

Parameters:

2.0.0.131 days_in_month

```
include std/datetime.e
public function days_in_month(datetime dt)
```

Return the number of days in the month of dt.

This takes into account leap year.

Parameters:

1. dt : a datetime to be queried.

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
```

See Also:

[is_leap_year](#)

2.0.0.132 days_in_year

```
include std/datetime.e
public function days_in_year(datetime dt)
```

Return the number of days in the year of dt.

This takes into account leap year.

Parameters:

1. dt : a datetime to be queried.

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
```

See Also:

[is_leap_year](#), [days_in_month](#)

2.0.0.133 db_cache_clear

```
include std/eds.e
public procedure db_cache_clear()
```

Forces the database index cache to be cleared.

Parameters:

None

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.
```

2.0.0.134 db_clear_table

```
include std/eds.e
public procedure db_clear_table(sequence name, integer init_records = INIT_RECORDS)
```

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

Parameters:

1. name : a sequence, the name of the table to clear.

Errors:

An error occurs if the current database is not defined.

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](#)

2.0.0.135 db_close

```
include std/eds.e
public procedure db_close()
```

Unlock and close the current database.

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.

2.0.0.136 db_compress

```
include std/eds.e
public function db_compress()
```

Compresses the current database.

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

Parameters:

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
```

2.0.0.137 db_create

```
include std/eds.e
public function db_create(sequence path, integer lock_method = DB_LOCK_NO, integer init_tables
```

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

Parameters:

1. `path` : a sequence, the path to the file that will contain the database.
2. `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).
3. `init_tables` : an integer giving the initial number of tables to reserve space for. The default is 5 and the minimum is 1.
4. `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 path, `s`, does not end in `.edb`, it 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`.

Example 1:

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

See Also:

[db_open](#), [db_select](#)

2.0.0.138 db_create_table

```
include std/eds.e
public function db_create_table(sequence name, integer init_records = INIT_RECORDS)
```

Create a new table within the current database.

Parameters:

1. `name` : a sequence, the name of the new table.
2. `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.

Errors:

An error occurs if the current database is not defined.

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.

Example 1:

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

See Also:

[db_select_table](#), [db_table_list](#)

2.0.0.139 db_current

```
include std/eds.e
public function db_current()
```

Get name of currently selected database.

Parameters:

1. None.

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.

Example 1:

```
s = db_current_database()
```

See Also:

[db_select](#)

2.0.0.140 db_current_table

```
include std/eds.e
public function db_current_table()
```

Parameters:

Get name of currently selected table

Parameters:

1. None.

Returns:

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

Example 1:

```
s = db_current_table()
```

See Also:

[db_select_table](#), [db_table_list](#)

2.0.0.141 db_delete_record

```
include std/eds.e
public procedure db_delete_record(integer key_location, object table_name = current_table_name)
```

Delete record number `key_location` from the current table.

Parameter:

1. `key_location` : a positive integer, designating the record to delete.
2. `table_name` : optional table name to delete record from.

Errors:

If the current table is not defined, or `key_location` is not a valid record index, an error will occur. Valid record indexes are between 1 and the number of records in the table.

Example 1:

```
db_delete_record(55)
```

See Also:

[db_find_key](#)

2.0.0.142 db_delete_table

```
include std/eds.e
public procedure db_delete_table(sequence name)
```

Delete a table in the current database.

Parameters:

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

Errors:

An error occurs if the current database is not defined.

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](#)

2.0.0.143 db_dump

```
include std/eds.e
public procedure db_dump(object file_id, integer low_level_too = 0)
```

print the current database in readable form to file `fn`

Parameters:

1. `fn` : the destination file for printing the current Euphoria database;
2. `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.

Parameters:

Errors:

If the current database is not defined, an error will occur.

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.
```

2.0.0.144 db_fatal_id

```
include std/eds.e
public integer db_fatal_id db_fatal_id
```

Exception handler

Set this to a valid `routine_id` value for a procedure that will be called whenever the library detects a serious error. Your 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.

2.0.0.145 db_fetch_record

```
include std/eds.e
public function db_fetch_record(object key, object table_name = current_table_name)
```

Returns the data for the record with supplied key.

Parameters:

1. `key` : the identifier of the record to be looked up.
2. `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 occurred. A sequence, the data for the record.

Errors:

If the current table is not defined, it returns 0.

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](#).

Example 1:

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

See Also:

[db_find_key](#), [db_record_data](#)

2.0.0.146 db_find_key

```
include std/eds.e
public function db_find_key(object key, object table_name = current_table_name)
```

Find the record in the current table with supplied key.

Parameters:

1. `key` : the identifier of the record to be looked up.
2. `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 occurred.

Errors:

If the current table is not defined, it returns 0.

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 don't exist, you'll at least get a negative value showing where they would be, if they existed. e.g. 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.

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
```

See Also:

[db_insert](#), [db_replace_data](#), [db_delete_record](#), [db_get_recid](#)

2.0.0.147 db_get_errors

```
include std/eds.e
public function db_get_errors(integer clearing = 1)
```

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

Parameters:

1. `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.

1. Error Code.
2. Error Text.
3. Name of library routine that recorded the error.
4. 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't 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
```

2.0.0.148 db_get_recid

```
include std/eds.e
public function db_get_recid(object key, object table_name = current_table_name)
```

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

Parameters:

Parameters:

1. `key` : the identifier of the record to be looked up.
2. `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 wasn't found.
- If equal to zero, an error occurred.

Errors:

If the table is not defined, an error is raised.

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 doesn't 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.

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
```

See Also:

[db_insert](#), [db_replace_data](#), [db_delete_record](#), [db_find_key](#)

2.0.0.149 db_insert

```
include std/eds.e
public function db_insert(object key, object data, object table_name = current_table_name)
```

Insert a new record into the current table.

Parameters:

1. `key` : an object, the record key, which uniquely identifies it inside the current table
2. `data` : an object, associated to `key`.
3. `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.

Example 1:

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

See Also:

[db_delete_record](#)

2.0.0.150 db_open

```
include std/eds.e
public function db_open(sequence path, integer lock_method = DB_LOCK_NO)
```

Open an existing Euphoria database.

Parameters:

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

Returns:

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

The return codes are:

```
public constant
  DB_OK = 0           -- success
  DB_OPEN_FAIL = -1  -- could not open the file
  DB_LOCK_FAIL = -3  -- could not lock the file in the
                    -- manner requested
```

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 *WIN32* 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.

Example 1:

```
tries = 0
while 1 do
  err = db_open("mydata", DB_LOCK_SHARED)
  if err = DB_OK then
    exit
  elseif 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 if
```

```
end while
```

See Also:

[db_create](#), [db_select](#)

2.0.0.151 db_record_data

```
include std/eds.e
public function db_record_data(integer key_location, object table_name = current_table_name)
```

Returns the data in a record queried by position.

Parameters:

1. `key_location` : the index of the record the data of which is being fetched.
2. `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.

Errors:

If the current table is not defined, or if the record index is invalid, an error will occur.

Example 1:

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

See Also:

[db_find_key](#), [db_replace_data](#)

2.0.0.152 db_record_key

```
include std/eds.e
public function db_record_key(integer key_location, object table_name = current_table_name)
```

Parameters:

1. `key_location` : an integer, the index of the record the key is being requested.
2. `table_name` : optional table name to get record key from.

Returns An **object**, the key of the record being queried by index.

NOTE This function calls `fatal()` and returns a value of -1 if an error prevented the correct data being returned.

Errors:

If the current table is not defined, or if the record index is invalid, an error will occur.

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.

Example 1:

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

See Also:

[db_record_data](#)

2.0.0.153 db_record_recid

```
include std/eds.e
public function db_record_recid(integer recid)
```

Parameters:

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

Parameters:

1. `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.

Example 1:

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

See Also:

[db_get_recid](#), [db_replace_recid](#)

2.0.0.154 `db_rename_table`

```
include std/eds.e
public procedure db_rename_table(sequence name, sequence new_name)
```

Rename a table in the current database.

Parameters:

1. `name`: a sequence, the name of the table to rename
2. `new_name`: a sequence, the new name for the table

Errors:

- An error occurs if the current database is not defined.
- If name does not exist on the current database, or if new_name does exist on the current database, an error will occur.

Comments:

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

See Also:

[db_table_list](#)

2.0.0.155 db_replace_data

```
include std/eds.e
public procedure db_replace_data(integer key_location, object data, object table_name = current
```

In the current table, replace the data portion of a record with new data.

Parameters:

1. key_location: an integer, the index of the record the data is to be altered.
2. data: an object, the new value associated to the key of the record.
3. 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.

Example 1:

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

See Also:

[db_find_key](#)

2.0.0.156 db_replace_recid

```
include std/eds.e
public procedure db_replace_recid(integer recid, object data)
```

In the current database, replace the data portion of a record with new data. This can 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](#)

Parameters:

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

Comments:

- `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.

Example 1:

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

See Also:

[db_replace_data](#), [db_find_key](#), [db_get_recid](#)

2.0.0.157 db_select

```
include std/eds.e
public function db_select(sequence path, integer lock_method = - 1)
```

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

Parameters:

1. `path` : a sequence, the path to the database to be the new current database.
2. `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.

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
```

See Also:

[db_open](#), [db_select](#)

2.0.0.158 db_select_table

```
include std/eds.e
public function db_select_table(sequence name)
```

2.0.0.159 Managing tables**Parameters:**

1. name : a sequence which defines the name of the new current table.

On success, the table with name given by name becomes the current table.

Returns:

An **integer**, either DB_OK on success or DB_OPEN_FAIL otherwise.

Errors:

An error occurs if the current database is not defined.

Comments:

All record-level database operations apply automatically to the current table.

Example 1:

```
if db_select_table("salary") != DB_OK then
    puts(2, "Couldn't find salary table!\n")
    abort(1)
end if
```

See Also:

[db_table_list](#)

2.0.0.160 db_set_caching

```
include std/eds.e
public function db_set_caching(atom new_setting)
```

Sets the key cache behavior.

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.

Parameters:

1. `integer` : 0 will turn off caching, 1 will turn it back on.

Returns:

An **integer**, the previous setting of the option.

Comments:

When caching is turned off, the current cache contents is totally cleared.

Example 1:

```
x = db_set_caching(0) -- Turn off key caching.
```

Page Contents

2.0.0.161 db_table_list

```
include std/eds.e  
public function db_table_list()
```

Lists all tables on the current database.

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.

Errors:

An error occurs if the current database is undefined.

Example 1:

```
sequence names = db_table_list()  
for i = 1 to length(names) do  
    puts(1, names[i] & '\n')  
end for
```

Parameters:

See Also:

[db_select_table](#), [db_create_table](#)

2.0.0.162 Managing Records

2.0.0.163 db_table_size

```
include std/eds.e
public function db_table_size(object table_name = current_table_name)
```

Get the size (number of records) of the default table.

Parameters:

1. `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 occurred.

Errors:

If the current table is undefined, an error will occur.

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
```

See Also:

[db_replace_data](#)

2.0.0.164 deallocate

```
include std/memory.e
export procedure deallocate(atom addr)
```

Parameters:

2.0.0.165 deallocate

```
include std/safe.e
export procedure deallocate(atom a)
```

2.0.0.166 decanonical

```
include std/localeconv.e
public function decanonical(sequence new_locale)
```

Get the translation of a locale string for current platform.

Parameters:

1. `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](#)

2.0.0.167 decode

```
include std/net/url.e
public function decode(sequence what)
```

Convert all encoded entities to their decoded counter parts

Parameters:

1. `what`: what value to decode

Returns:

A decoded sequence

Example 1:

```
puts(1, decode("Fred+%26+Ethel"))
-- Prints "Fred & Ethel"
```

See Also:

[encode](#)

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](#)

2.0.0.168 defaulted_value

```
include std/get.e
public function defaulted_value(object st, object def, integer start_point = 1)
```

Perform a value() operation on a sequence, returning the value on success or the default on failure.

Parameters:

1. `st` : object to retrieve value from.
2. `def` : the value returned if `st` is an atom or `value(st)` fails.
3. `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#`.

Examples:

```
object 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}
```

See Also:

[value](#)

Page Contents

2.0.0.169 defaulttext

```
include std/filesys.e
public function defaulttext(sequence path, sequence defext)
```

Returns the supplied filepath with the supplied extension, if the filepath does not have an extension already.

Parameters:

1. `path` : the path to check for an extension.
2. `defext` : the extension to add if `path` does not have one.

Returns:

A **sequence**, the path with an extension.

Example:

```
-- ensure that the supplied path has an extension, but if it doesn't use "tmp".
theFile = defaulttext(UserFileName, "tmp")
```

See Also:

[pathinfo](#)

2.0.0.170 `define_c_func`

```
include std/dll.e
public function define_c_func(object lib, object routine_name, sequence arg_types, atom return_
```

Define the characteristics of either a C function, or a machine-code routine that returns a value.

Parameters:

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

Returns:

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

Errors:

The length of `name` should not exceed 1,024 characters.

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`.

Parameters:

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 won't have to use `atom_to_float64()` and `poke()` to get the floating-point values into memory.

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(user32, "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 WIN32 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
```

See Also:

demo\callmach.ex, [c_func](#), [define_c_proc](#), [c_proc](#), [open_dll](#)

2.0.0.171 define_c_proc

```
include std/dll.e
public function define_c_proc(object lib, object routine_name, sequence arg_types)
```

Define the characteristics of either a C function, or a machine-code routine that you wish to call as a procedure from your Euphoria program.

Parameters:

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

Returns:

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

Errors:

The length of `name` should not exceed 1,024 characters.

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()`.

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
```

See Also:

[c_proc](#), [define_c_func](#), [c_func](#), [open_dll](#)

2.0.0.172 define_c_var

```
include std/dll.e
public function define_c_var(atom lib, sequence variable_name)
```

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

Parameters:

1. `lib` : an atom, the address of a Linux or FreeBSD shared library, or Windows `.dll`, as returned by `open_dll()`.
2. `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.

Example:

see `euphoria/demo/linux/mylib.ex`

See Also:

[c_proc](#), [define_c_func](#), [c_func](#), [open_dll](#)

2.0.0.173 define_map

```
include std/sets.e
public function define_map(sequence mapping, set target)
```

Returns a map which sends each element of its source set to the corresponding one in a list.

Parameters:

1. `mapping` : the sequence mapped to
2. `target` : the target set that contains the elements `mapping` refers to by index

Returns:

The requested **map**, descriptor.

Example 1:

```
sequence s0 = {2, 3, 4, 1, 4, 2}
set s1 = {-1, 1, 2, 3, 4}
map f = define_map(s0,s1)
-- As a sequence, f is {3, 4, 5, 2, 5, 3, 6, 5}
```

See Also:

[map](#), [sequences_to_map](#), [direct_map](#)

2.0.0.174 define_operation

```
include std/sets.e
public function define_operation(sequence left_actions)
```

Returns an operation that splits by left action into the supplied mappings.

Parameters:

1. `left_actions` : a sequence of maps, the left actions of each element in the left hand set.

Returns:

An operation F , realizing the conditions above, with minimal cardinal values, or "" if the maps are not defined on the same set.

Errors:

`left_actions` must be a rectangular matrix.

Comments:

If F is the result, and is defined from $E1 \times E2$ to E , then each left action is a map from $E2$ to E , the "left multiplication" by an element of $E1$.

Example 1:

```
sequence s = {{2, 3, 2, 3}, {3, 1, 2, 5}, {1, 2, 2, 2}, {2, 3, 2, 4}, {3, 1, 2, 3}}
operation F = define_operation(s)
-- F is now {{2,3},{3,1},{1,2},{2,3},{3,1}},{5,2,3}
? operation(s) -- prints out 1.
```

See Also:

[operation](#)

2.0.0.175 deg2rad

```
include std/math.e
public function deg2rad(object x)
```

Convert an angle measured in degrees to an angle measured in radians

Parameters:

1. `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 $\text{PI}/180$.

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.

Example 1:

```
x = deg2rad(194)
-- x is 3.385938749
```

See Also:

[rad2deg](#)

2.0.0.176 delete

```
<built-in> procedure delete( object x )
```

Calls the cleanup routines associated with the object, and removes the association with those routines.

Parameters:

Comments:

The cleanup routines associated with the object are called in reverse order than they were added. If the object is an integer, or if no cleanup routines are associated with the object, then nothing happens.

After the cleanup routines are called, the value of the object is unchanged, though the cleanup routine will no longer be associated with the object.

2.0.0.177 delete_file

```
include std/filesys.e
public function delete_file(sequence name)
```

Delete a file.

Parameters:

1. name : a sequence, the name of the file to delete.

Returns:

An **integer**, 0 on failure, 1 on success.

2.0.0.178 delete_routine

```
<built-in> function delete_routine( object x, integer rid )
```

Associates a routine for cleaning up after a euphoria object.

Comments:

delete_routine() associates a euphoria object with a routine id meant to clean up any allocated resources. It always returns an atom (double) or a sequence, depending on what was passed (integers are promoted to atoms).

The routine specified by delete_routine() should be a procedure that takes a single parameter, being the object to be cleaned up after. Objects are cleaned up under one of two circumstances. The first is if it's called as a parameter to delete(). After the call, the association with the delete routine is removed.

The second way for the delete routine to be called is when its reference count is reduced to 0. Before its memory is freed, the delete routine is called. A default delete will be used if the cleanup parameter to one of the **allocate** routines is true.

Parameters:

delete_routine() may be called multiple times for the same object. In this case, the routines are called in reverse order compared to how they were associated.

2.0.0.179 delta

```
include std/sets.e
public function delta(set s1, set s2)
```

Returns the set of elements belonging to either of two sets.

Parameters:

1. s1 : One of the sets to take a symmetrical difference with
2. s2 : the other set.

Returns:

The **set**, of all elements belonging to either s1 or s2.

Example 1:

```
set s0,s1,s2
s1={1,3,5,7} s2={-1,2,3,7,11}
s0=delta(s1,s2)  -- s0 is now {-1,1,2,5,11}.
```

See Also:

[intersection](#), [union](#), [difference](#)

2.0.0.180 dep_works

```
include std/memory.e
export function dep_works()
```

Returns 1 if the DEP executing data only memory would cause an exception

2.0.0.181 dep_works

```
include std/safe.e
export function dep_works()
```

Parameters:

2.0.0.182 dequote

```
include std/text.e
public function dequote(sequence text_in, object quote_pairs = {"\"", "\""}, integer esc = -
```

Removes 'quotation' text from the argument.

Parameters:

1. `text_in`: The string or set of strings to de-quote.
2. `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.
3. `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"
```

2.0.0.183 deserialize

```
include std/serialize.e
public function deserialize(object sdata, integer pos = 1)
```

Convert a serialized object in to a standard Euphoria object.

Parameters:

1. `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.
2. `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})
```

Parameters:

```

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]

```

2.0.0.184 diagram_commutates

```

include std/sets.e
public function diagram_commutates(sequence f12a, sequence f12b, sequence f2a3, sequence f2b3)

```

Decide whether taking two different paths along a square map diagrams results in the same map.

Parameters:

1. `from_base_path_1` : the outgoing map along path 1
2. `from_base_path_2` : the outgoing map along path 2
3. `to_target_path_1` : the incoming map along path 1
4. `to_target_path_2` : the incoming map along path 2

Returns:

An **integer**, either 1 if `to_target_path_1 o from_base_path_1 = to_target_path_2 o from_base_path_2`.

Example 1:

```
map f12a, f12b, f2a3, f2b3
  f12a={2, 3, 1, 1, 2, 5, 3}
  f2a3={4, 8, 1, 2, 6, 7, 6, 9}
  f12b={2, 4, 2, 3, 1, 5, 4}
  f2b3={8, 8, 4, 1, 3, 5, 8}
  ?diagram_commutates (f12a, f12b, f2a3, f2b3)    -- prints out 0
```

See Also:

[compose_map](#)

2.0.0.185 diff

```
include std/datetime.e
public function diff(datetime dt1, datetime dt2)
```

Compute the difference, in seconds, between two dates.

Parameters:

1. `dt1` : the end datetime
2. `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.

Example 1:

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

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

See Also:

add, subtract

Cross platform file operations for Euphoria

2.0.0.186 difference

```
include std/sets.e
public function difference(set base, set removed)
```

Returns the set of elements belonging to some set and not to another.

Parameters:

1. base : the set from which a difference is to be taken
2. removed : the set of elements to remove from base.

Returns:

The **set**, of elements belonging to base but not to removed.

Example 1:

```
set s0,s1,s2
s1={1,3,5,7} s2={-1,2,3,7,11}
s0=difference(s1,s2)  -- s0 is now {1,5}.
```

See Also:

[remove_from](#), [is_subset](#), [delta](#)

2.0.0.187 dir

```
include std/filesys.e
public function dir(sequence name)
```

Return directory information for the specified file or directory.

Parameters:

1. 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

Errors:

The length of name should not exceed 1,024 characters.

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:

```
public constant
  -- File Attributes
  D_NAME      = 1,
  D_ATTRIBUTES = 2,
  D_SIZE      = 3,
  D_YEAR      = 4,
```

Parameters:

```

D_MONTH      = 5,
D_DAY        = 6,
D_HOUR       = 7,
D_MINUTE     = 8,
D_SECOND     = 9,
D_MILLISECOND = 10,
D_ALTNAME    = 11

```

The **attributes** element is a string sequence containing characters chosen from:

Attribute	Description
'd'	directory
'r'	read only file
'h'	hidden file
's'	system file
'v'	volume-id entry
'a'	archive file
'c'	compressed file
'e'	encrypted file
'N'	not indexed
'D'	a device name
'O'	offline
'R'	reparse point or symbolic link
'S'	sparse file
'T'	temporary file
'V'	virtual file

A normal file without special attributes would just have an empty string, "", in this field.

The top level directory, e.g. c:\ does not have "." or ".." entries.

This function is often used just to test if a file or directory exists.

Under *WIN32*, the argument can have a long file or directory name anywhere in the path.

Under *Unix*, the only attribute currently available is 'd' and the milliseconds are always zero.

WIN32: The file name returned in [D_NAME] will be a long file name. If [D_ALTNAME] is not zero, it contains the 'short' name of the file.

Example 1:

```

d = dir(current_dir())

-- d might have:
-- {
--   {".", "d", 0 1994, 1, 18, 9, 30, 02},

```

Parameters:

```
--      {"..", "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"

See Also:

[walk_dir](#)

2.0.0.188 dir_size

```
include std/filesys.e
public function dir_size(sequence dir_path, integer count_all = 0)
```

Returns the amount of space used by a directory.

Parameters:

1. `dir_path`: A sequence. This is the path that identifies the directory to inquire upon.
2. `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
```

2.0.0.189 direct_map

```
include std/sets.e
public function direct_map(map f, set s1, sequence s0, set s2)
```

Returns the image of a list by a map, given the input and output sets.

Parameters:

1. `f` : the map to apply
2. `input` : the source set
3. `elements` : the sequence to map
4. `output` : the target set.

Returns:

A **sequence**, of elements of `output` obtained by applying `f` to the corresponding element of `input`.

Errors:

This function errors out if `f` cannot map `input` to `output`.

Comments:

If `elements` has items which are not on `input`, they are ignored. Items may appear in any order any number of times.

Example:

```
sequence s0 = {2,3,4,1,4}
set t1,t2
t1={1,2,2.5,3,4} t2={11,13,17,19,23,29}
map f = {3,1,4,5,3,5,5}
sequence s2 = direct_map(f,t1,s0,t2)
-- s2 is now {11,29,17,17,17}.
```

See Also:

[reverse_map](#)

2.0.0.190 dirname

```
include std/filesys.e
public function dirname(sequence path, integer pcd = 0)
```

Return the directory name of a fully qualified filename

Parameters:

1. `path` : the path from which to extract information
2. `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.

Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

See Also:

[driveid](#), [filename](#), [pathinfo](#)

2.0.0.191 disk_metrics

```
include std/filesys.e
public function disk_metrics(object disk_path)
```

Returns some information about a disk drive.

Parameters:

1. `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]
```

2.0.0.192 disk_size

```
include std/filesys.e
public function disk_size(object disk_path)
```

Returns the amount of space for a disk drive.

Parameters:

1. `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])
```

2.0.0.193 display

```
include std/console.e
public procedure display(object data_in, object args = 1, integer finalnl = - 918273645)
```

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

Parameters:

Parameters:

1. `data_in` : Any object.
2. `args` : Optional arguments used to format the output. Default is 1.
3. `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.

Examples:

```
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'}

```

2.0.0.194 display_text_image

```

include std/console.e
public procedure display_text_image(text_point xy, sequence text)

```

Display a text image in any text mode.

Parameters:

1. `xy` : a pair of 1-based coordinates representing the point at which to start writing
2. `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()/display_text_image()` in a text-mode graphical user interface, to allow "pop-up" dialog boxes, and drop-down menus to appear and disappear without losing what was previously on the screen.

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.

```

See Also:

[save_text_image](#), [put_screen_char](#)

2.0.0.195 distributes_over

```
include std/sets.e
public function distributes_over(operation product, operation sum, integer transpose = 0)
```

Determine whether a product map distributes over a sum

Parameters:

1. `product`: the operation that may be distributive over `sum`
2. `sum`: the operations over which `product` might distribute
3. `transpose`: an integer, nonzero if `product` is a right operation. Defaults to 0.

Returns:

An **integer**, either of

- `SIDE_NONE` -- `product` does not distribute either way over `sum`
- `SIDE_LEFT` -- `product` distributes over `sum` on the left only
- `SIDE_RIGHT` -- `product` distributes over `sum` on the right only
- `SIDE_BOTH` -- `product` distributes over `sum` o(both ways)

Example 1:

```
operation sum = {{{{1,2,3},{2,3,1},{3,1,2}}},{3,3,3}}
operation product = {{{{1,1,1},{1,2,3},{1,3,2}}},{3,3,3}}
?distributes_right(product,sum,0)  -- prints out 1.
```

2.0.0.196 driveid

```
include std/filesys.e
public function driveid(sequence path)
```

Return the drive letter of the path on *WIN32* platforms.

Parameters:

Parameters:

1. `path` : the path from which to extract information

Returns:

A **sequence**, the file extension part of `path`.

TODO: Test

Example:

```
letter = driveid("C:\\EUPHORIA\\Readme.txt")
-- letter is "C"
```

See Also:

[pathinfo](#), [dirname](#), [filename](#)

2.0.0.197 dump

```
include std/serialize.e
public function dump(sequence data, sequence filename)
```

Saves a Euphoria object to disk in a binary format.

Parameters:

1. `data` : any Euphoria object.
2. `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 doesn't exist it is created, otherwise it is overwritten.

You can use the [load](#) function to recover the data from the file.

Example :

```
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
```

2.0.0.198 dup

```
include std/stack.e
public procedure dup(stack sk)
```

Repeat the top element of a stack.

Parameters:

1. sk : the stack.

Side effects:

The value of top() is pushed onto the stack, thus the stack size grows by one.

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.

Errors:

If the stack has no elements, an error occurs.

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)  -- ""
```

Parameters:

```
? 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
```

2.0.0.199 edges_only

```
include std/memory.e
public integer edges_only
```

2.0.0.200 edges_only

```
include std/safe.e
public integer edges_only
```

Determine whether to flag accesses to remote memory areas.

Comments:

If this integer is 1 (the default under *WIN32*), only check for references to the leader or trailer areas just outside each registered block, and don't complain about addresses that are far out of bounds (it's probably a legitimate block from another source)

For a stronger check, set this to 0 if your program will never read/write an unregistered block of memory.

On *WIN32* people often use unregistered blocks.

2.0.0.201 embed_union

```
include std/sets.e
public function embed_union(set s1, set s2)
```

Returns the embedding of a set into its union with another.

Parameters:

1. S1 : the set to embed
2. S2 : the other set

Returns:

A **set**, of indexes representing S1 inside union (S1, S2). Its length is `length(S1)`, and the values range from 1 to `length(S1) + length(S2)`.

Example 1:

```
set s1 = {2, 5, 7}, s2 = {1, 3, 4}
sequence s = embed_union(s1,s2) -- s is now {2, 5, 6}
```

See Also:

[embedding](#), [union](#)

2.0.0.202 embedding

```
include std/sets.e
public function embedding(set small, set large)
```

Returns the set of indexes of the elements of a set in a larger set, or 0 if not applicable

Parameters:

1. small : the set to embed
2. large : the supposedly larger set

Returns:

A **set**, of indexes if `small is_subset() large`, else 0. Each element is the index in `large` of the corresponding element of `small`. Its length is `length(small)` and the values range from 1 to `length(large)`.

Parameters:

Example 1:

```
set s0 = {1,3,5,7}
set s = embedding({3,5},s0)  -- s is now {2,3}
```

See Also:

[subsets](#), [belongs_to](#), [difference](#), [is_subset](#)

2.0.0.203 emovavg

```
include std/stats.e
public function emovavg(object data_set, atom smoothing_factor)
```

Returns the exponential moving average of a set of data points.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want a moving average.
2. `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}
```

See also:

[average](#)

2.0.0.204 encode

```
include std/net/url.e
public function encode(sequence what, sequence spacecode = "+")
```

Converts all non-alphanumeric characters in a string to their percent-sign hexadecimal representation, or plus sign for spaces.

Parameters:

1. what : the string to encode
2. spacecode : what to insert in place of a space

Returns:

A **sequence**, the encoded string.

Comments:

spacecode defaults to + as it is more correct, however, some sites want %20 as the space encoding.

Example 1:

```
puts(1, encode("Fred & Ethel"))
-- Prints "Fred+%26+Ethel"
```

See Also:

[decode](#)

2.0.0.205 ends

```
include std/search.e
public function ends(object sub_text, sequence full_text)
```

Test whether a sequence ends another one.

Parameters:

1. `sub_text` : an object to be looked for
2. `full_text` : a sequence, the tail of which is being inspected.

Returns:

An **integer**, 1 if `sub_text` ends `full_text`, else 0.

Example 1:

```
s = ends("def", "abcdef")
-- s is 1
s = begins("bcd", "abcdef")
-- s is 0
```

See Also:

[begins](#), [tail](#)

2.0.0.206 ensure_in_list

```
include std/math.e
public function ensure_in_list(object item, sequence list, integer default = 1)
```

Ensures that the `item` is in a list of values supplied by `list`

Parameters:

1. `item` : The object to test for.
2. `list` : A sequence of elements that `item` should be a member of.
3. `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)
```

2.0.0.207 ensure_in_range

```
include std/math.e
public function ensure_in_range(object item, sequence range_limits)
```

Ensures that the `item` is in a range of values supplied by inclusive `range_limits`

Parameters:

1. `item`: The object to test for.
2. `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)
```

2.0.0.208 equal

<built-in> `function equal(object left, object right)`

Compare two Euphoria objects to see if they are the same.

Parameters:

1. `left` : one of the objects to test
2. `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.

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
```

See Also:

[compare](#)

2.0.0.209 error_code

```
include std/socket.e
public function error_code()
```

Get the error code.

Returns:

Integer **OK** on no error, otherwise any one of the `ERR_` constants to follow.

2.0.0.210 error_message

```
include std/regex.e
public function error_message(object re)
```

If **new** returns an atom, this function will return a text error message as to the reason.

Parameters:

1. `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
```

2.0.0.211 error_names

```
include std/regex.e
public constant error_names
```

2.0.0.212 error_no

```
include std/pipeio.e
public function error_no()
```

Get error no from last call to a pipe function

Comments:

Value returned will be OS-specific, and is not always set on Windows at least

Example 1:

```
integer error = error_no()
```

2.0.0.213 error_to_string

```
include std/regex.e
public function error_to_string(integer i)
```

Converts an regex error to a string.

This can be useful for debugging and even something rough to give to the user incase of a regex failure. It's preferable to a number.

See Also:

[error_message](#)

2.0.0.214 escape

```
include std/regex.e
public function escape(string s)
```

Escape special regular expression characters that may be entered into a search string from user input.

Notes:

Special regex characters are:

. \ + * ? [^] \$ () { } = ! < > | : -

Parameters:

1. s: string sequence to escape

Returns:

An escaped sequence representing s.

Example 1:

```
include std/regex.e as re
sequence search_s = re:escape("Payroll is $***15.00")
-- search_s = "Payroll is \\$\\*\\*\\*15\\.00"
```

2.0.0.215 escape

```
include std/text.e
public function escape(sequence s, sequence what = "\\")
```

Escape special characters in a string

Parameters:

1. s: string to escape
2. what: sequence of characters to escape defaults to escaping a double quote.

Returns:

An escaped sequence representing s.

Example 1:

```
sequence s = escape("John \"Mc\" Doe")
puts(1, s)
-- output is: John \"Mc\" Doe
```

See Also:

[quote](#)

2.0.0.216 et_error_string

```
include tokenize.e
public function et_error_string(integer err)
```

return error string from error code

2.0.0.217 et_keep_blanks

```
include tokenize.e
public procedure et_keep_blanks(integer toggle)
```

return blank lines as tokens default is FALSE

2.0.0.218 et_keep_comments

```
include tokenize.e
public procedure et_keep_comments(integer toggle)
```

return comments as tokens default is FALSE

2.0.0.219 et_string_numbers

```
include tokenize.e
public procedure et_string_numbers(integer toggle)
```

return TDATA for all T_NUMBER tokens in "string" format

by default:

- T_NUMBER tokens return atoms
 - T_CHAR tokens return single integer chars
 - T_EOF tokens return undefined data
 - all other tokens return strings
-

Parameters:

2.0.0.220 et_tokenize_file

```
include tokenize.e
public function et_tokenize_file(sequence fname)
```

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.

2.0.0.221 `et_tokenize_string`

```
include tokenize.e
public function et_tokenize_string(sequence code)
```

2.0.0.222 `euphoria_copyright`

```
include info.e
public function euphoria_copyright()
```

Get the copyright statement for Euphoria

Returns:

A **sequence**, containing 2 sequences: product name and copyright message

Example 1:

```
sequence info = euphoria_copyright()
-- info = {
--     "Euphoria v4.0.0 alpha 3",
--     "Copyright (c) XYZ, ABC\n" &
--     "Copyright (c) ABC, DEF"
-- }
```

2.0.0.223 `exec`

```
include std/pipeio.e
public function exec(sequence cmd, sequence pipe)
```

Open process with command line `cmd`

Returns:

A **handle**, process handles { **PID**, **STDIN**, **STDOUT**, **STDERR** }

Example 1:

```
object p = exec("dir", create())
```

2.0.0.224 exp

```
include std/math.e  
public function exp(atom x)
```

Computes some power of E.

Parameters:

1. `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.

Example 1:

```
x = exp(5.4)  
-- x is 221.4064162
```

See Also:

[log](#)

2.0.0.225 extract

```
include std/sequence.e
public function extract(sequence source, sequence indexes)
```

Picks out from a sequence a set of elements according to the supplied set of indexes.

Parameters:

1. `source` : the sequence from which to extract elements
2. `indexes` : a sequence of atoms, the indexes of the elements to be fetched in `source`.

Returns:

A **sequence**, of the same length as `indexes`.

Example 1:

```
s = extract({11,13,15,17},{3,1,2,1,4})
-- s is {15,11,13,11,17}
```

See Also:

[slice](#)

2.0.0.226 fetch

```
include std/sequence.e
public function fetch(sequence source, sequence indexes)
```

Retrieves an element nested arbitrarily deep into a sequence.

Parameters:

1. `source` : the sequence from which to fetch
2. `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[$]]`

Errors:

If the path cannot be followed to its end, an error about reading a nonexistent element, or subscripting an atom, will occur.

Comments:

The last element of `indexes` may be a pair `{lower,upper}`, in which case a slice of the innermost referenced sequence is returned.

Example 1:

```
x = fetch({0,1,2,3, {"abc", "def", "ghi"}, 6}, {5,2,3})
-- x is 'f', or 102.
```

See Also:

[store](#), [Subscripting of Sequences](#)

2.0.0.227 fib

```
include std/math.e
public function fib(integer i)
```

Computes the Nth Fibonacci Number

Parameters:

1. `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
```

2.0.0.228 fiber_over

```
include std/sets.e
public function fiber_over(map f, set source, set target)
```

Given a map between two sets, returns {list of antecedents of elements in target, effective target}.

Parameters:

1. `f` : the inspected map
2. `source` : the source set
3. `target` : the target set.

Returns:

A **sequence**, which is empty on failure. On success, it has two elements:

- A sequence of sets; each of these sets is included in `source` and is mapped to a single point by `f`.
- A set, the points in `target` hit by `f`.

Comments:

The listed sets, which are reverse images of points in `target`, are called *fibers* of `f` over points, specially if they are isomorphic to one another for some extra algebraic or topological structure.

The fibers are enumerated in the same order as the points in the effective target, i.e. the points in `target` `f` hits.

Example 1:

```
set s1,s2
  s1={5,7,9,11} s2={13,17,19,23,29}
  map f = {2,1,4,1,4,5}
  sequence s = fiber_over(f,s1,s2)
  -- s is now {{7,11},{5},{9}},{13,17,23}}.
```

See Also:

[reverse_map](#), [fiber_product](#)

2.0.0.229 fiber_product

```
include std/sets.e
public function fiber_product(set first, set second, set base, map from_1_to_base, map from_2_to_base)
```

Returns the set of all pairs in a product on which two given componentwise maps agree.

Parameters:

1. `first` : the first product component
2. `second` : the second product component
3. `base` : the base set the fiber product is built on
4. `from_1_to_base` : the map from `first` to `base`.
5. `from_2_to_base` : the map from `second` to `base`.

Returns:

The `set`, of pairs whose coordinates are mapped consistently to `base` by `from_1_to_base` and `from_2_to_base` respectively.

Example 1:

```
set s0,s1,s2
  s0={1,2,3} s1={5,7,9,11} s2={13,17,19,23,29}
  map f10,f20
  f10={2,1,2,1,4,3} f20={1,3,3,2,3,5,3}
  set s = fiber_product(s1,s2,s0,f10,f20)
  -- s is now {{5,23},{7,13},{9,23},{11,13}}.
```

See Also:

[reverse_map](#), [amalgamated_sum](#), [fiber_over](#)

2.0.0.230 file_exists

```
include std/filesys.e
public function file_exists(object name)
```

Check to see if a file exists

Parameters:

1. 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
```

2.0.0.231 file_length

```
include std/filesys.e
public function file_length(sequence filename)
```

Return the size of a file.

Parameters:

1. 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](#)

2.0.0.232 file_number

```
include std/io.e
public type file_number(integer f)
```

File number type

2.0.0.233 file_position

```
include std/io.e
public type file_position(atom p)
```

File position type

2.0.0.234 file_timestamp

```
include std/filesys.e
public function file_timestamp(sequence fname)
```

Get the timestamp of the file

Parameters:

1. 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.

2.0.0.235 file_type

```
include std/filesys.e
public function file_type(sequence filename)
```

Get the type of a file.

Parameters:

1. `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`

2.0.0.236 filebase

```
include std/filesys.e
public function filebase(sequence path)
```

Return the base filename of path.

Parameters:

1. `path` : the path from which to extract information

Returns:

A **sequence**, the base file name part of path.

TODO: Test

Example 1:

```
base = filebase("/opt/euphoria/readme.txt")
-- base is "readme"
```

See Also:

[pathinfo](#), [filename](#), [fileext](#)

2.0.0.237 fileext

```
include std/filesys.e
public function fileext(sequence path)
```

Return the file extension of a fully qualified filename

Parameters:

1. `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.

Example 1:

```
fname = fileext("/opt/euphoria/docs/readme.txt")
-- fname is "txt"
```

See Also:

[pathinfo](#), [filename](#), [filebase](#)

2.0.0.238 filename

```
include std/filesys.e
public function filename(sequence path)
```

Parameters:

Return the file name portion of a fully qualified filename

Parameters:

1. `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.

Example 1:

```
fname = filename("/opt/euphoria/docs/readme.txt")
-- fname is "readme.txt"
```

See Also:

[pathinfo](#), [filebase](#), [fileext](#)

2.0.0.239 filter

```
include std/sequence.e
public function filter(sequence source, object rid, object userdata = {}, object rangetype = "")
```

Filter a sequence based on a user supplied comparator function.

Parameters:

- `source` : sequence to filter
- `rid` : Either a [routine_id](#) of function to use as comparator or one of the predefined comparitors.
- `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 comparator 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...

"<" or "lt"	return items in <code>source</code> that are less than <code>userdata</code>
"<=" or "le"	return items in <code>source</code> that are less than or equal to <code>userdata</code>
"=" or "==" or "eq"	return items in <code>source</code> that are equal to <code>userdata</code>
"!=" or "ne"	return items in <code>source</code> that are not equal to <code>userdata</code>
<td>return items in <code>source</code> that are greater than <code>userdata</code></td>	return items in <code>source</code> that are greater than <code>userdata</code>
<td>return items in <code>source</code> that are greater than or equal to <code>userdata</code></td>	return items in <code>source</code> that are greater than or equal to <code>userdata</code>
"in"	return items in <code>source</code> that are in <code>userdata</code>
"out"	return items in <code>source</code> that are not in <code>userdata</code>

- Range Type Usage

Range Type**Meaning**

"["	Inclusive range. Lower and upper are in the range.
"]"	Low Inclusive range. Lower is in the range but upper is not.
"["	High Inclusive range. Lower is not in the range but upper is.
"]"	Exclusive range. Lower and upper are not in the range.

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}
```

Parameters:

```

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, "in", {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)
  length(s) < 2 then
    s return
  end if
  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}

```

See Also:[apply](#)**2.0.0.240 find**

```
<built-in> function find(object needle, sequence haystack, integer start)
```

Find the first occurrence of a "needle" as an element of a "haystack", starting from position "start"..

Parameters:

1. needle : an object whose presence is being queried
2. haystack : a sequence, which is being looked up for needle
3. 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.

Comments:

`find()` and `find_from()` are identical, but you can omit giving `find()` a starting point.

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
```

See Also:

[find_from](#), [match](#), [match_from](#), [compare](#)

2.0.0.241 find

```
include std/regex.e
public function find(regex re, string haystack, integer from = 1, option_spec options = DEFAULT)
```

Return the first match of `re` in `haystack`. You can optionally start at the position `from`.

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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.
5. `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 matched pairs. For the explanation of the returned sequence, please see the first example.

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]+)
-- }
```

2.0.0.242 find_all

```
include std/regex.e
public function find_all(regex re, string haystack, integer from = 1, option_spec options = DEF
```

Return all matches of `re` in `haystack` optionally starting at the sequence position `from`.

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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}}
-- }
```

2.0.0.243 find_all

```
include std/search.e
public function find_all(object needle, sequence haystack, integer start = 1)
```

Find all occurrences of an object inside a sequence, starting at some specified point.

Parameters:

1. `needle` : an object, what to look for
2. `haystack` : a sequence to search in
3. `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.

Example 1:

```
s = find_all('A', "ABCABAB")
-- s is {1,4,6}
```

See Also:

[find](#), [match](#), [match_all](#)

2.0.0.244 find_any

```
include std/search.e
public function find_any(sequence needles, sequence haystack, integer start = 1)
```

Find any element from a list inside a sequence. Returns the location of the first hit.

Parameters:

1. `needles` : a sequence, the list of items to look for
2. `haystack` : a sequence, in which "needles" are looked for
3. `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.

Example 1:

```
location = find_any("aeiou", "John Smith", 3)
-- location is 8
```

Example 2:

```
location = find_any("aeiou", "John Doe")
-- location is 2
```

See Also:

[find](#), [find_from](#)

2.0.0.245 find_each

```
include std/search.e
public function find_each(sequence needles, sequence haystack, integer start = 1)
```

Find all instances of any element from the needle sequence that occur in the haystack sequence. Returns a list of indexes.

Parameters:

1. `needles` : a sequence, the list of items to look for
2. `haystack` : a sequence, in which "needles" are looked for
3. `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.

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}
```

See Also:

[find](#), [find_from](#), [find_any](#)

2.0.0.246 find_from

```
<built-in> function find_from(object needle, object haystack, integer start)
```

Find the first occurrence of a "needle" as an element of a "haystack". Search starts at a specified index.

Parameters:

1. `needle` : an object whose presence is being queried
2. `haystack` : a sequence, which is being looked up for `needle`
3. `start` : an integer, the index in `haystack` at which to start searching.

Returns:

An **integer**, 0 if `needle` is not on `haystack` past position `start`, else the smallest index, not less than `start`, of an element of `haystack` that equals `needle`.

Comments:

`start` may have any value from 1 to the length of `haystack` plus 1. (Analogous to the first index of a slice of `haystack`).

`find()` and `find_from()` are identical, but you can omit giving `find()` a starting point.

Example 1:

```
location = find_from(11, {11, 8, 11, 2, 3}, 2)
-- location is set to 3
```

Example 2:

```
names = {"mary", "rob", "george", "mary", ""}
location = find_from("mary", names, 3)
-- location is set to 4
```

See Also:

[find](#), [match](#), [match_from](#), [compare](#)

2.0.0.247 find_nested

```
include std/search.e
public function find_nested(object needle, sequence haystack, integer flags = 0, integer rtn_id)
```

Find any object (among a list) in a sequence of arbitrary shape at arbitrary nesting.

Parameters:

1. `needle` : an object, either what to look up, or a list of items to look up
2. `haystack` : a sequence, where to look up
3. `flags` : options to the function, see Comments section. Defaults to 0.
4. `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}.

The following flags are available to fine tune the search:

- `NESTED_BACKWARD` -- if on `flags`, search is performed backward. Default is forward.
- `NESTED_ALL` -- if on `flags`, all occurrences are looked for. Default is one hit only.

- `NESTED_ANY` -- if present on `flags`, `needle` is a list of items to look for. Not the default.
- `NESTED_INDEXES` -- if present on `flags`, an individual result is a pair {position, index in `needle`}. Default is just return the position.

If `s` is a single index list, or position, from the returned sequence, then `fetch(haystack, s) = needle`.

If a routine id is supplied, the routine must behave like `equal()` if the `NESTED_ANY` flag is not supplied, and like `find()` if it is. The routine is being passed the current `haystack` item and `needle`. The returned integer is interpreted as if returned by `equal()` or `find()`.

If the `NESTED_ANY` flag is specified, and `needle` is an atom, then the flag is removed.

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}}
```

See Also:

[find](#), [rfind](#), [find_any](#), [fetch](#)

2.0.0.248 find_replace

```
include std/regex.e
public function find_replace(regex ex, string text, sequence replacement, integer from = 1, opt
```

Replaces all matches of a regex with the replacement text.

Parameters:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `replacement` : a string, used to replace each of the full matches

4. `from` : optional start position
5. `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.

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"
```

2.0.0.249 find_replace

```
include std/search.e
public function find_replace(object needle, sequence haystack, object replacement, integer max
```

Finds a "needle" in a "haystack", and replace any, or only the first few, occurrences with a replacement.

Parameters:

1. `needle` : an object to search and perhaps replace
2. `haystack` : a sequence to be inspected
3. `replacement` : an object to substitute for any (first) instance of `needle`
4. `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.

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" }
```

See Also:

[find](#), [replace](#), [match_replace](#)

2.0.0.250 find_replace_callback

```
include std/regex.e
public function find_replace_callback(regex ex, string text, integer rid, integer limit = 0, in
```

When `limit` is positive, this routine replaces up to `limit` matches of `ex` in `text` with the result of the user defined callback, `rid`, and when `limit` is 0, replaces all matches of `ex` in `text` with the result of this user defined callback, `rid`.

The callback should take one sequence. The first member of this sequence will be a string representing the entire match and the subsequent members, if they exist, will be strings for the captured groups within the regular expression.

Parameters:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `rid` : routine id to execute for each match
4. `limit` : the number of matches to process
5. `from` : optional start position
6. `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`.

Example 1:

```
include std/regex.e as re
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 "
```

Page Contents

2.0.0.251 find_replace_limit

```
include std/regex.e
public function find_replace_limit(regex ex, string text, sequence replacement, integer limit,
```

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.

This function is identical to `find_replace` except it allows you to limit the number of replacements to perform. Please see the documentation for `find_replace` for all the details.

Parameters:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `replacement` : a string, used to replace each of the full matches
4. `limit` : the number of matches to process
5. `from` : optional start position
6. `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`.

See Also:

[find_replace](#)

2.0.0.252 flags_to_string

```
include std/flags.e
public function flags_to_string(object flag_bits, sequence flag_names, integer expand_flags = 0
```

Returns a list of strings that represent the human-readable identities of the supplied flag(s).

Parameters:

1. `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.
2. `flag_names` : A sequence of two-element sub-sequences. Each sub-sequence contains `{FlagValue, FlagName}`, where `FlagName` is a string and `FlagValue` is the set of bits that set the flag on.
3. `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 "?".

Examples:

```
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"
--> },
--> {
-->   "?"
--> },
--> {
```

Parameters:

```

-->     "?"
--> },
--> {
-->     "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"
--> }

```

A map is a special array, often called an associative array or dictionary, in which the index to the data can be any Euphoria object and not just an integer. These sort of indexes are also called keys. For example we can code things like this...

```

custrec = new() -- Create a new map
  put(custrec, "Name", "Joe Blow")
  put(custrec, "Address", "555 High Street")
  put(custrec, "Phone", 555675632)

```

This creates three elements in the map, and they are indexed by "Name", "Address" and "Phone", meaning that to get the data associated with those keys we can code ...

```
object data = get(custrec, "Phone")
-- data now set to 555675632
```

Note: Only one instance of a given key can exist in a given map, meaning for example, we couldn't have two separate "Name" values in the above *custrec* map.

Maps automatically grow to accommodate all the elements placed into it.

Associative arrays can be implemented in many different ways, depending on what efficiency trade-offs have been made. This implementation allows you to decide if you want a *small* map or a *large* map.

small map

Faster for small numbers of elements. Speed is usually proportional to the number of elements.

large map

Faster for large number of elements. Speed is usually the same regardless of how many elements are in the map. The speed is often slower than a small map.

Note: If the number of elements placed into a *small* map take it over the initial size of the map, it is automatically converted to a *large* map.

2.0.0.253 flatten

```
include std/sequence.e
public function flatten(sequence s, object delim = "")
```

Remove all nesting from a sequence.

Parameters:

1. *s* : the sequence to flatten out.
2. *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"
```

2.0.0.254 float32_to_atom

```
include std/convert.e
public function float32_to_atom(sequence_4 ieee32)
```

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

Parameters:

1. `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.

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)
```

See Also:

[float64_to_atom](#), [bytes_to_int](#), [atom_to_float32](#)

2.0.0.255 float64_to_atom

```
include std/convert.e
public function float64_to_atom(sequence_8 ieee64)
```

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

Parameters:

1. `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.

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)
```

See Also:

[float32_to_atom](#), [bytes_to_int](#), [atom_to_float64](#)

2.0.0.256 floor

```
<built-in> function floor(object value)
```

Rounds `value` down to the next integer less than or equal to `value`. It does not simply truncate the fractional part, but actually rounds towards negative infinity.

Parameters:

Parameters:

1. `value` : any Euphoria object; each atom in `value` will be acted upon.

Returns:

An **object**, the same shape as `value` but with each item guaranteed to be an integer less than or equal to the corresponding item in `value`.

Example 1:

```
y = floor({0.5, -1.6, 9.99, 100})
-- y is {0, -2, 9, 100}
```

See Also:

[ceil](#), [round](#)

2.0.0.257 flush

```
include std/io.e
public procedure flush(file_number fn)
```

Force writing any buffered data to an open file or device.

Parameters:

1. `fn` : an integer, the handle to the file or device to close.

Errors:

The target file or device must be open.

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.

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
```

See Also:

[close](#), [crash_routine](#)

2.0.0.258 for_each

```
include std/map.e
public function for_each(map source_map, integer user_rid, object user_data = 0, integer in_sor
```

Calls a user-defined routine for each of the items in a map.

Parameters:

1. `source_map`: The map containing the data to process
2. `user_rid`: The `routine_id` of a user defined processing function
3. `user_data`: An object. Optional. This is passed, unchanged to each call of the user defined routine. By default, zero (0) is used.
4. `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.
5. `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.

Comment:

- The user defined routine is a function that must accept four parameters.
 1. Object: an Item Key
 2. Object: an Item Value
 3. Object: The `user_data` value. This is never used by `for_each()` itself, merely passed to the user routine.
 4. 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
-----
```

Page Contents

2.0.0.259 format

```
include std/datetime.e
public function format(datetime d, sequence pattern = "%Y-%m-%d %H:%M:%S")
```

Format the date according to the format pattern string

Parameters:

1. `d` : a datetime which is to be printed out
2. `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

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"
```

See Also:

[to_unix, parse](#)

2.0.0.260 format

```
include std/text.e
public function format(sequence format_pattern, object arg_list = {})
```

Formats a set of arguments in to a string based on a supplied pattern.

Parameters:

1. `format_pattern`: A sequence: the pattern string that contains zero or more tokens.
2. `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.

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
,	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.
.X	N.B. if hex or binary output was specified, the separators are every 4 digits otherwise they are every three digits.

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.

Examples:

```
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] [51] [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"

```

See Also:[sprintf](#)**2.0.0.261 frac**

```

include std/math.e
public function frac(object x)

```

Return the fractional portion of a number.

Parameters:

1. 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`

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}
```

See Also:

[trunc](#)

2.0.0.262 free

```
include std/eumem.e
export procedure free(atom mem_p)
```

Deallocate a block of (pseudo) memory

Parameters:

1. `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)
```

2.0.0.263 free

```
include std/machine.e
public procedure free(object addr)
```

Free up a previously allocated block of memory.

2.0.0.264 free_code

```
include std/machine.e
public procedure free_code( atom addr, integer size, valid_wordsize wordsize = 1 )
```

Frees up allocated code memory

Parameters:

1. `addr` : must be an address returned by `allocate_code()` or `allocate_protect()`. Do **not** pass memory returned from `allocate()` here!
2. `size` : is the length of the sequence passed to `allocate_code()` or the size you specified when you called `allocate_protect()`.
3. `wordsz`: `valid_wordsize` default = 1

Comments:

Chances are you will not need to call this function because code allocations are typically public scope operations that you want to have available until your process exits.

See Also: [allocate_code](#), [free](#)

2.0.0.265 free_code

```
include std/memory.e
public procedure free_code(atom addr, integer size, valid_wordsize wordsize = 1)
```

This is a slower DEBUGGING VERSION of machine.e

How To Use This File:

1. If your program doesn't already include machine.e add: include std/machine.e to your main .ex[w][u] file at the top.

2. To turn debug version on, issue

```
with define SAFE
```

in your main program, before the statement including machine.e.

3. If necessary, call register_block(address, length, memory_protection) to add additional "external" blocks of memory to the safe_address_list. These are blocks of memory that are safe to use but which you did not acquire through Euphoria's allocate(), allocate_data(), allocate_code() or memory_protect(). Call unregister_block(address) when you want to prevent further access to an external block.

4. Run your program. It might be 10x slower than normal but it's worth it to catch a nasty bug.

5. If a bug is caught, you will hear some "beep" sounds. Press Enter to clear the screen and see the error message. There will be a "divide by zero" traceback in ex.err so you can find the statement that is making the illegal memory access.

6. To switch between normal and debug versions, simply comment in or out the "with define SAFE" directive. In means debugging and out means normal. Alternatively, you can use -D SAFE as a switch on the command line (debug) or not (normal).

7. The older method of switching files and renaming them *no longer works*. machine.e conditionally includes safe.e.

This file is equivalent to machine.e, but it overrides the built-in

routines:

poke, peek, poke4, peek4s, peek4u, call, mem_copy, and mem_set

and it provides alternate versions of:

allocate, free

Your program will only be allowed to read/write areas of memory that it allocated (and hasn't freed), as well as areas in low memory that you list below, or add dynamically via `register_block()`.

2.0.0.266 free_code

```
include std/safe.e
public procedure free_code(atom addr, integer size, valid_wordsize wordsize = 1)
```

2.0.0.267 free_console

```
include std/console.e
public procedure free_console()
```

Free (delete) any console window associated with your program.

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 WIN32 this window will automatically disappear when your program terminates, but you can call `free_console()` to make it disappear sooner. On Linux or FreeBSD, 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-style systems, `free_console()` will set the terminal parameters back to normal, undoing the effect that `curses` has on the screen.

In an xterm window, 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's a WIN32 API routine, `FreeConsole()` that does something similar to `free_console()`. You should 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](#)

2.0.0.268 free_pointer_array

```
include std/machine.e
public procedure free_pointer_array(atom pointers_array)
```

Free a NULL terminated pointers array.

Parameters:

1. `pointers_array` : memory address of where the NULL terminated array exists at.

Comments:

This is for NULL terminated lists, such as allocated by [allocate_pointer_array](#). Do not call `free_pointer_array()` for a pointer that was allocated to be cleaned up automatically. Instead, use [delete](#).

See Also:

[allocate_pointer_array](#), [allocate_string_pointer_array](#)

2.0.0.269 from_date

```
include std/datetime.e
public function from_date(sequence src)
```

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

Parameters:

1. `src` : a sequence which `date()` might have returned

Returns:

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

Example 1:

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

See Also:

[date](#), [from_unix](#), [now](#), [new](#)

2.0.0.270 from_unix

```
include std/datetime.e
public function from_unix(atom unix)
```

Create a datetime value from the unix numeric format (seconds since EPOCH)

Parameters:

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

Returns:

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

Example 1:

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

See Also:

[to_unix](#), [from_date](#), [now](#), [new](#)

2.0.0.271 gcd

```
include std/math.e
public function gcd(atom p, atom q)
```

Returns the greater common divisor of two atoms

Parameters:

1. `p` : one of the atoms to consider
2. `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

2.0.0.272 geomean

```
include std/stats.e
public function geomean(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the geometric mean of the atoms in a sequence.

Parameters:

1. `data_set` : the values to take the geometric mean of.
2. `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.

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
```

See Also:

[average](#)

2.0.0.273 get

```
include std/locale.e
public function get()
```

Get current locale string

Returns:

A **sequence**, a locale string.

See Also:

[set](#)

2.0.0.274 get

```
include std/map.e
public function get(map the_map_p, object the_key_p, object default_value_p = 0)
```

Parameters:

Retrieves the value associated to a key in a map.

Parameters:

1. `the_map_p` : the map to inspect
2. `the_key_p` : an object, the `the_key_p` being looked tp
3. `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.

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
```

See Also:

[has](#)

2.0.0.275 get

```
include std/get.e
public function get(integer file, integer offset = 0, integer answer = GET_SHORT_ANSWER)
```

Input, from an open file, a human-readable string of characters representing a Euphoria object. Convert the string into the numeric value of that object.

Parameters:

1. `file` : an integer, the handle to an open file from which to read
2. `offset` : an integer, an offset to apply to file position before reading. Defaults to 0.

Parameters:

3. `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, e.g. `{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'd 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.

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:

See `bin\mydata.ex`

See Also:

[value](#)

2.0.0.276 `get_active_id`

```
include std/win32/msgbox.e  
public constant get_active_id
```

2.0.0.277 get_bytes

```
include std/io.e
public function get_bytes(integer fn, integer n)
```

Read the next bytes from a file.

Parameters:

1. `fn` : an integer, the handle to an open file to read from.
2. `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've 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.

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"
```

See Also:

[getc](#), [gets](#), [get_integer32](#), [get_dstring](#)

2.0.0.278 get_charsets

```
include std/types.e
public function get_charsets()
```

Gets the definition for each of the defined character sets.

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.

Example 1:

```
sequence sets
sets = get_charsets()
```

See Also:

[set_charsets](#), [set_default_charsets](#)

2.0.0.279 get_def_lang

```
include std/locale.e
public function get_def_lang()
```

Gets the default language (translation) map

Parameters:

none.

Returns:

An **object**, a language map, or zero if there is no default language map yet.

Example:

```
object langmap = get_def_lang()
```

2.0.0.280 get_dstring

```
include std/io.e  
public function get_dstring(integer fh, integer delim = 0)
```

Read a delimited byte string from an opened file .

Parameters:

1. `fh` : an integer, the handle to an open file to read from.
2. `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.

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
```

See Also:

[getc](#), [gets](#), [get_bytes](#), [get_integer32](#)

2.0.0.281 `get_encoding_properties`

```
include std/text.e
public function get_encoding_properties()
```

Gets the table of lowercase and uppercase characters that is used by `lower` and `upper`

Parameters:

none

Returns:

A **sequence**, containing three items.
{Encoding_Name, LowerCase_Set, UpperCase_Set}

Example 1:

```
encode_sets = get_encoding_properties()
```

See Also:

[lower](#), [upper](#), [set_encoding_properties](#)

2.0.0.282 `get_http`

```
include std/net/http.e
public function get_http(sequence inet_addr, sequence hostname, sequence file, integer timeout)
```

Returns data from an http internet site.

Parameters:

1. `inet_addr` : a sequence holding an address
2. `hostname` : a string, the name for the host
3. `file` : a file name to transmit

Returns:

A **sequence**, empty sequence on error, of length 2 on success, like {sequence header, sequence data}.

2.0.0.283 get_http_use_cookie

```
include std/net/http.e
public function get_http_use_cookie(sequence inet_addr, sequence hostname, sequence file)
```

Works the same as [get_url\(\)](#), but maintains an internal state register based on cookies received.

Warning:

This function is not yet implemented.

Parameters:

1. `inet_addr` : a sequence holding an address
2. `hostname` : a string, the name for the host
3. `file` : a file name to transmit

Returns:

A **sequence**, {header, body} on success, or an empty sequence on error.

Example 1:

```
addrinfo = getaddrinfo("www.yahoo.com", "http", 0)
if atom(addrinfo) or length(addrinfo) < 1 or
   length(addrinfo[1]) < 5 then
   puts(1, "Uh, oh")
   return {}
else
   inet_addr = addrinfo[1][5]
end if
data = get_http_use_cookie(inet_addr, "www.yahoo.com", "")
```

See also:

[get_url](#)

2.0.0.284 get_integer16

```
include std/io.e
public function get_integer16(integer fh)
```

Read the next two bytes from a file and returns them as a single integer.

Parameters:

Parameters:

1. `fh` : an integer, the handle to an open file to read from.

Returns:

An **atom**, made of the bytes that could be read from the file.

Comments:

- This function is normally used with files opened in binary mode, "rb".
- Assumes that there at least two bytes available to be read.

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

atom file_type_code
file_type_code = get_integer16(fn)
```

See Also:

[getc](#), [gets](#), [get_bytes](#), [get_dstring](#)

2.0.0.285 get_integer32

```
include std/io.e
public function get_integer32(integer fh)
```

Read the next four bytes from a file and returns them as a single integer.

Parameters:

1. `fh` : an integer, the handle to an open file to read from.

Returns:

An **atom**, made of the bytes that could be read from the file.

Comments:

- This function is normally used with files opened in binary mode, "rb".
- Assumes that there at least four bytes available to be read.

Example 1:

```
integer fn
fn = open("temp", "rb")  -- an existing file

atom file_type_code
file_type_code = get_integer32(fn)
```

See Also:

[getc](#), [gets](#), [get_bytes](#), [get_dstring](#)

2.0.0.286 get_key

<built-in> [function](#) `get_key()`

Get the next keystroke without waiting for it or echoing it on the console.

Parameters:

1. None.

Returns:

An **integer**, the code number for the key pressed. If there is no key press waiting, then this returns -1.

See Also:

[gets](#), [getc](#)

2.0.0.287 get_key

<built-in> [function](#) `get_key()`

Return the key that was pressed by the user, without waiting. Special codes are returned for the function keys, arrow keys etc.

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.

Example 1:

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

See Also:

[wait_key](#)

2.0.0.288 get_lang_path

```
include std/locale.e
public function get_lang_path()
```

Get the language path.

Returns:

An **object**, the current language path.

See Also:

[get_lang_path](#)

2.0.0.289 get_mouse

```
include std/mouse.e
public function get_mouse()
```

Parameters:

Queries the last mouse event.

Returns:

An **object**, either -1 if there has not been a mouse event since the last time `get_mouse()` was called. Otherwise, returns a triple `{event, x, y}`.

Constants have been defined in `mouse.e` for the possible mouse events (the values for `event`):

```
public constant
    MOVE = 1,
    LEFT_DOWN = 2,
    LEFT_UP = 4,
    RIGHT_DOWN = 8,
    RIGHT_UP = 16,
    MIDDLE_DOWN = 32,
    MIDDLE_UP = 64
```

`x` and `y` are the coordinates of the mouse pointer at the time that the event occurred.

Comments:

`get_mouse()` returns immediately with either a -1 or a mouse event, without waiting for an event to occur. So, you must check it frequently enough to avoid missing an event: when the next event occurs, the current event will be lost, if you haven't read it. In practice it is not hard to catch almost all events. Losing a MOVE event is generally not too serious, as the next MOVE will tell you where the mouse pointer is.

Sometimes multiple events will be reported. For example, if the mouse is moving when the left button is clicked, `get_mouse()` will report an event value of `LEFT_DOWN+MOVE`, i.e. 2+1 or 3. For this reason you should test for a particular event using `and_bits()`. See examples below. Further, you can determine which events will be reported using `mouse_events`.

In *Linux*, no scaling is required - `x` and `y` correspond to the line and column on the screen, with (1,1) at the top left.

In *Linux*, mouse movement events are not reported in an xterm window, only in the text console.

In *Linux*, `LEFT_UP`, `RIGHT_UP` and `MIDDLE_UP` are not distinguishable from one another.

The first call that you make to `get_mouse()` will turn on a mouse pointer, or a highlighted character.

The `x,y` coordinate returned could be that of the very tip of the mouse pointer or might refer to the pixel pointed-to by the mouse pointer.

Example 1:**a return value of:**

{2, 100, 50} would indicate that the left button was pressed down when the mouse pointer was at location x=100, y=50 on the screen.

Example 2:

To test for LEFT_DOWN, write something like the following:

```
while 1 do
  object event = get_mouse()
  if sequence(event) then
    if and_bits(event[1], LEFT_DOWN) then
      -- left button was pressed
      exit
    end if
  end if
end while
```

See Also:

[mouse_events](#), [mouse_pointer](#)

2.0.0.290 get_option

```
include std/socket.e
public function get_option(socket sock, integer level, integer optname)
```

Get options for a socket.

Parameters:

1. sock : the socket
2. level : an integer, the option level
3. 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**.

Returns:

An **atom**, On error, an atom indicating the error code.

A **sequence** or **atom**, On success, either an atom or a sequence containing the option value.

See also:

[get_option](#)

2.0.0.291 get_ovector_size

```
include std/regex.e
public function get_ovector_size(regex ex, integer maxsize = 0)
```

Returns the number of capturing subpatterns (the ovector size) for a regex

Parameters:

1. `ex` : a regex
2. `maxsize` : optional maximum number of named groups to get data from

Returns:

An **integer**

2.0.0.292 get_page_size

```
include std/unix/mmap.e
public function get_page_size()
```

2.0.0.293 get_pid

```
include std/os.e
public function get_pid()
```

Parameters:

Return the ID of the current Process (pid)

Returns:

An atom: The current process' id.

Example:

```
mypid = get_pid()
```

2.0.0.294 get_position

```
include std/graphics.e  
public function get_position()
```

Return the current line and column position of the cursor

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](#)

2.0.0.295 get_recvheader

```
include std/net/http.e  
public function get_recvheader(object field)
```

Return the value of a named field in the received http header as returned by the most recent call to [get_http](#).

Parameters:

1. `field`: an object, either a string holding a field name (case insensitive), 0 to return the whole header, or a numerical index.

Returns:

An **object**,

- -1 on error
 - a sequence in the form, {`field name`, `field value`} on success.
-
-

2.0.0.296 get_screen_char

```
include std/console.e
public function get_screen_char(positive_atom line, positive_atom column, integer fgbg = 0)
```

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

Parameters:

1. `line`: the 1-base line number of the location
2. `column`: the 1-base column number of the location
3. `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.
- The `fg_color` and `bg_color` are integers in the range 0 to 15, which correspond to...

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

- With `get_screen_char()` and `put_screen_char()` you can save and restore a character on the screen along with its `attribute_code`.

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}
```

See Also:

[put_screen_char](#), [save_text_image](#)

2.0.0.297 `get_sendheader`

```
include std/net/http.e
public function get_sendheader(object field)
```

Retrieve either the whole sendheader sequence, or just a single field.

Parameters:

Parameters:

1. `field`: an object indicating which part is being requested, see Comments section.

Returns:

An **object**, either:

- -1 if the field cannot be found,
- `{{"label", "delimiter", "value"}, ...}` for the whole sendheader sequence
- a three-element sequence in the form `{"label", "delimiter", "value"}` when only a single field is selected.

Comments:

`field` can be either an `HTTP_HEADER_xxx` access constant, the number 0 to retrieve the whole sendheader sequence, or a string matching one of the header field labels. The string is not case sensitive.

2.0.0.298 get_text

```
include std/text.e
public function get_text(integer MsgNum, sequence LocalQuals = {}, sequence DBBase = "teksto")
```

Get the text associated with the message number in the requested locale.

Parameters:

1. `MsgNum`: An integer. The message number whose text you are trying to get.
2. `LocalQuals`: A sequence. Zero or more locale codes. Default is `{}`.
3. `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.
An **integer**, if not associated text can be found.

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.
-
-
-
-
-

2.0.0.299 get_url

```
include std/net/http.e
public function get_url(sequence url, sequence post_data = "")
```

Returns data from an http internet site.

Parameters:

1. url: URL to access
2. post_data: Optional post data

Returns:

A **sequence** {header, body} on success, or an empty sequence on error.

Comments:

If post_data is empty, then a normal GET request is done. If post_data is non-empty then get_url will perform a POST request and supply post_data during the request.

Example 1:

```
url = "http://banners.wunderground.com/weathersticker/mini" &
      "Weather2_metric_cond/language/www/US/PA/Philadelphia.gif"

temp = get_url(url)
if length(temp)>=2 and length(temp[2])>0 then
    tempfp = open(TEMPDIR&"current_weather.gif", "wb")
    puts(tempfp,temp[2])
    close(tempfp)
end if
```

2.0.0.300 `getc`

```
<built-in> function getc(integer fn)
```

Get the next character (byte) from a file or device `fn`.

Parameters:

1. `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.

Errors:

The target file or device must be open.

Comments:

File input using `getc()` is buffered, i.e. `getc()` does not actually go out to the disk for each character. Instead, a large block of characters will be read in at one time and returned to you one by one from a memory buffer.

When `getc()` reads from the keyboard, it will not see any characters until the user presses Enter. Note that the user can type CTRL+Z, which the operating system treats as "end of file". **EOF** will be returned.

See Also:

[gets](#), [get_key](#)

2.0.0.301 `getenv`

```
<built-in> function getenv(sequence var_name)
```

Return the value of an environment variable.

Parameters:

1. `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.

Example:

```
e = getenv("EUDIR")
-- e will be "C:\EUPHORIA" -- or perhaps D:, E: etc.
```

See Also:

[setenv](#), [command_line](#)

2.0.0.302 gets

```
<built-in> function gets(integer fn)
```

Get the next sequence (one line, including '\n') of characters from a file or device.

Parameters:

1. `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.

Errors:

The file or device must be open.

Comments:

The characters will have values from 0 to 255.

If the line had an end of line marker, a ~'\n' terminates the line. The last line of a file needs not have an end of line marker.

After reading a line of text from the keyboard, you should normally output a \n character, e.g. 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.

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")
```

See Also:

[getc](#), [read_lines](#)

2.0.0.303 graphics_mode

```
include std/graphics.e
public function graphics_mode(mode m = - 1)
```

Attempt to set up a new graphics mode.

Parameters:

1. `m` : an integer, 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](#)

2.0.0.304 graphics_point

```
include std/image.e
public type graphics_point(sequence p)
```

2.0.0.305 harmean

```
include std/stats.e
public function harmean(sequence data_set, object subseq_opt = ST_ALLNUM)
```

Returns the harmonic mean of the atoms in a sequence.

Parameters:

1. `data_set` : the values to take the harmonic mean of.
2. `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.

Example 1:

```
? harmean({3, "abc", -2, 6}, ST_IGNSTR) -- = 0.  
? harmean({{2, 3, 4}}) -- 3 / (1/2 + 1/3 + 1/4) = 2.769230769
```

See Also:

[average](#)

2.0.0.306 has

```
include std/map.e  
public function has(map the_map_p, object the_key_p)
```

Check whether map has a given key.

Parameters:

1. `the_map_p` : the map to inspect
2. `the_key_p` : an object to be looked up

Returns:

An **integer**, 0 if not present, 1 if present.

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
```

See Also:

[get](#)

2.0.0.307 has_inverse

```
include std/sets.e
public function has_inverse(integer x, operation f)
```

Returns the bilateral inverse of an element by a operation if it exists and the operation has a unit.

Parameters:

1. `x` : the element to test
2. `f` : the operation involved.

Returns:

If `f`, has a bilateral unit `e` and there is a (necessarily unique) `y` such that `f(x, y) = e`, `y` is returned. Otherwise, 0 is returned..

Example 1:

```
operation f = {{{1, 2, 3}, {2, 3, 1}, {3, 1, 2}}, {3, 3, 3}}
? has_inverse(3, f) -- prints out 2.
```

See Also:

[has_unit](#)

2.0.0.308 has_match

```
include std/regex.e
public function has_match(regex re, string haystack, integer from = 1, option_spec options = DE
```

Determine if `re` matches any portion of `haystack`.

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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.

2.0.0.309 has_unit

```
include std/sets.e
public function has_unit(operation f, integer flags = SIDE_BOTH)
```

Returns an unit of a given kind for an operation if there is any, else 0.

Parameters:

1. `f` : the operation to test.
2. `flags` : an integer, which says whether one or two sided units are looked for. Defaults to `SIDE_BOTH`.

Returns:

An **integer**, if `f` has a unit of the requested type, it is returned. Otherwise, 0 is returned..

Parameters:

Comments:

If there is a two sided inverse, it is unique.

Only the two lower bits of `flags` matter. They must be `SIDE_LEFT` to check for left units, `SIDE_RIGHT` for right units. Otherwise, two sided units are determined.

Example 1:

```
operation f = {{{1,2,3},{2,3,1},{3,1,2}},{3,3,3}}
  ? has_unit(f)  -- prints out 1.
```

See Also:

[all_left_units](#), [all_right_units](#), [is_unit](#)

2.0.0.310 hash

```
<built-in> function hash(object source, atom algo)
```

Calculates a hash value from *key* using the algorithm *algo*

Parameters:

1. *source* : Any Euphoria object
2. *algo* : A code indicating which algorithm to use.
 - ◆ -5 uses Hsieh. Fastest and good dispersion
 - ◆ -4 uses Fletcher. Very fast and good dispersion
 - ◆ -3 uses Adler. Very fast and reasonable dispersion, especially for small strings
 - ◆ -2 uses MD5 (not implemented yet) Slower but very good dispersion. Suitable for signatures.
 - ◆ -1 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 ($\text{hash} = \text{hash} * \text{algo} + c$). 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 **integer**, Except for the MD5 and SHA256 algorithms, this is a 32-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:

```
x = hash("The quick brown fox jumps over the lazy dog", 0)
-- x is 242399616
x = hash("The quick brown fox jumps over the lazy dog", 99.94)
-- x is 723158
x = hash("The quick brown fox jumps over the lazy dog", -99.94)
-- x is 4175585990
x = hash("The quick brown fox jumps over the lazy dog", -4)
-- x is 467406810
```

2.0.0.311 head

```
<built-in> function head(sequence source, atom size=1)
```

Return the first *size* item(s) of a sequence.

Parameters:

1. *source* : the sequence from which elements will be returned
2. *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.

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"}
```

See Also:

[tail](#), [mid](#), [slice](#)

2.0.0.312 hex_text

```
include std/convert.e
public function hex_text(sequence text)
```

Convert a text representation of a hexadecimal number to an atom

Parameters:

1. `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.

Example 1:

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

See Also:

[value](#)

2.0.0.313 host_by_addr

```
include std/net/dns.e
public function host_by_addr(sequence address)
```

Get the host information by address.

Parameters:

1. address : host address

Returns:

A sequence, containing

```
{
  official name,
  { alias1, alias2, ... },
  { ip1, ip2, ... },
  address_type
}
```

Example 1:

```
object data = host_by_addr("74.125.93.147")
-- data = {
--   "www.l.google.com",
--   {
--     "www.google.com"
--   },
--   {
--     "74.125.93.104",
--     "74.125.93.147",
--     ...
--   },
--   2
-- }
```

2.0.0.314 host_by_name

```
include std/net/dns.e
public function host_by_name(sequence name)
```

Get the host information by name.

Parameters:

1. name : host name

Returns:

A sequence, containing

```
{
  official name,
  { alias1, alias2, ... },
  { ip1, ip2, ... },
  address_type
}
```

Example 1:

```
object data = host_by_name("www.google.com")
-- data = {
--   "www.l.google.com",
--   {
--     "www.google.com"
--   },
--   {
--     "74.125.93.104",
--     "74.125.93.147",
--     ...
--   },
--   2
-- }
```

2.0.0.315 iff

```
include std/utils.e
public function iff(atom test, object ifTrue, object ifFalse)
```

Used to embed an 'if' test inside an expression.

Parameters:

1. `test` : an atom, the result of a boolean expression
2. `ifTrue` : an object, returned if `test` is **non-zero**
3. `ifFalse` : an object, returned if `test` is zero

Returns:

An object. Either `ifTrue` or `ifFalse` is returned depending on the value of `test`.

Example 1:

```
msg = sprintf("%s: %s", {
iff(ErrType = 'E', "Fatal error", "Warning"),
    errortext } )
```

2.0.0.316 image

```
include std/sets.e
public function image(map f, object x, set input, set output)
```

If an object is in some input set, returns how it is mapped to a set.

Parameters:

1. `f` : the map to apply
2. `x` : the object to apply `f` to
3. `input` : the source set
4. `output` : the target set.

Returns:

An **object**, `f(x)` if it can be reckoned.

Errors:

`x` must belong to `input` for `f(x)` to be computed. `f` must not map to sets larger than `output`; otherwise, it cannot be defined from `input` to `output`.

Example 1:

```
map f={3,1,2,2,4,3}
set s1,s2
s1={"Albert","Beatrix","Conrad","Doris"} s2={13,17,19}
object x = image(f,"Conrad",s1,s2)
-- x is now 17.
```

See Also:

[direct_map](#)

2.0.0.317 include_paths

```
<built-in> function include_paths(integer convert)
```

Returns the list of include paths, in the order in which they are searched

Parameters:

1. `convert` : an integer, nonzero to include converted path entries that were not validated yet.

Returns:

A **sequence**, of strings, each holding a fully qualified include path.

Comments:

`convert` is checked only under *Windows*. If a path has accented characters in it, then it may or may not be valid to convert those to the OEM code page. Setting `convert` to a nonzero value will force conversion for path entries that have accents and which have not been checked to be valid yet. The extra entries, if any, are returned at the end of the returned sequence.

The paths are ordered in the order they are searched:

1. current directory
2. configuration file,
3. command line switches,
4. EUINC
5. a default based on EUDIR.

Example 1:

```
sequence s = include_paths(0)
-- s might contain
{
  "/usr/euphoria/tests",
  "/usr/euphoria/include",
  "./include",
  "../include"
}
```

See Also:

[eu.cfg](#), [include](#), [option_switches](#)

2.0.0.318 Notes

Due to a bug, Euphoria does not handle STDERR properly STDERR cannot be captured for Euphoria programs (other programs will work fully) The IO 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.

2.0.0.319 info

```
include std/safe.e
public function info()
```

2.0.0.320 init_class

```
include syncolor.e
public procedure init_class()
```

2.0.0.321 init_curdir

```
include std/filesys.e
public function init_curdir()
```

Returns the original current directory

Parameters:

1. None.

Returns:

A **sequence**, the current directory at the time the program started running.

Comment:

You would use this if the program might change the current directory during its processing and you wanted to return to the original directory.

Note:

This always ensures that the returned value has a trailing SLASH character.

Example 1:

```
res = init_curdir() -- Find the original current directory.
```

2.0.0.322 insert

```
<built-in> function insert(sequence target, object what, integer index)
```

Insert an object into a sequence as a new element at a given location.

Parameters:

1. `target` : the sequence to insert into
2. `what` : the object to insert
3. `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`.

`insert()`ing a sequence into a string returns a sequence which is no longer a string.

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}
```

See Also:

[remove](#), [splice](#), [append](#), [prepend](#)

2.0.0.323 `insertion_sort`

```
include std/sort.e
public function insertion_sort(sequence s, object e = "", integer compfunc = - 1, object userda
```

Sort a sequence, and optionally another object together.

Parameters:

1. `s` : a sequence, holding data to be sorted.
2. `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.
3. `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 (mostly) 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 comparison 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 appear after the second one, and 0 if the order doesn't matter.

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
```

See Also:

[compare](#), [sort](#), [merge](#)

2.0.0.324 instance

```
include std/os.e
public function instance()
```

Return hInstance on *Windows* and Process ID (pid) on *Unix*.

Comments:

On *Windows* the hInstance can be passed around to various *Windows* routines.

2.0.0.325 int_to_bits

```
include std/convert.e
public function int_to_bits(atom x, integer nbits = 32)
```

Extracts the lower bits from an integer.

Parameters:

1. `x` : the atom to convert
2. `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.

Example 1:

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

See Also:

[bits_to_int](#), [int_to_bytes](#), [Relational operators](#), [operations on sequences](#)

2.0.0.326 int_to_bytes

```
include std/convert.e
public function int_to_bytes(atom x)
```

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

Parameters:

1. `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 $\text{floor}(x/\text{power}(2, 24))$. 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+.

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}
```

See Also:

[bytes_to_int](#), [int_to_bits](#), [atom_to_float64](#), [poke4](#)

2.0.0.327 intdiv

```
include std/math.e
public function intdiv(object a, object b)
```

Return an integral division of two objects.

Parameters:

1. `divided`: any Euphoria object.
2. `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
```

2.0.0.328 integer

```
<built-in> function integer(object x)
```

Tests the supplied argument `x` to see if it is an integer or not.

Returns:

1. An integer.
 - ◆ 1 if `x` is an integer.
 - ◆ 0 if `x` is not an integer.

Example 1:

```
? integer(1) --> 1
? integer(1.1) --> 0
? integer("1") --> 0
```

See Also:

[sequence\(\)](#), [object\(\)](#), [atom\(\)](#)

2.0.0.329 integer_array

```
include std/types.e
public type integer_array(object x)
```

Parameters:

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
```

2.0.0.330 intersection

```
include std/sets.e
public function intersection(set S1, set S2)
```

Returns the set of elements belonging to both s1 and s2.

Parameters:

1. S1 : One of the sets to intersect
2. S2 : the other set.

Returns:

A set, made of all elements belonging to both S1 and S2.

Example 1:

```
set s0,s1,s2
s1={1,3,5,7} s2={-1,2,3,7,11}
s0=intersection(s1,s2)  -- s0 is now {3,7}.
```

See Also:

[is_subset](#), [subsets](#), [belongs_to](#)

2.0.0.331 is_DEP_supported

```
include std/machine.e
public function is_DEP_supported()
```

2.0.0.332 is_associative

```
include std/sets.e
public function is_associative(operation f)
```

Determine whether the identity $f(f(x,y),z)=f(x,f(y,z))$ always makes sense and holds.

Parameters:

1. f : the operation to test.

Returns:

An **integer**, 1 if f is an internal operation on a set and is associative, else 0.

Comments:

Being associative is equivalent to not depending on parentheses for defining iterated execution.

Example 1:

```
operation f = {{{1, 2, 3}, {2, 3, 1}, {3, 1, 2}}, {3, 3, 3}}
-- f is the addition modulo 3 from {0, 1, 2} x {0, 1, 2} to {0, 1, 2}.
? is_symmetric(f)  -- prints out 1.
```

See Also:

[operation](#), [has_unit](#)

2.0.0.333 is_bijective

```
include std/sets.e
public function is_bijective(map f)
```

Determine whether a map is one-to-one.

Parameters:

1. `f` : the map to test.

Returns:

An **integer**, 1 if `f` is one-to-one, else 0.

Example 1:

```
map f = {2,3,1,1,2,5,3}
? is_surjective(f) -- prints out 0
```

See Also:

[is_surjective](#), [is_bijective](#), [direct_map](#), [has_inverse](#)

2.0.0.334 is_empty

```
include std/stack.e
public function is_empty(stack sk)
```

Determine whether a stack is empty.

Parameters:

1. `sk` : the stack being queried.

Returns:

An **integer**, 1 if the stack is empty, else 0.

See Also:

[size](#)

2.0.0.335 is_even

```
include std/math.e
public function is_even(integer test_integer)
```

Test if the supplied integer is a even or odd number.

Parameters:

1. `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}
```

2.0.0.336 is_even_obj

```
include std/math.e
public function is_even_obj(object test_object)
```

Test if the supplied Euphoria object is even or odd.

Parameters:

1. `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
```

2.0.0.337 is_in_list

```
include std/search.e
public function is_in_list(object item, sequence list)
```

Tests to see if the `item` is in a list of values supplied by `list`

Parameters:

1. `item`: The object to test for.
2. `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
```

2.0.0.338 is_in_range

```
include std/search.e
public function is_in_range(object item, sequence range_limits, sequence boundries = "[")
```

Tests to see if the `item` is in a range of values supplied by `range_limits`

Parameters:

1. `item`: The object to test for.
2. `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.
3. `boundries`: a sequence. This determines if the range limits are inclusive or not. Must be one of "[", "]", "(", 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 "[", "]", "(", and ")" 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
```

2.0.0.339 is_inetaddr

```
include std/net/common.e
public function is_inetaddr(sequence address)
```

Checks if x is an IP address in the form (#.#.#.#[:#])

Parameters:

1. address : the address to check

Returns:

An **integer**, 1 if x is an inetaddr, 0 if it is not

Comments:

Some ip validation algorithms do not allow 0.0.0.0. We do here because many times you will want to bind to 0.0.0.0. However, you cannot connect to 0.0.0.0 of course.

With sockets, normally binding to 0.0.0.0 means bind to all interfaces that the computer has.

2.0.0.340 is_injective

```
include std/sets.e
public function is_injective(map f)
```

Determines whether there is a point in an output set hit twice or more by a map.

Parameters:

1. f : the map being queried.

Returns:

An **integer**, 0 if f ever maps two points to the same element, else 1.

Example 1:

```
map f = {2,3,1,1,2,5,3}
?is_injective(f) -- prints out 0
```

See Also:

[is_surjective](#), [is_bijective](#), [reverse_map](#), [fiber_over](#)

2.0.0.341 is_leap_year

```
include std/datetime.e
public function is_leap_year(datetime dt)
```

Determine if dt falls within leap year.

Parameters:

1. dt : a datetime to be queried.

Returns:

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

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
```

See Also:

[days_in_month](#)

2.0.0.342 is_match

```
include std/regex.e
public function is_match(regex re, string haystack, integer from = 1, option_spec options = DEF
```

Determine if the entire `haystack` matches `re`.

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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.

2.0.0.343 is_match

```
include std/wildcard.e
public function is_match(sequence pattern, sequence string)
```

Determine whether a string matches a pattern. The pattern may contain `*` and `?` wildcards.

Parameters:

1. `pattern` : a string, the pattern to match
2. `string` : the string to be matched against

Returns:

An **integer**, TRUE if `string` matches `pattern`, else FALSE.

Comments:

Character comparisons are case sensitive. If you want case insensitive comparisons, pass both `pattern` and `string` through **upper()**, or both through **lower()**, before calling `is_match()`.

If you want to detect a pattern anywhere within a string, add `*` to each end of the pattern:

```
i = is_match('*' & pattern & '*', string)
```

There is currently no way to treat * or ? literally in a pattern.

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:

bin/search.ex

See Also:

[wildcard_file](#), [upper](#), [lower](#), [Regular Expressions](#)

2.0.0.344 is_page_aligned_address

```
include std/unix/mmap.e
public function is_page_aligned_address(atom a)
```

2.0.0.345 is_subset

```
include std/sets.e
public function is_subset(set small, set large)
```

Checks whether a set is a subset of another.

Parameters:

Parameters:

1. `small` : the set to test
2. `large` : the supposedly larger set.

Returns:

An **integer**, 1 if `small` is a subset of `large`, else 0.

Example 1:

```
set s0 = {1,3,5,7}
? is_subset({3,5},s0)  -- prints out 1
```

See Also:

[subsets](#), [belongs_to](#), [difference](#), [embedding](#), [embed_union](#)

2.0.0.346 is_surjective

```
include std/sets.e
public function is_surjective(map f)
```

Determine whether all points in the output set are hit by a map.

Parameters:

1. `f` : the map to test.

Returns:

An **integer**, 0 if `f` ever misses some point in the target set, else 1.

Example 1:

```
map f = {2,3,1,1,2,5,3}
?is_surjective(f)  -- prints out 1
```

See Also:

[is_surjective](#), [is_bijective](#), [direct_map](#), [section](#)

2.0.0.347 is_symmetric

```
include std/sets.e
public function is_symmetric(operation f)
```

Determine whether $f(x,y)$ always equals $f(y,x)$.

Parameters:

1. f : the operation to test.

Returns:

An **integer**, 1 if exchanging operands makes sense and has no effect, else 0.

Example 1:

```
operation f = {{1,2,3},{2,3,4},{3,4,5}},{3,3,5}}
-- f is the addition from {0,1,2}x{0,1,2} to {0,1,2,3,4}.
? is_symmetric(f)  -- prints out 1.
```

See Also:

[operation](#), [has_unit](#)

2.0.0.348 is_unit

```
include std/sets.e
public function is_unit(integer x, operation f)
```

Determines if an element is a (one sided) unit for an operation.

Parameters:

1. x : an integer, the element to test
2. f : the operation involved

Returns:

An **integer**, either `SIDE_NONE`, `SIDE_LEFT`, `SIDE_RIGHT` or `SIDE_BOTH`.

Example 1:

```
operation f = {{1, 2, 3}, {1, 2, 3}, {3, 1, 2}}, {3, 3, 3 }}
? is_left_unit(3, f)  -- prints out 0.
```

See Also:

[all_left_units](#), [has_unit](#)

2.0.0.349 is_using_DEP

```
include std/machine.e
public function is_using_DEP()
```

2.0.0.350 is_valid_memory_protection_constant

```
include std/unix/mmap.e
public function is_valid_memory_protection_constant(integer x)
```

2.0.0.351 is_win_nt

```
include std/os.e
public function is_win_nt()
```

Decides whether the host system is a newer Windows version (NT/2K/XP/Vista).

Returns:

An **integer**, 1 if host system is a newer Windows (NT/2K/XP/Vista), else 0.

2.0.0.352 join

```
include std/sequence.e
public function join(sequence items, object delim = " ")
```

Join sequences together using a delimiter.

Parameters:

1. `items` : the sequence of items to join.
2. `delim` : an object, the delimiter to join by. Defaults to " ".

Comments:

This function may be applied to a string sequence or a complex sequence

Example 1:

```
result = join({"John", "Middle", "Doe"})
-- result is "John Middle Doe"
```

Example 2:

```
result = join({"John", "Middle", "Doe"}, ",")
-- result is "John,Middle,Doe"
```

See Also:

[split](#), [split_any](#), [breakup](#)

2.0.0.353 kernel_dll

```
include std/memconst.e
export atom kernel_dll
```

2.0.0.354 keys

```
include std/map.e
public function keys(map the_map_p, integer sorted_result = 0)
```

Return all keys in a map.

Parameters:

1. `the_map_p`: the map being queried
2. `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 predictable.

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}
```

See Also:

[has](#), [values](#), [pairs](#)

2.0.0.355 keyvalues

```
include std/text.e
public function keyvalues(sequence source, object pair_delim = ";,", object kv_delim = ":", ob
```

Converts a string containing Key/Value pairs into a set of sequences, one per K/V pair.

Parameters:

1. `source` : a text sequence, containing the representation of the key/values.
2. `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 (,).
3. `kv_delim` : an object containing a list of elements that delimit the key from its value. The defaults are colon (:) and equal (=).
4. `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 (`).
5. `whitespace` : an object containing a list of elements that are regarded as whitespace characters. The defaults are space, tab, new-line, and carriage-return.
6. `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]"} } }
```

2.0.0.356 keywords

```
include keywords.e
public constant keywords
```

Sequence of Euphoria keywords

2.0.0.357 kill

```
include std/pipeio.e
public procedure kill(process p, atom signal = 15)
```

Close pipes and kill process p with signal signal (default 15)

Comments:

Signal is ignored on Windows.

Example 1:

```
kill(p)
```

Parameters:

2.0.0.358 kurtosis

```
include std/stats.e
public function kurtosis(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns a measure of the spread of values in a dataset when compared to a *normal* probability curve.

Parameters:

1. `data_set` : a list of 1 or more numbers whose kurtosis is required.
2. `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. Otherwise 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
```

See also:

[skewness](#)

For a complete overview of the task system, please see the mini-guide [Multitasking in Euphoria](#).

The task system does not yet function in a shared library. Task routine calls that are compiled into a shared library are emitted as a NOP (no operation) and will therefore have no effect.

It is planned to allow the task system to function in shared libraries in future versions of OpenEuphoria.

2.0.0.359 lang_load

```
include std/locale.e
public function lang_load(sequence filename)
```

Load a language file.

Parameters:

1. `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.

Example:

```
# 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
```

See Also:

[translate](#)

2.0.0.360 largest

```
include std/stats.e
public function largest(object data_set)
```

Returns the largest of the data points that are atoms.

Parameters:

1. `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: {}
```

See also:

[range](#)

2.0.0.361 last

```
include std/stack.e
public function last(stack sk)
```

Retrieve the end element on a stack.

Parameters:

1. `sk` : the stack to inspect.

Returns:

An **object**, the end element on a stack.

Comments:

This call is equivalent to `at(sk, 0)`.

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
```

See Also:

[at](#), [pop](#), [peek_end](#), [top](#)

2.0.0.362 length

```
<built-in> function length(object target)
```

Return the length of an object.

Parameters:

1. `target` : the object being queried

Returns:

An **integer**, the number of elements involved with `target`.

Comments:

- An atom only ever has a length of 1.
- The length of a sequence is the number of elements in the sequence.
- 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.

Example 1:

```
length({{1,2}, {3,4}, {5,6}}) -- 3
length("") -- 0
length({}) -- 0
length( 7 ) -- 1
length( 3.14 ) -- 1
```

See Also:

[append](#), [prepend](#), &

2.0.0.363 linear

```
include std/sequence.e
public function linear(object start, object increment, integer count)
```

Returns a sequence in arithmetic progression.

Parameters:

1. `start` : the initial value from which to start
2. `increment` : the value to recursively add to `start` to get new elements
3. `count` : an integer, the number of additions to perform.

Returns:

An **object**, either 0 on failure or `{start, start+increment, ..., start+count*increment}`

Comments:

If `count` is negative, or if adding `start` to `increment` would prove to be impossible, then 0 is returned. Otherwise, a sequence, of length `count+1`, starting with `start` and whose adjacent elements differ exactly by `increment`, is returned.

Example 1:

```
s = linear({1,2,3},4,3)
-- s is {{1,2,3},{5,6,7},{9,10,11}}
```

See Also:

[repeat_pattern](#)

2.0.0.364 listen

```
include std/socket.e
public function listen(socket sock, integer backlog)
```

Parameters:

Start monitoring a connection. Only works with TCP sockets.

Parameters:

1. `sock` : the socket
2. `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()`.

2.0.0.365 load

```
include std/serialize.e
public function load(sequence filename)
```

Restores a Euphoria object that has been saved to disk by `dump`.

Parameters:

1. `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 :

```
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
```

2.0.0.366 load_map

```
include std/map.e
public function load_map(object file_name_p)
```

Loads a map from a file

Parameters:

1. `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.

Example 1:

```
object loaded
map AppOptions
sequence SavedMap = "c:\myapp\options.txt"
loaded = load_map(SavedMap)
```

Parameters:

```
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
    ShowInstructions()
end if
```

See Also:

[new](#), [save_map](#)

2.0.0.367 locale_canonical

```
include std/localeconv.e
public constant locale_canonical
```

2.0.0.368 locate_file

```
include std/filesys.e
public function locate_file(sequence filename, sequence search_list = {}, sequence subdir = {})
```

Locates a file by looking in a set of directories for it.

Parameters:

1. `filename` : a sequence, the name of the file to search for.
2. `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.
3. `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
```

2.0.0.369 lock_file

```
include std/io.e
public function lock_file(file_number fn, lock_type t, byte_range r = {})
```

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.

Parameters:

1. `fn` : an integer, the handle to the file or device to (partially) lock.
2. `t` : an integer which defines the kind of lock to apply.
3. `r` : a sequence, defining a section of the file to be locked, or { } for the whole file (the default).

Returns:

An **integer**, 0 on failure, 1 on success.

Errors:

The target file or device must be open.

Comments:

`lock_file()` attempts to place a lock on an open file, `fn`, to stop other processes from using the file while your program is reading it or writing it.

Under *Unix*, there are two types of locks that you can request using the `t` parameter. (Under *WIN32* the parameter `t` is ignored, but should be an integer.) Ask for a **shared** lock when you intend to read a file, and you want to temporarily block other processes from writing it. Ask for an **exclusive** lock when you intend to write to a file and you want to temporarily block other processes from reading or writing it. It's ok for many processes to simultaneously have shared locks on the same file, but only one process can have an exclusive lock, and that can happen only when no other process has any kind of lock on the file. `io.e` contains the following declarations:

```
public enum
    LOCK_SHARED,
    LOCK_EXCLUSIVE
```

On *WIN32* you can lock a specified portion of a file using the `r` parameter. `r` is a sequence of the form: `{first_byte, last_byte}`. It indicates the first byte and last byte in the file, that the lock applies to. Specify the empty sequence `{}`, if you want to lock the whole file, or don't specify it at all, as this is the default. In the current release for *Unix*, locks always apply to the whole file, and you should use this default value.

`lock_file()` does not wait for other processes to relinquish their locks. You may have to call it repeatedly, before the lock request is granted.

On *Unix*, these locks are called advisory locks, which means they aren't enforced by the operating system. It is up to the processes that use a particular file to cooperate with each other. A process can access a file without first obtaining a lock on it. On *WIN32* locks are enforced by the operating system.

Example 1:

```
include std/io.e
integer v
atom t
v = open("visitor_log", "a") -- open for append
t = time()
while not lock_file(v, LOCK_EXCLUSIVE, {}) do
    if time() > t + 60 then
        puts(STDOUT, "One minute already ... I can't wait forever!\n")
        abort(1)
    end if
    sleep(5) -- let other processes run
end while
puts(v, "Yet another visitor\n")
unlock_file(v, {})
```

Parameters:

`close(v)`

See Also:

[unlock_file](#)

2.0.0.370 lock_type

```
include std/io.e
public type lock_type(integer t)
```

Lock Type

2.0.0.371 log

<built-in> `function log(object value)`

Return the natural logarithm of a positive number.

Parameters:

1. `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.

Errors:

If any atom in `value` is not greater than zero, an error occurs as its logarithm is not defined.

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.

Example 1:

```
a = log(100)
-- a is 4.60517
```

See Also:

[E](#), [exp](#), [log10](#)

2.0.0.372 log10

```
include std/math.e
public function log10(object x1)
```

Return the base 10 logarithm of a number.

Parameters:

1. `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.

Errors:

If any atom in `value` is not greater than zero, its logarithm is not a real number and an error occurs.

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.

Example 1:

```
a = log10(12)
-- a is 2.48490665
```

See Also:

[log](#)

2.0.0.373 lookup

```
include std/search.e
public function lookup(object find_item, sequence source_list, sequence target_list, object def_value)
```

If the supplied item is in the source list, this returns the corresponding element from the target list.

Parameters:

1. `find_item`: an object that might exist in `source_list`.
2. `source_list`: a sequence that might contain `find_item`.
3. `target_list`: a sequence from which the corresponding item will be returned.
4. `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.

Examples:

```
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"
```

2.0.0.374 lower

```
include std/text.e
public function lower(object x)
```

Convert an atom or sequence to lower case.

Parameters:

Parameters:

1. `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 non-Windows, 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.

Example 1:

```
s = lower("Euphoria")
-- s is "euphoria"

a = lower('B')
-- a is 'b'

s = lower({"Euphoria", "Programming"})
-- s is {"euphoria", "programming"}
```

See Also:

[upper](#), [proper](#), [set_encoding_properties](#), [get_encoding_properties](#)

2.0.0.375 machine_addr

```
include std/memory.e
public type machine_addr(object a)
```

Machine address type

2.0.0.376 machine_addr

```
include std/safe.e
public type machine_addr(atom a)
```

Parameters:

2.0.0.377 machine_func

```
<built-in> function machine_func(integer machine_id, object args={})
```

Perform a machine-specific operation that returns a value.

Returns:

Depends on the called internal facility.

Comments:

This function is mainly used by the standard library files to implement machine dependent operations such as graphics and sound effects. This routine should normally be called indirectly via one of the library routines in a Euphoria include file. User programs normally do not need to call `machine_func`.

A direct call might cause a machine exception if done incorrectly.

See Also:

[machine_proc](#)

2.0.0.378 machine_proc

```
<built-in> procedure machine_proc(integer machine_id, object args={})
```

Perform a machine-specific operation that does not return a value.

Comments:

This procedure is mainly used by the standard library files to implement machine dependent operations such as graphics and sound effects. This routine should normally be called indirectly via one of the library routines in a Euphoria include file. User programs normally do not need to call `machine_proc`.

A direct call might cause a machine exception if done incorrectly.

See Also:

[machine_func](#)

2.0.0.379 malloc

```
include std/eumem.e
export function malloc(object mem_struct_p = 1, integer cleanup_p = 1)
```

Allocate a block of (pseudo) memory

Parameters:

1. `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}
2. `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
```

2.0.0.380 map

```
include std/map.e
public type map(object obj_p)
```

Defines the datatype 'map'

Comments:

Used when declaring a map variable.

Parameters:

Example:

```
map SymbolTable = new() -- Create a new map to hold the symbol table.
```

2.0.0.381 map

```
include std/sets.e
public type map(object s)
```

Returns 1 if a sequence of integers is a valid map descriptor, else 0.

Comments:

A map is a sequence of indexes. Each index is between 1 and the maximum allowed for the particular map.

Actually, what is being called a `map` is a class of maps, as the elements of the input sequence, except for the last two, are ordinals rather than set elements. A map contains the information required to map as expected the elements of a set, given by index, to another set, where the images are indexes again. Technically, those are maps of the category of finite sets quotiented by equality of cardinal.

The objects that `map.e` handle are completely unrelated to these.

Example 1:

```
sequence s0 = {2, 3, 4, 1, 4, 2, 6, 4}
? map(s0) -- prints out 1.
```

See Also:

[define_map](#), [fiber_over](#), [restrict](#), [direct_map](#), [reverse_map](#), [is_injective](#), [is_surjective](#), [is_bijective](#)

2.0.0.382 mapping

```
include std/sequence.e
public function mapping(object source_arg, sequence from_set, sequence to_set, integer one_level)
```

Each item from `source_arg` found in `from_set` is changed into the corresponding item in `to_set`

Parameters:

1. `source_arg` : Any Euphoria object to be transformed.
2. `from_set` : A sequence of objects representing the only items from `source_arg` that are actually transformed.
3. `to_set` : A sequence of objects representing the transformed equivalents of those found in `from_set`.
4. `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"
```

2.0.0.383 match

```
<built-in> function match(sequence needle, sequence haystack, integer start)
```

Try to match a "needle" against some slice of a "haystack", starting at position "start".

Parameters:

1. `needle` : a sequence whose presence as a "substring" is being queried
2. `haystack` : a sequence, which is being looked up for `needle` as a sub-sequence
3. `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.

Comments:

`match()` and `match_from()` are identical, but you can omit giving `match()` a starting point.

Example 1:

```
location = match("pho", "Euphoria")
-- location is set to 3
```

See Also:

[find](#), [find_from](#), [compare](#), [match_from](#), [wildcard:is_match](#)

2.0.0.384 match_all

```
include std/search.e
public function match_all(sequence needle, sequence haystack, integer start = 1)
```

Match all items of `haystack` in `needle`.

Parameters:

1. `needle` : a sequence, what to look for
2. `haystack` : a sequence to search in
3. `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.

Example 1:

```
s = match_all("the", "the dog chased the cat under the table.")
-- s is {1,16,30}
```

See Also:

[match](#), [regex:find_all](#) [find](#), [find_all](#)

2.0.0.385 match_any

```
include std/search.e
public function match_any(sequence needles, sequence haystack, integer start = 1)
```

Determines if any element from `needles` is in `haystack`.

Parameters:

1. `needles` : a sequence, the list of items to look for
2. `haystack` : a sequence, in which "needles" are looked for
3. `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.

Example 1:

```
ok = match_any("aeiou", "John Smith")
-- okay is 1
ok = match_any("xyz", "John Smith" )
-- okay is 0
```

See Also:

[find_any](#)

2.0.0.386 match_from

```
<built-in> function match_from(sequence needle, sequence haystack, integer start)
```

Try to match a "needle" against some slice of a "haystack", starting from some index.

Parameters:

1. `needle` : an sequence whose presence as a sub-sequence is being queried
2. `haystack` : a sequence, which is being looked up for `needle` as a sub-sequence
3. `start` : an integer, the index in `haystack` at which to start searching.

Returns:

An **integer**, 0 if no slice of `haystack` with lower index at least `start` is `needle`, else the smallest such index.

Comments:

`start` may have any value from 1 to the length of `haystack` plus 1. (Just like the first index of a slice of `haystack`.)

`match()` and `match_from()` are identical, but you can omit giving `match()` a starting point.

Example 1:

```
location = match_from("pho", "phoEuphoria", 4)
-- location is set to 6
```

See Also:

[find](#), [find_from](#), [match](#), [compare](#), [wildcard:is_match](#), [regex:find](#)

2.0.0.387 match_replace

```
include std/search.e
public function match_replace(object needle, sequence haystack, object replacement, integer max
```

Finds a "needle" in a "haystack", and replace any, or only the first few, occurrences with a replacement.

Parameters:

1. `needle` : an object to search and perhaps replace
2. `haystack` : a sequence to be inspected
3. `replacement` : an object to substitute for any (first) instance of `needle`
4. `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.

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"
```

See Also:

[find](#), [replace](#), [regex:find_replace](#), [find_replace](#)

2.0.0.388 matches

```
include std/regex.e
public function matches(regex re, string haystack, integer from = 1, option_spec options = DEFA
```

Get the matched text only.

Parameters:

1. `re` : a regex for a subject to be matched against
2. `haystack` : a string in which to searched
3. `from` : an integer setting the starting position to begin searching from. Defaults to 1
4. `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** if 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`.

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", re:STRING_OFFSETS)
-- matches is:
-- {
--   { "John Doe", 1, 8 }, -- full match data
--   { "John",     1, 4 }, -- first group
--   { "Doe",      6, 8 } -- second group
-- }
```

See Also:

[all_matches](#)

2.0.0.389 max

```
include std/math.e
public function max(object a)
```

Parameters:

Computes the maximum value among all the argument's elements

Parameters:

1. `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.

Example 1:

```
a = max({10,15.4,3})
-- a is 15.4
```

See Also:

[min](#), [compare](#), [flatten](#)

2.0.0.390 `maybe_any_key`

```
include std/console.e
public procedure maybe_any_key(sequence prompt = "Press Any Key to continue...", integer con =
```

Display a prompt to the user and wait for any key **only** if the user is running under a GUI environment.

Parameters:

1. `prompt` : Prompt to display, defaults to "Press Any Key to continue..."
2. `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.

Example 1:

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

Example 2:

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

See Also:

[wait_key](#)

2.0.0.391 median

```
include std/stats.e
public function median(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the mid point of the data points.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want the mean.
2. `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.

The equation for average is:

```
median(X) ==> sort(X)[N/2]
```

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:

```
? median( {7,2,8,5,6,6,4,8,6,6,3,3,4,1,8,4} ) -- Ans: 5
```

See also:

[average](#), [geomean](#), [harmean](#), [movavg](#), [emovavg](#)

2.0.0.392 memDLL_id

```
include std/memconst.e
export atom memDLL_id
```

2.0.0.393 mem_copy

```
<built-in> procedure mem_copy(atom destination, atom origin, integer len)
```

Copy a block of memory from an address to another.

Parameters:

1. `destination`: an atom, the address at which data is to be copied
2. `origin`: an atom, the address from which data is to be copied
3. `len`: an integer, how many bytes are to be copied.

Comments:

The bytes of memory will be copied correctly even if the block of memory at `destination` overlaps with the block of memory at `origin`.

`mem_copy(destination, origin, len)` is equivalent to: `poke(destination, peek({origin, len}))` but is much faster.

Example 1:

```
dest = allocate(50)
src = allocate(100)
poke(src, {1,2,3,4,5,6,7,8,9})
```

Parameters:

```
mem_copy(dest, src, 9)
```

See Also:

[mem_set](#), [peek](#), [poke](#), [allocate](#), [free](#)

2.0.0.394 mem_copy

```
include std/safe.e
override procedure mem_copy(machine_addr
```

2.0.0.395 mem_set

```
<built-in> procedure mem_set(atom destination, integer byte_value, integer how_many)
```

Sets a contiguous range of memory locations to a single value.

Parameters:

1. `destination`: an atom, the address starting the range to set.
2. `byte_value`: an integer, the value to copy at all addresses in the range.
3. `how_many`: an integer, how many bytes are to be set.

Comments:

The low order 8 bits of `byte_value` are actually stored in each byte. `mem_set(destination, byte_value, how_many)` is equivalent to: `poke(destination, repeat(byte_value, how_many))` but is much faster.

Example 1:

```
destination = allocate(1000)
mem_set(destination, ' ', 1000)
-- 1000 consecutive bytes in memory will be set to 32
-- (the ASCII code for ' ')
```

See Also:

[peek](#), [poke](#), [allocate](#), [free](#), [mem_copy](#)

2.0.0.396 mem_set

```
include std/safe.e
override procedure mem_set(machine_addr
```

2.0.0.397 memory_used

```
include std/safe.e
public function memory_used()
```

2.0.0.398 merge

```
include std/sort.e
public function merge(sequence a, sequence b, integer compfunc = - 1, object userdata = "")
```

Merge two pre-sorted sequences into a single sequence.

Parameters:

1. a : a sequence, holding pre-sorted data.
2. b : a sequence, holding pre-sorted data.
3. compfunc : an integer, either -1 or the routine id of a user-defined comparison 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 comparison 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 doesn't matter.

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}
```

See Also:

[compare](#), [sort](#)

2.0.0.399 message_box

```
include std/win32/msgbox.e
public function message_box(sequence text, sequence title, object style)
```

Displays a window with a title, message, buttons and an icon, usually known as a message box.

Parameters:

1. `text`: a sequence, the message to be displayed
2. `title`: a sequence, the title the box should have
3. `style`: an object which defines which icon should be displayed, if any, and which buttons will be presented.

Returns:

An **integer**, the button which was clicked to close the message box, or 0 on failure.

Comments:

See [Style Constants](#) above for a complete list of possible values for `style` and [Return Value Constants](#) for the returned value. If `style` is a sequence, its elements will be or'ed together.

2.0.0.400 mid

```
include std/sequence.e
public function mid(sequence source, atom start, atom len)
```

Returns a slice of a sequence, given by a starting point and a length.

Parameters:

1. `source` : the sequence some elements of which will be returned
2. `start` : an integer, the lower index of the slice to return

3. `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`.

Errors:

If `len` is less than `-length(source)`, an error occurs.

Comments:

`len` may be negative, in which case it is added `length(source)` once.

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}
```

See Also:

[head](#), [tail](#), [slice](#)

2.0.0.401 min

```
include std/math.e
public function min(object a)
```

Computes the minimum value among all the argument's elements

Parameters:

1. `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
```

2.0.0.402 minsize

```
include std/sequence.e
public function minsize(sequence source_data, integer min_size, object new_data)
```

Ensures that the supplied sequence is at least the supplied minimum length.

Parameters:

1. `source_data` : A sequence that might need extending.
2. `min_size`: An integer. The minimum length that `source_data` must be.
3. `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`.

Returns:

A **sequence**.

Example:

```
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}
```

2.0.0.403 mixture

```
include std/graphcst.e
public type mixture(sequence s)
```

Mixture Type**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}.

2.0.0.404 mlock

```
include std/unix/mmap.e
public function mlock(atom addr, integer length)
```

2.0.0.405 mmap

```
include std/unix/mmap.e
public function mmap(object start, integer length, valid_memory_protection_constant protection,
```

2.0.0.406 mod

```
include std/math.e
public function mod(object x, object y)
```

Compute the remainder of the division of two objects using floored division.

Parameters:

1. `dividend` : any Euphoria object.
2. `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/divisor` away from zero whereas `remainder()` rounds towards zero.

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}
```

See Also:

[remainder](#), [Relational operators](#), [Operations on sequences](#)

2.0.0.407 mode

```
include std/stats.e
public function mode(sequence data_set, object subseq_opt = ST_ALLNUM)
```

Returns the most frequent point(s) of the data set.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want the mode.
2. `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}
```

See also:

[average](#), [geomean](#), [harmean](#), [movavg](#), [emovavg](#)

2.0.0.408 money

```
include std/locale.e
public function money(object amount)
```

Converts an amount of currency into a string representing that amount.

Parameters:

1. `amount` : an atom, the value to write out.

Returns:

A **sequence**, a string that writes out amount of current currency.

Example 1:

```
-- Assuming an en_US locale
? money(1020.5) -- returns "$1,020.50"
```

See Also:

[set](#), [number](#)

2.0.0.409 month_abbrs

```
include std/datetime.e
public sequence month_abbrs
```

Abbreviations of month names

2.0.0.410 month_names

```
include std/datetime.e
public sequence month_names
```

Parameters:

2.0.0.411 mouse_events

```
include std/mouse.e
public procedure mouse_events(integer events)
```

Select the mouse events `get_mouse()` is to report.

Parameters:

1. `events`: an integer, all requested event codes or'ed together.

Comments:

By default, `get_mouse()` will report all events. `mouse_events()` can be called at various stages of the execution of your program, as the need to detect events changes. Under *Unix*, `mouse_events()` currently has no effect.

It is good practice to ignore events that you are not interested in, particularly the very frequent MOVE event, in order to reduce the chance that you will miss a significant event.

The first call that you make to `mouse_events()` will turn on a mouse pointer, or a highlighted character.

Example 1:

```
mouse_events(LEFT_DOWN + LEFT_UP + RIGHT_DOWN)
```

will restrict `get_mouse()` to reporting the left button being pressed down or released, and the right button being pressed down. All other events will be ignored.

See Also:

[get_mouse](#), [mouse_pointer](#)

2.0.0.412 mouse_pointer

```
include std/mouse.e
public procedure mouse_pointer(integer show_it)
```

Turn mouse pointer on or off.

Parameters:

1. `show_it` : an integer, 0 to hide and 1 to show.

Comments:

Multiple calls to hide the pointer will require multiple calls to turn it back on. The first call to either `get_mouse()` or `mouse_events()` will also turn the pointer on (once).

Under *Linux*, `mouse_pointer()` currently has no effect

It may be necessary to hide the mouse pointer temporarily when you update the screen.

After a call to `text_rows()` you may have to call `mouse_pointer(1)` to see the mouse pointer again.

See Also:

`get_mouse`, `mouse_pointer`

2.0.0.413 movavg

```
include std/stats.e
public function movavg(object data_set, object period_delta)
```

Returns the average (mean) of the data points for overlapping periods. This can be either a simple or weighted moving average.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want a moving average.
2. `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:

1. the first returned element is the average of the first five data points [1..5],
2. 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}
```

See also:

[average](#)

2.0.0.414 move_file

```
include std/filesys.e
public function move_file(sequence src, sequence dest, integer overwrite = 0)
```

Move a file to another location.

Parameters:

1. `src` : a sequence, the name of the file or directory to move
2. `dest` : a sequence, the new location for the file
3. `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](#)

2.0.0.415 mprotect

```
include std/unix/mmap.e
public function mprotect(atom addr, integer length, valid_memory_protection_constant protection)
```

2.0.0.416 munlock

```
include std/unix/mmap.e
public function munlock(atom addr, integer length)
```

2.0.0.417 munmap

```
include std/unix/mmap.e
public function munmap(atom addr, integer length)
```

2.0.0.418 my_dir

```
include std/filesys.e
public integer my_dir
```

Deprecated, so therefore not documented.

2.0.0.419 nested_get

```
include std/map.e
public function nested_get(map the_map_p, sequence the_keys_p, object default_value_p = 0)
```

Parameters:

Returns the value that corresponds to the object `the_keys_p` in the nested map `the_map_p`. `the_keys_p` is a sequence of keys. If any key is not in the map, the object `default_value_p` is returned instead.

2.0.0.420 nested_put

```
include std/map.e
public procedure nested_put(map the_map_p, sequence the_keys_p, object the_value_p, integer operation_p, integer trigger_p)
```

Adds or updates an entry on a map.

Parameters:

1. `the_map_p`: the map where an entry is being added or updated
2. `the_keys_p`: a sequence of keys for the nested maps
3. `the_value_p`: an object, the value to add, or to use for updating.
4. `operation_p`: an integer, indicating what is to be done with `value`. Defaults to PUT.
5. `trigger_p`: an integer. Default is 51. See Comments for details.

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

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 )
```

See also: [put](#)

2.0.0.421 new

```
include std/datetime.e
public function new(integer year = 0, integer month = 0, integer day = 0, integer hour = 0, integer minute = 0, integer second = 0)
```

Create a new datetime value.

Parameters:

1. `year` -- the full year.
2. `month` -- the month (1-12).
3. `day` -- the day of the month (1-31).
4. `hour` -- the hour (0-23) (defaults to 0)
5. `minute` -- the minute (0-59) (defaults to 0)
6. `second` -- the second (0-59) (defaults to 0)

Example 1:

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

See Also:

[from_date](#), [from_unix](#), [now](#), [new_time](#)

2.0.0.422 new

```
include std/map.e
public function new(integer initial_size_p = 690)
```

Create a new map data structure

Parameters:

1. `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
```

2.0.0.423 new

```
include std/regex.e
public function new(string pattern, option_spec options = DEFAULT)
```

Return an allocated regular expression

Parameters:

1. `pattern`: a sequence representing a human readable regular expression
2. `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

Parameters:

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.

```
-- 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

-- 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 1:

```
include std/regex.e as re
re:regex number = re:new("[0-9]+")
```

Note:

For simple matches, the built-in Euphoria routine `eu:match` and the library routine `wildcard:is_match` are often times easier to use and a little faster. Regular expressions are faster for complex searching/matching.

See Also:

[error_message](#), [find](#), [find_all](#)

2.0.0.424 new

```
include std/stack.e
public function new(integer typ = FILO)
```

Create a new stack.

Parameters:

1. `stack_type` : an integer, defining the semantics of the stack. The default is `FIFO`.

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`

2.0.0.425 new

```
include std/wildcard.e
public function new(sequence s)
```

Return a text pattern

Parameters:

1. `pattern` : a sequence representing a text string pattern

Returns:

A the same `pattern`.

Comments:

You might wonder why this function even exists. If you use it and `is_match` you can easily change to the routines found in `std/regex.e`. And if using `std/regex.e` and you restrict yourself to only using `regex:new` and

`regex:is_match`, you can change to using `std/wildcards.e` with very little modification to your source code.

Suppose you work for a hotel and you set up a system for looking up guests.

```
-- The user can use regular expressions to find guests at a hotel...
include std/regex.e as uip -- user input patterns 'uip'
puts(1,"Enter a person to find.  You may use regular expressions:")

sequence person_to_find
object pattern
person_to_find = gets(0)
person_to_find_pattern = uip:new(person_to_find[1..$-1])
while sequence(line) do
    line = line[1..$-1]
    if uip:is_match(person_to_find_pattern, line) then
        -- code for telling users the person is there.
    end if
    -- code loads next name into 'line'
end while
close(dbfd)
```

Later the hotel manager tells you that the users would rather use wildcard matching you need only change the include line and the prompt for the pattern.

```
-- This will make things simpler...
include std/wildcard.e as uip -- user input patterns 'uip'.
puts(1,"Enter a person to find.  You may use '*' and '?' wildcards:")
```

See Also: [is_match](#)

2.0.0.426 new_extra

```
include std/map.e
public function new_extra(object the_map_p, integer initial_size_p = 690)
```

Returns either the supplied map or a new map.

Parameters:

1. `the_map_p`: An object, that could be an existing map
2. `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.
```

2.0.0.427 new_from_kvpairs

```
include std/map.e
public function new_from_kvpairs(sequence kv_pairs)
```

Converts a set of Key-Value pairs to a map.

Parameters:

1. `kv_pairs` : A sequence 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"
```

2.0.0.428 new_from_string

```
include std/map.e
public function new_from_string(sequence kv_string)
```

Converts a set of Key-Value pairs contained in a string to a map.

Parameters:

1. `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`

Comment:

This function actually calls `keyvalues()` to convert the string to key-value pairs, which are then used to create the map.

Example 1:

Given that a file called "xyz.config" contains the lines ...

```
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"
```

2.0.0.429 new_time

```
include std/datetime.e  
public function new_time(integer hour, integer minute, atom second)
```

Create a new datetime value with a date of zeros.

Parameters:

1. `hour` : is the hour (0-23)
2. `minute` : is the minute (0-59)
3. `second` : is the second (0-59)

Example 1:

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

See Also:

[from_date](#), [from_unix](#), [now](#), [new](#)

2.0.0.430 next_prime

```
include std/primes.e
public function next_prime(integer n, object fail_signal_p = - 1, atom time_out_p = 1)
```

Return the next prime number on or after the supplied number

Parameters:

1. *n* : an integer, the starting point for the search
2. *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.

Example 1:

```
? next_prime(997)
-- On a very slow computer, you might get -997, but 1003 is expected.
```

See Also:

[calc_primes](#)

2.0.0.431 not_bits

```
<built-in> function not_bits(object a)
```

Perform the logical NOT operation on each bit in an object. A bit in the result will be 1 when the corresponding bit in `x1` is 0, and will be 0 when the corresponding bit in `x1` is 1.

Parameters:

1. `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`.

Example 1:

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFFF08 interpreted as a negative number)
```

See Also:

[and_bits](#), [or_bits](#), [xor_bits](#), [int_to_bits](#)

2.0.0.432 now

```
include std/datetime.e
public function now()
```

Create a new datetime value initialized with the current date and time

Returns:

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

Example 1:

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

See Also:

[from_date](#), [from_unix](#), [new](#), [new_time](#), [now_gmt](#)

2.0.0.433 now_gmt

```
include std/datetime.e
public function now_gmt()
```

Create a new datetime value that falls into the Greenwich Mean Time (GMT) timezone. This function will return a datetime that is GMT, no matter what timezone the system is running under.

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
```

See Also:

[now](#)

2.0.0.434 number

```
include std/locale.e
public function number(object num)
```

Converts a number into a string representing that number.

Parameters:

1. `num` : an atom, the value to write out.

Returns:

A **sequence**, a string that writes out `num`.

Example 1:

```
-- Assuming an en_US locale
? number(1020.5) -- returns "1,020.50"
```

See Also:

[set](#), [money](#)

2.0.0.435 number_array

```
include std/types.e
public type number_array(object x)
```

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
```

2.0.0.436 object

```
<built-in> function object(object x)
```

Returns information about the object type of the supplied argument `x`.

Parameters:

Returns:

1. 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.

Example 1:

```
? object(1) --> OBJ_INTEGER
? object(1.1) --> OBJ_ATOM
? object("1") --> OBJ_SEQUENCE
object x
? object(x) --> OBJ_UNASSIGNED
```

See Also:

[sequence\(\)](#), [integer\(\)](#), [atom\(\)](#)

2.0.0.437 open

```
<built-in> function open(sequence path, sequence mode, integer cleanup = 0)
```

Open a file or device, to get the file number.

Parameters:

1. `path` : a string, the path to the file or device to open.
2. `mode` : a string, the mode being used o open the file.
3. `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's references goes to 0, or if called as a parameter to `delete()`.

Returns:

A small **integer**, -1 on failure, else 0 or more.

Errors:

There is a limit on the number of files that can be simultaneously opened, currently 40. If this limit is reached, the next attempt to `open()` a file will error out.

The length of `path` should not exceed 1,024 characters.

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.

WIN32 and *Unix*: Long filenames are fully supported for reading and writing and creating.

WIN32: Be careful not to use the special device names in a file name, even if you add an extension. e.g. CON.TXT, CON.DAT, CON.JPG etc. all refer to the CON device, **not a file**.

Example 1:

```
integer file_num, file_num95
sequence first_line
constant ERROR = 2
```

Parameters:

```
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
```

2.0.0.438 open_dll

```
include std/dll.e
public function open_dll(sequence file_name)
```

Open a Windows dynamic link library (.dll) file, or a *Unix* shared library (.so) file.

Parameters:

1. `file_name` : a sequence, the name of the shared library to open or a sequence of filename's to try to open.

Returns:

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

Errors:

The length of `file_name` (or any filename contained therein) should not exceed 1,024 characters.

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'll get the same number returned each time.

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

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
```

See Also:

[define_c_func](#), [define_c_proc](#), [define_c_var](#), [c_func](#), [c_proc](#)

2.0.0.439 operation

```
include std/sets.e
public type operation(object s)
```

Returns 1 if the data represents a map from the product of two sets to a third one.

Comments:

An operation from $F \times G$ to H is defined as a sequence of mappings from G to H , plus the cardinals of the sets F , G and H . If the input data is consistent with this description, 1 is returned, else 0.

Example 1:

```
sequence s = {{2, 3}, {3, 1}, {1, 2}, {2, 3}, {3, 1}}, {5,2,3}
-- s represents the addition modulo 3 from {0, 1, 2, 3, 4} x {1, 2} to {0, 1, 2}
? operation(s)  -- prints out 1.
```

Parameters:

2.0.0.440 optimize

```
include std/map.e
public procedure optimize(map the_map_p, integer max_p = 25, atom grow_p = 1.333)
```

Widens a map to increase performance.

Parameters:

1. `the_map_p` : the map being optimized
2. `max_p` : an integer, the maximum desired size of a bucket. Default is 25. This must be 3 or higher.
3. `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](#)

2.0.0.441 option_spec

```
include std/regex.e
public type option_spec(object o)
```

Regular expression option specification type

Although the functions do not use this type (they return an error instead), you can use this to check if your routine is receiving something sane.

2.0.0.442 option_spec_to_string

```
include std/regex.e
public function option_spec_to_string(option_spec o)
```

Converts an option spec to a string.

Parameters:

This can be useful for debugging what options were passed in. Without it you have to convert a number to hex and lookup the constants in the source code.

2.0.0.443 option_switches

```
<built-in> function option_switches()
```

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

Returns:

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

Comments:

All switches are recorded in upper case.

Example 1:

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

See Also:

[Command line switches](#)

2.0.0.444 or_all

```
include std/math.e  
public function or_all(object a)
```

Or's together all atoms in the argument, no matter how deeply nested.

Parameters:

1. `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.

Example 1:

```
a = sum({10, 7, 35})
-- a is 47
```

See Also:

[can_add](#), [sum](#), [product](#), [or_bits](#)

2.0.0.445 or_bits

```
<built-in> function or_bits(object a, object b)
```

Perform the logical OR operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are both 0.

Parameters:

1. a : one of the objects involved
2. b : the second object

Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by logical 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.

Example 1:

```
a = or_bits(#0F0F0000, #12345678)
-- a is #1F3F5678
```

Example 2:

```
a = or_bits(#FF, {#123456, #876543, #2211})
-- a is {#1234FF, #8765FF, #22FF}
```

See Also:

[and_bits](#), [xor_bits](#), [not_bits](#), [int_to_bits](#)

2.0.0.446 pad_head

```
include std/sequence.e
public function pad_head(sequence target, integer size, object ch = ' ')
```

Pad the beginning of a sequence with an object so as to meet a minimum length condition.

Parameters:

1. `target` : the sequence to pad.
2. `size` : an integer, the target minimum size for `target`
3. `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.

Example 1:

```
s = pad_head("ABC", 6)
-- s is "   ABC"

s = pad_head("ABC", 6, '-')
-- s is "---ABC"
```

See Also:

[trim_head](#), [pad_tail](#), [head](#)

2.0.0.447 pad_tail

```
include std/sequence.e
public function pad_tail(sequence target, integer size, object ch = ' ')
```

Pad the end of a sequence with an object so as to meet a minimum length condition.

Parameters:

1. `target` : the sequence to pad.
2. `size` : an integer, the target minimum size for `target`
3. `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.

Comments:

`pad_tail()` will not remove characters. If `length(str)` is greater than `params`, this function simply returns `str`. see [tail\(\)](#) if you wish to truncate long sequences.

Example 1:

```
s = pad_tail("ABC", 6)
-- s is "ABC  "

s = pad_tail("ABC", 6, '-')
-- s is "ABC---"
```

See Also:

[trim_tail](#), [pad_head](#), [tail](#)

2.0.0.448 page_aligned_address

```
include std/machine.e
public type page_aligned_address(atom a)
```

page aligned address type

2.0.0.449 pairs

```
include std/map.e
public function pairs(map the_map_p, integer sorted_result = 0)
```

Return all key/value pairs in a map.

Parameters:

1. `the_map_p` : the map to get the data from
2. `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.

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"}}
```

See Also:

[get](#), [keys](#), [values](#)

2.0.0.450 parse

```
include std/datetime.e
public function parse(sequence val, sequence fmt = "%Y-%m-%d %H:%M:%S")
```

Parse a datetime string according to the given format.

Parameters:

1. `val` : string datetime value
2. `fmt` : datetime format. Default is "%Y-%m-%d %H:%M:%S"

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 -- year

More format codes will be added in future versions.

Parameters:

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.

Example 1:

```
datetime d = parse("05/01/2009 10:20:30", "%m/%d/%Y %H:%M:%S")
```

See Also:

[format](#)

2.0.0.451 parse

```
include std/net/url.e  
public function parse(sequence url, integer querystring_also = 0)
```

Parse a URL returning its various elements.

Parameters:

1. url: URL to parse
2. querystring_also: Parse the query string into a map also?

Returns:

A multi-element sequence containing:

1. protocol
2. host name
3. port
4. path
5. user name
6. password
7. query string

Or, zero if the URL could not be parsed.

Notes:

If the host name, port, path, username, password or query string are not part of the URL they will be returned as an integer value of zero.

Example 1:

```
sequence parsed = parse("http://user:pass@www.debian.org:80/index.html?name=John&age=39")
-- parsed is
-- {
--     "http",
--     "www.debian.org",
--     80,
--     "/index.html",
--     "user",
--     "pass",
--     "name=John&age=39"
-- }
```

2.0.0.452 parse_commandline

```
include std/cmdline.e
public function parse_commandline(sequence cmdline)
```

Parse a command line string breaking it into a sequence of command line options.

Parameters:

1. `cmdline` : Command line sequence (string)

Returns:

A **sequence**, of command line options

Example 1:

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

See Also:

[build_commandline](#)

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](#)

2.0.0.453 parse_ip_address

```
include std/net/common.e
public function parse_ip_address(sequence address, integer port = - 1)
```

Converts a text "address:port" into {"address", port} format.

Parameters:

1. `address` : ip address to connect, optionally with `:PORT` at the end
2. `port` : optional, if not specified you may include `:PORT` in the address parameter otherwise the default port 80 is used.

Comments:

If `port` is supplied, it overrides any `:PORT` value in the input address.

Returns:

A **sequence**, of two elements: "address" and integer port number.

Example 1:

```
addr = parse_ip_address("11.1.1.1") --> {"11.1.1.1", 80} -- default port
addr = parse_ip_address("11.1.1.1:110") --> {"11.1.1.1", 110}
addr = parse_ip_address("11.1.1.1", 345) --> {"11.1.1.1", 345}
```

2.0.0.454 parse_querystring

```
include std/net/url.e
public function parse_querystring(object query_string)
```

Parse a query string into a map

Parameters:

1. `query_string`: Query string to parse

Returns:

`map` containing the key/value pairs

Example 1:

```
map qs = parse_querystring("name=John&age=18")
printf(1, "%s is %s years old\n", { map:get(qs, "name"), map:get(qs, "age") })
```

2.0.0.455 parse_recvheader

```
include std/net/http.e
public procedure parse_recvheader(sequence header)
```

Populates the internal sequence `recvheader` from the flat string header.

Parameters:

1. `header` : a string, the header data

Comments:

This must be called prior to calling `get_recvheader()`.

2.0.0.456 parse_url

```
include std/net/common.e
public function parse_url(sequence url)
```

Parse a common URL. Currently supported URLs are `http(s)`, `ftp(s)`, `gopher(s)` and `mailto`.

Parameters:

1. `url` : url to be parsed

Returns:

A **sequence**, containing the URL details. You should use the `URL_` constants to access the values of the returned sequence. You should first check the protocol (`URL_PROTOCOL`) that was returned as the data contained in the return value can be of different lengths.

On a parse error, -1 will be returned.

Example 1:

```
object url_data = parse_url("http://john.com/index.html?name=jeff")
-- url_data = {
--   "http://john.com/index.html?name=jeff", -- URL_ENTIRE
--   "http", -- URL_PROTOCOL
--   "john.com", -- URL_DOMAIN
--   "/index.html", -- URL_PATH
--   "?name=jeff" -- URL_QUERY
-- }

url_data = parse_url("mailto:john@mail.doe.com?subject=Hello%20John%20Doe")
-- url_data = {
--   "mailto:john@mail.doe.com?subject=Hello%20John%20Doe",
--   "mailto",
--   "john@mail.doe.com",
--   "john",
--   "mail.doe.com",
--   "?subject=Hello%20John%20Doe"
-- }
```

Based on EuNet project, version 1.3.2 at SourceForge.

2.0.0.457 patch

```
include std/sequence.e
public function patch(sequence target, sequence source, integer start, object filler = '')
```

Changes a sequence slice, possibly with padding

Parameters:

1. `target` : a sequence, a modified copy of which will be returned
2. `source` : a sequence, to be patched inside or outside `target`
3. `start` : an integer, the position at which to patch
4. `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.

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"
```

See Also:

[mid](#), [replace](#)

2.0.0.458 pathinfo

```
include std/filesys.e
public function pathinfo(sequence path, integer std_slash = 0)
```

Parse a fully qualified pathname.

Parameters:

Parameters:

1. `path` : a sequence, the path to parse

Returns:

A **sequence**, of length 5. Each of these elements is a string:

- The path name
- 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.

Example 1:

```
-- WIN32
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", "", ""}
```

See Also:

[driveid](#), [dirname](#), [filename](#), [fileext](#), [PATH_BASENAME](#), [PATH_DIR](#), [PATH_DRIVEID](#), [PATH_FILEEXT](#), [PATH_FILENAME](#)

2.0.0.459 pathname

```
include std/filesys.e
public function pathname(sequence path)
```

Return the directory name of a fully qualified filename

Parameters:

1. `path` : the path from which to extract information
2. `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.

Example 1:

```
fname = dirname("/opt/euphoria/docs/readme.txt")
-- fname is "/opt/euphoria/docs"
```

See Also:

[driveid](#), [filename](#), [pathinfo](#)

2.0.0.460 pcre_copyright

```
include info.e
public function pcre_copyright()
```

Get the copyright statement for PCRE.

Returns:

A **sequence**, containing 2 sequences: product name and copyright message.

See Also:

[euphoria_copyright\(\)](#)

2.0.0.461 peek

```
<built-in> function peek(object addr_n_length)
```

Fetches a byte, or some bytes, from an address in memory.

Parameters:

1. `addr_n_length` : an object, either of
 - ◆ an atom `addr` -- to fetch one byte at `addr`, or
 - ◆ a pair `{addr, len}` -- to fetch `len` bytes at `addr`

Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are bytes, in the range 0..255.

Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a EUPHORIA error.

When supplying a `{address, count}` sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several bytes at once using the second form of `peek()` than it is to read one byte at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek()` takes just one argument, which in the second form is actually a 2-element sequence.

Example 1:

```
-- The following are equivalent:  
-- method 1  
s = {peek(100), peek(101), peek(102), peek(103)}
```

Parameters:

```
-- method 2  
s = peek({100, 4})
```

See Also:

[poke](#), [peek](#), [peek4u](#), [allocate](#), [free](#), [peek2u](#)

2.0.0.462 peek

```
include std/safe.e  
override function peek(object
```

2.0.0.463 peek2s

```
<built-in> function peek2s(object addr_n_length)
```

Fetches a *signed* word, or some *signed* words , from an address in memory.

Parameters:

1. `addr_n_length` : an object, either of
 - ◆ an atom `addr` -- to fetch one word at `addr`, or
 - ◆ a pair { `addr`, `len`}, to fetch `len` words at `addr`

Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are double words, in the range -32768..32767.

Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a EUPHORIA error.

When supplying a {address, count} sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several words at once using the second form of `peek()` than it is to read one word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek2s()` takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between `peek2s()` and `peek2u()` is how words with the highest bit set are returned. `peek2s()` assumes them to be negative, while `peek2u()` just assumes them to be large and positive.

Example 1:

```
-- The following are equivalent:
-- method 1
s = {peek2s(100), peek2s(102), peek2s(104), peek2s(106)}

-- method 2
s = peek2s({100, 4})
```

See Also:

[poke2](#), [peeks](#), [peek4s](#), [allocate](#), [free peek2u](#)

2.0.0.464 peek2s

```
include std/safe.e
override function peek2s(object
```

2.0.0.465 peek2u

```
<built-in> function peek2u(object addr_n_length)
```

Fetches an *unsigned* word, or some *unsigned* words, from an address in memory.

Parameters:

1. `addr_n_length`: an object, either of
 - ◆ an atom `addr` -- to fetch one double word at `addr`, or
 - ◆ a pair `{addr, len}` -- to fetch `len` double words at `addr`

Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are words, in the range 0..65535.

Errors:

Peek() in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a EUPHORIA error.

When supplying a {address, count} sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several words at once using the second form of peek() than it is to read one word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that peek2u() takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between peek2s() and peek2u() is how words with the highest bit set are returned. peek2s() assumes them to be negative, while peek2u() just assumes them to be large and positive.

Example 1:

```
-- The following are equivalent:
-- method 1
Get 4 2-byte numbers starting address 100.
s = {peek2u(100), peek2u(102), peek2u(104), peek2u(106)}

-- method 2
Get 4 2-byte numbers starting address 100.
s = peek2u({100, 4})
```

See Also:

[poke2](#), [peek](#), [peek2s](#), [allocate](#), [free peek4u](#)

2.0.0.466 peek2u

```
include std/safe.e
override function peek2u(object
```

Parameters:

2.0.0.467 peek4s

```
<built-in> function peek4s(object addr_n_length)
```

Fetches a *signed* double words, or some *signed* double words, from an address in memory.

Parameters:

1. `addr_n_length` : an object, either of
 - ◆ an atom `addr` -- to fetch one double word at `addr`, or
 - ◆ a pair { `addr`, `len` } -- to fetch `len` double words at `addr`

Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are double words, in the range $-\text{power}(2,31)..\text{power}(2,31)-1$.

Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a EUPHORIA error.

When supplying a {address, count} sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several double words at once using the second form of `peek()` than it is to read one double word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek4s()` takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between `peek4s()` and `peek4u()` is how double words with the highest bit set are returned. `peek4s()` assumes them to be negative, while `peek4u()` just assumes them to be large and positive.

Example 1:

```
-- The following are equivalent:
-- method 1
s = {peek4s(100), peek4s(104), peek4s(108), peek4s(112)}

-- method 2
s = peek4s({100, 4})
```

See Also:

[poke4](#), [peeks](#), [peek4u](#), [allocate](#), [free](#), [peek2s](#)

2.0.0.468 peek4s

```
include std/safe.e
override function peek4s(object
```

2.0.0.469 peek4u

```
<built-in> function peek4u(object addr_n_length)
```

Fetches an *unsigned* double word, or some *unsigned* double words, from an address in memory.

Parameters:

1. `addr_n_length`: an object, either of
 - ◆ an atom `addr` -- to fetch one double word at `addr`, or
 - ◆ a pair `{addr, len}` -- to fetch `len` double words at `addr`

Returns:

An **object**, either an atom if the input was a single address, or a sequence of atoms if a sequence was passed. In both cases, atoms returned are double words, in the range $0..power(2,32)-1$.

Errors:

Peek() in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a EUPHORIA error.

When supplying a `{address, count}` sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several double words at once using the second form of `peek()` than it is to read one double word at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peek4u()` takes just one argument, which in the second form is actually a 2-element sequence.

The only difference between `peek4s()` and `peek4u()` is how double words with the highest bit set are returned. `peek4s()` assumes them to be negative, while `peek4u()` just assumes them to be large and positive.

Example 1:

```
-- The following are equivalent:
-- method 1
s = {peek4u(100), peek4u(104), peek4u(108), peek4u(112)}

-- method 2
s = peek4u({100, 4})
```

See Also:

[poke4](#), [peek](#), [peek4s](#), [allocate](#), [free](#), [peek2u](#)

2.0.0.470 peek4u

```
include std/safe.e
override function peek4u(object
```

2.0.0.471 peek_end

```
include std/stack.e
public function peek_end(stack sk, integer idx = 1)
```

Gets an object, relative to the end, from a stack.

Parameters:

1. `sk` : the stack to get from.
2. `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.

Errors:

- If the stack is empty, an error occurs.
- If the `idx` is greater than the number of items in the stack, an error occurs.

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 ...

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)
```

See Also:

[pop](#), [top](#), [is_empty](#), [size](#), [peek_top](#)

2.0.0.472 peek_string

```
<built-in> procedure peek_string(atom addr)
```

Read an ASCII string in RAM, starting from a supplied address.

Parameters:

1. `addr` : an atom, the address at which to start reading.

Returns:

A **sequence**, of bytes, the string that could be read.

Errors:

Further, `peek()` memory that doesn't belong to your process is something the operating system could prevent, and you'd crash with a machine level exception.

Comments:

An ASCII string is any sequence of bytes and ends with a 0 byte. If you `peek_string()` at some place where there is no string, you will get a sequence of garbage.

See Also:

[peek](#), [peek_wstring](#), [allocate_string](#)

2.0.0.473 peek_string

```
include std/safe.e
override function peek_string(object
```

2.0.0.474 peek_top

```
include std/stack.e
public function peek_top(stack sk, integer idx = 1)
```

Gets an object, relative to the top, from a stack.

Parameters:

1. `sk` : the stack to get from.
2. `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.

Errors:

- If the stack is empty, an error occurs.
- If the `idx` is greater than the number of items in the stack, an error occurs.

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 ...

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)
```

Parameters:

See Also:

[pop](#), [top](#), [is_empty](#), [size](#), [peek_end](#)

2.0.0.475 peek_wstring

```
include std/machine.e
public function peek_wstring(atom addr)
```

Return a unicode (utf16) string that are stored at machine address a.

Parameters:

1. `addr` : an atom, the address of the string in memory

Returns:

The **string**, at the memory position. The terminator is the null word (two bytes equal to 0).

See Also:

[peek_string](#)

2.0.0.476 peeks

```
<built-in> function peeks(object addr_n_length)
```

Fetches a byte, or some bytes, from an address in memory.

Parameters:

1. `addr_n_length` : an object, either of
 - ◆ an atom `addr` : to fetch one byte at `addr`, or
 - ◆ a pair `{addr, len}` : to fetch `len` bytes at `addr`

Returns:

An **object**, either an integer if the input was a single address, or a sequence of integers if a sequence was passed. In both cases, integers returned are bytes, in the range -128..127.

Errors:

Peeking in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the define safe these routines will catch these problems with a EUPHORIA error.

When supplying a {address, count} sequence, the count must not be negative.

Comments:

Since addresses are 32-bit numbers, they can be larger than the largest value of type integer (31-bits). Variables that hold an address should therefore be declared as atoms.

It is faster to read several bytes at once using the second form of `peek()` than it is to read one byte at a time in a loop. The returned sequence has the length you asked for on input.

Remember that `peeks()` takes just one argument, which in the second form is actually a 2-element sequence.

Example 1:

```
-- The following are equivalent:
-- method 1
s = {peeks(100), peek(101), peek(102), peek(103)}

-- method 2
s = peeks({100, 4})
```

See Also:

[poke](#), [peek4s](#), [allocate](#), [free](#), [peek2s](#), [peek](#)

2.0.0.477 peeks

```
include std/safe.e
override function peeks(object
```

2.0.0.478 pivot

```
include std/sequence.e
public function pivot(object data_p, object pivot_p = 0)
```

Returns a sequence of three sub-sequences. The sub-sequences contain all the elements less than the supplied pivot value, equal to the pivot, and greater than the pivot.

Parameters:

Parameters:

1. `data_p` : Either an atom or a list. An atom is treated as if it is one-element sequence.
2. `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 quicksort(sequence s)
  length(s) < 2 then
    s
  end if
  sequence
    k(s, pivot and length(s))

    quicksort(k[1]) & k[2] & quicksort(k[3])
end function
sequence t2 = {5, 4, 7, 2, 4, 9, 1, 0, 4, 32, 7, 54, 2, 5, 8, 445, 67}
? quicksort(t2) --> {0, 1, 2, 2, 4, 4, 4, 5, 5, 7, 7, 8, 9, 32, 54, 67, 445}
```

2.0.0.479 platform

```
<built-in> function platform()
```

Indicates the platform that the program is being executed on.

Returns:

An **integer**,

```
public constant
    WIN32,
    LINUX,
    FREEBSD,
    OSX,
    SUNOS,
    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.

Example 1:

```
ifdef WIN32 then
    -- call system Beep routine
    err = c_func(Beep, {0,0})
elseif
    -- do nothing (Linux/FreeBSD)
end if
```

See Also:

[Platform-Specific Issues, ifdef statement](#)

2.0.0.480 platform_locale

```
include std/localeconv.e
public constant platform_locale
```

2.0.0.481 platform_name

```
include info.e
public function platform_name()
```

Get the platform name

Returns:

A **sequence**, containing the platform name, i.e. Windows, Linux, DOS, FreeBSD or OS X.

2.0.0.482 poke

```
<built-in> procedure poke(atom addr, object x)
```

Stores one or more bytes, starting at a memory location.

Parameters:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a byte or a non empty sequence of bytes.

Errors:

`Poke()` in memory you don't own may be blocked by the OS, and cause a machine exception. The `-D SAFE` option will make `poke()` catch this sort of issues.

Comments:

The lower 8-bits of each byte value, i.e. `remainder(x, 256)`, is actually stored in memory.

It is faster to write several bytes at once by poking a sequence of values, than it is to write one byte at a time in a loop.

Writing to the screen memory with `poke()` can be much faster than using `puts()` or `printf()`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, `ed`, never uses `poke()`.

Example 1:

```
a = allocate(100)  -- allocate 100 bytes in memory
-- poke one byte at a time:
poke(a, 97)
```

Parameters:

```
poke(a+1, 98)
poke(a+2, 99)

-- poke 3 bytes at once:
poke(a, {97, 98, 99})
```

Example 2:

demo/callmach.ex

See Also:

[peek](#), [peeks](#), [poke4](#), [allocate](#), [free](#), [poke2](#), [call](#), [mem_copy](#), [mem_set](#)

2.0.0.483 poke

```
include std/safe.e
override procedure poke(atom
```

2.0.0.484 poke2

```
<built-in> procedure poke2(atom addr, object x)
```

Stores one or more words, starting at a memory location.

Parameters:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a word or a non empty sequence of words.

Errors:

Poke() in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a EUPHORIA error.

Comments:

There is no point in having `poke2s()` or `poke2u()`. For example, both 32768 and -32768 are stored as #F000 when stored as words. It's up to whoever reads the value to figure it out.

It is faster to write several words at once by poking a sequence of values, than it is to write one words at a time in a loop.

Parameters:

Writing to the screen memory with `poke2()` can be much faster than using `puts()` or `printf()`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, `ed`, never uses `poke2()`.

The 2-byte values to be stored can be negative or positive. You can read them back with either `peek2s()` or `peek2u()`. Actually, only `remainder(x,65536)` is being stored.

Example 1:

```
a = allocate(100)    -- allocate 100 bytes in memory

-- poke one 2-byte value at a time:
poke2(a, 12345)
poke2(a+2, #FF00)
poke2(a+4, -12345)

-- poke 3 2-byte values at once:
poke4(a, {12345, #FF00, -12345})
```

See Also:

[peek2s](#), [peek2u](#), [poke](#), [poke4](#), [allocate](#), [free](#), [call](#)

2.0.0.485 `poke2`

```
include std/safe.e
override procedure poke2(atom
```

2.0.0.486 `poke4`

```
<built-in> procedure poke4(atom addr, object x)
```

Stores one or more double words, starting at a memory location.

Parameters:

1. `addr` : an atom, the address at which to store
2. `x` : an object, either a double word or a non empty sequence of double words.

Errors:

`Poke()` in memory you don't own may be blocked by the OS, and cause a machine exception. If you use the `define safe` these routines will catch these problems with a `EUPHORIA` error.

Comments:

There is no point in having `poke4s()` or `poke4u()`. For example, both `+power(2,31)` and `-power(2,31)` are stored as `#F0000000`. It's up to whoever reads the value to figure it out.

It is faster to write several double words at once by poking a sequence of values, than it is to write one double words at a time in a loop.

Writing to the screen memory with `poke4()` can be much faster than using `puts()` or `printf()`, but the programming is more difficult. In most cases the speed is not needed. For example, the Euphoria editor, `ed`, never uses `poke4()`.

The 4-byte values to be stored can be negative or positive. You can read them back with either `peek4s()` or `peek4u()`. However, the results are unpredictable if you want to store values with a fractional part or a magnitude greater than `power(2,32)`, even though Euphoria represents them all as atoms.

Example 1:

```
a = allocate(100)    -- allocate 100 bytes in memory

-- poke one 4-byte value at a time:
poke4(a, 9712345)
poke4(a+4, #FF00FF00)
poke4(a+8, -12345)

-- poke 3 4-byte values at once:
poke4(a, {9712345, #FF00FF00, -12345})
```

See Also:

[peek4s](#), [peek4u](#), [poke](#), [poke2](#), [allocate](#), [free](#), [call](#)

2.0.0.487 poke4

```
include std/safe.e
override procedure poke4(atom
```

2.0.0.488 poke_string

```
include std/machine.e
public function poke_string(atom buffaddr, integer buffsize, sequence s)
```

Stores a C-style null-terminated ANSI string in memory

Parameters:

1. `buffaddr`: an atom, the RAM address to to the string at.
2. `buffsize`: an integer, the number of bytes available, starting from `buffaddr`.
3. `s`: a sequence, the string to store at address `buffaddr`.

Comments:

- This does not allocate an RAM. You must supply the preallocated area.
- This can only be used on ANSI strings. It cannot be used for double-byte strings.
- If `s` is not a string, nothing is stored and a zero is returned.

Returns:

An atom. If this is zero, then nothing was stored, otherwise it is the address of the first byte after the stored string.

Example 1:

```
atom title

title = allocate(1000)
if poke_string(title, 1000, "The Wizard of Oz") then
    -- successful
else
    -- failed
end if
```

See Also:

[allocate](#), [allocate_string](#)

2.0.0.489 poke_wstring

```
include std/machine.e
public function poke_wstring(atom buffaddr, integer buffsize, sequence s)
```

Stores a C-style null-terminated Double-Byte string in memory

Parameters:

1. `buffaddr`: an atom, the RAM address to to the string at.
2. `buffsize`: an integer, the number of bytes available, starting from `buffaddr`.
3. `s`: a sequence, the string to store at address `buffaddr`.

Comments:

- This does not allocate an RAM. You must supply the preallocated area.
- This uses two bytes per string character. **Note** that `buffsize` is the number of *bytes* available in the buffer and not the number of *characters* available.
- If `s` is not a double-byte string, nothing is stored and a zero is returned.

Returns:

An atom. If this is zero, then nothing was stored, otherwise it is the address of the first byte after the stored string.

Example 1:

```
atom title

title = allocate(1000)
if poke_wstring(title, 1000, "The Wizard of Oz") then
    -- successful
else
    -- failed
end if
```

See Also:

[allocate](#), [allocate_wstring](#)

2.0.0.490 pop

```
include std/stack.e
public function pop(stack sk, integer idx = 1)
```

Removes an object from a stack.

Parameters:

1. `sk` : the stack to pop
2. `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.

Errors:

- If the stack is empty, an error occurs.
- If the `idx` is greater than the number of items in the stack, an error occurs.

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 ...

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)
```

Parameters:

```
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}
```

See Also:

[push](#), [top](#), [is_empty](#)

2.0.0.491 position

```
<built-in> procedure position(integer row, integer column)
```

Parameters:

1. `row`: an integer, the index of the row to position the cursor on.
2. `column`: an integer, the index of the column to position the cursor on.

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.

Example 1:

```
position(2,1)
-- the cursor moves to the beginning of the second line from the top
```



See Also:

[get_position](#)

2.0.0.492 positive_int

```
include std/console.e
public type positive_int(integer x)
```

2.0.0.493 positive_int

```
include std/memory.e
export type positive_int(integer x)
```

Positive integer type

2.0.0.494 positive_int

```
include std/safe.e
export type positive_int(integer x)
```

2.0.0.495 posix_names

```
include std/localeconv.e
public constant posix_names
```

POSIX 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

Parameters:

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

2.0.0.496 power

<built-in> `function power(object base, object exponent)`

Raise a base value to some power.

Parameters:

1. `base` : an object, the value(s) to raise to some power.
2. `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`.

Errors:

If some atom in `base` is negative and is raised to a non integer exponent, an error will occur, as the result is undefined.

If 0 is raised to any negative power, this is the same as a zero divide and causes an error.

`power(0, 0)` is illegal, because there is not an unique value that can be assigned to that quantity.

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.

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}
```

See Also:

[log](#), [Operations on sequences](#)

2.0.0.497 powof2

```
include std/math.e
public function powof2(object p)
```

Tests for power of 2

Parameters:

1. `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}
```

2.0.0.498 prepare_block

```
include std/memory.e
export function prepare_block(atom addr, integer a, integer protection)
```

2.0.0.499 prepare_block

```
include std/safe.e
export function prepare_block(int_addr iaddr, positive_int n, natural protection)
```

2.0.0.500 prepend

```
<built-in> function prepend(sequence target, object x)
```

Adds an object as the first element of a sequence.

Parameters:

1. `source` : the sequence to add to
2. `x` : the object to add

Parameters:

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` itself is `prepend()`ed to is handled very efficiently.

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}
```

See Also:

[append](#), [&](#)

2.0.0.501 pretty_print

```
include std/pretty.e
public procedure pretty_print(integer fn, object x, sequence options = PRETTY_DEFAULT)
```

Print an object to a file or device, using braces { , , }, indentation, and multiple lines to show the structure.

Parameters:

1. `fn` : an integer, the file/device number to write to
2. `x` : the object to display/convert to printable form
3. `options` : is an (up to) 10-element options sequence.

Comments:

Pass {} in `options` to select the defaults, or set options as below:

1. 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`
2. amount to indent for each level of sequence nesting -- default: 2
3. column we are starting at -- default: 1
4. approximate column to wrap at -- default: 78
5. format to use for integers -- default: "%d"
6. format to use for floating-point numbers -- default: "%.10g"
7. minimum value for printable ASCII -- default 32
8. maximum value for printable ASCII -- default 127
9. maximum number of lines to output
10. 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:

1. `DISPLAY_ASCII`
2. `INDENT`
3. `START_COLUMN`
4. `WRAP`
5. `INT_FORMAT`
6. `FP_FORMAT`
7. `MIN_ASCII`
8. `MAX_ASCII`
9. `MAX_LINES`
10. `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"`

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  
  }  
}
```

See Also:

[print](#), [sprint](#), [printf](#), [sprintf](#), [pretty_sprint](#)

2.0.0.502 pretty_sprint

```
include std/pretty.e
public function pretty_sprint(object x, sequence options = PRETTY_DEFAULT)
```

Format an object using braces { , , }, indentation, and multiple lines to show the structure.

Parameters:

1. `x` : the object to display
2. `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](#)

Page Contents

2.0.0.503 prime_list

```
include std/primes.e
public function prime_list(integer top_prime_p = 0)
```

Returns a list of prime numbers.

Parameters:

Parameters:

1. `top_prime_p` : The list will end with the prime less than or equal to this value. If this is zero, the current list calculated primes is returned.

Returns:

An **sequence**, a list of prime numbers from 2 to `top_prime_p`

Example 1:

```
sequence pList = prime_list(1000)
-- pList will now contain all the primes from 2 up to the largest less than or
-- equal to 1000.
```

See Also:

[calc_primes](#), [next_prime](#)

2.0.0.504 print

```
<built-in> procedure print(integer fn, object x)
```

Writes out a **text** representation of an object to a file or device. If the object `x` is a sequence, it uses braces `{ , , , }` to show the structure.

Parameters:

1. `fn` : an integer, the handle to a file or device to output to
2. `x` : the object to print

Errors:

The target file or device must be open.

Comments:

This is not used to write to "binary" files as it only outputs text.

Example 1:

```
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:

```
print(STDOUT, repeat({10,20}, 3)) -- output is: {{10,20},{10,20},{10,20}}
```

See Also:

?, puts

2.0.0.505 printf

```
<built-in> procedure printf(integer fn, sequence format, object values)
```

Print one or more values to a file or device, using a format string to embed them in and define how they should be represented.

Parameters:

1. `fn` : an integer, the handle to a file or device to output to
2. `format` : a sequence, the text to print. This text may contain format specifiers.
3. `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.

Errors:

If there are less values to show than format specifiers, a run time error will occur.

The target file or device must be open.

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 the middle.

`format` will be scanned for format specifiers. Whenever one is found, the current value in `values` will be turned into a string according to the format specifier. The resulting string will be plugged in the result, as if

replacing the modifier with the printed value. Then moving on to next value and carrying the process on.

This way, `printf()` always takes exactly 3 arguments, no matter how many values are to be printed. Only the length of the last argument, containing the values to be printed, will vary.

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 the `'%'` character itself. This is not an actual format specifier.

Field widths can be added to the basic formats, e.g. `%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.

If the field width is negative, e.g. `%-5d` then the value will be left-justified within the field. Normally it will be right-justified. If the field width starts with a leading 0, e.g. `%08d` then leading zeros will be supplied to fill up the field. If the field width starts with a '+' e.g. `%+7d` then a plus sign will be printed for positive values.

Comments:

Watch out for the following common mistake:

```
name="John Smith"
printf(STDOUT, "%s", name)    -- error!
```

This will print only the first character, J, of name, as each element of name is taken to be a separate value to be formatted. You must say this instead:

```
name="John Smith"
printf(STDOUT, "%s", {name})  -- correct
```

Now, the third argument of `printf()` is a one-element sequence containing the item to be formatted.

If there is only one `%` format specifier, and if the value it stands for is an atom, then `values` may be simply that atom.

Example 1:

```
rate = 7.875
printf(my_file, "The interest rate is: %8.2f\n", rate)
```

```
--      The interest rate is:      7.88
```

Example 2:

```
name="John Smith"
score=97
printf(STDOUT, "%15s, %5d\n", {name, score})
```

```
--      John Smith,      97
```

Example 3:

```
printf(STDOUT, "%-10.4s $ %s", {"ABCDEFGHJKLMNOP", "XXX"})
--      ABCD      $ XXX
```

Example 4:

```
printf(STDOUT, "%d %e %f %g", repeat(7.75, 4)) -- same value in different formats
--      7 7.750000e+000 7.750000 7.75
```

See Also:

[sprintf](#), [sprint](#), [print](#)

2.0.0.506 process

```
include std/pipeio.e
public type process(object o)
```

Process Type

2.0.0.507 process_lines

```
include std/io.e
public function process_lines(object file, integer proc, object user_data = 0)
```

Process the contents of a file, one line at a time.

Parameters:

1. `file` : an object. Either a file path or the handle to an open file. An empty string signifies STDIN - the console keyboard.
2. `proc` : an integer. The `routine_id` of a function that will process the line.
3. `user_data` : an 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.

Example:

```
-- Format each supplied line according to the format pattern supplied as well.
function show(sequence aLine, integer line_no, object data)
  writeln( 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})
```

See Also:

[gets](#), [read_lines](#), [read_file](#)

2.0.0.508 product

```
include std/math.e
public function product(object a)
```

Compute the product of all the atom in the argument, no matter how deeply nested.

Parameters:

Parameters:

1. `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

Example 1:

```
a = product({10, 20, 30})
-- a is 6000

a = product({10.5, {11.2} , 8.1})
-- a is 952.56
```

See Also:

[can_add](#), [sum](#), [or_all](#)

2.0.0.509 product

```
include std/sets.e
public function product(set S1, set S2)
```

Returns the set of all pairs made of an element of a set and an element of another set.

Parameters:

1. `S1` : The set where the first coordinate lives
2. `S2` : The set where the second coordinate lives

Returns:

The **set**, of all pairs made of an element of `S1` and an element of `S2`.

Example 1:

```
set s0,s1,s2
s1 = {1, 3, 5, 7} s2 = {-1, 3}
s0 = product(s1, s2)  -- s0 is now {{1, -1}, {1, 3}, {3, -1}, {3, 3}, {5, -1}, {5, 3}, {7, -1}}
```

See Also:

[product_map](#), [amalgamated_sum](#), [fiber_product](#)

2.0.0.510 product_map

```
include std/sets.e
public function product_map(map f1, map f2)
```

Builds a map to a product from a map to each of its components.

Parameters:

1. f1 : the map going to the first component
2. f2 : the map going to the second component

Returns:

A **map**, $f=f1 \times f2$ defined by $f(x, y) = \{f1(x), f2(y)\}$ wherever this makes sense.

Example 1:

```
set s = {1,3,5,7}
map f = {3,1,4,1,4,4}
map f1 = product(f,f)
-- f1 is {11,9,12,9,3,1,4,1,15,13,16,13,3,1,4,1,16,16}.
```

See Also:

[product](#), [amalgamated_sum](#), [fiber_product](#)

2.0.0.511 project

```
include std/sequence.e
public function project(sequence source, sequence coords)
```

Parameters:

Creates a list of sequences based on selected elements from sequences in the source.

Parameters:

1. `source` : a list of sequences.
2. `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.

Example 1:

```
s = project({ "ABCD", "789"}, {{1,2}, {3,1}, {2}})
-- s is {"AB","CA","B"}, {"78","97","8"}
```

See Also:

[vslice](#), [extract](#)

2.0.0.512 `prompt_number`

```
include std/console.e
public function prompt_number(sequence prompt, sequence range)
```

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

Parameters:

1. `st` : is a string of text that will be displayed on the screen.
2. `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.

Errors:

If **puts()** cannot display `st` on standard output, or if the first or second element of `s` is a sequence, a runtime error will be raised.

If user tries cancelling the prompt by hitting Ctrl-Z, the program will abort as well, issuing a type check error.

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.

Example 1:

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

Example 2:

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

See Also:

[puts](#), [prompt_string](#)

2.0.0.513 prompt_string

```
include std/console.e
public function prompt_string(sequence prompt)
```

Prompt the user to enter a string of text.

Parameters:

1. `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.

Example 1:

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

See Also:

[prompt_number](#)

2.0.0.514 proper

```
include std/text.e
public function proper(sequence x)
```

Convert a text sequence to capitalized words.

Parameters:

1. *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.

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.

```

See Also:

[lower upper](#)

2.0.0.515 push

```

include std/stack.e
public procedure push(stack sk, object value)

```

Adds something to a stack.

Parameters:

1. `sk` : the stack to augment
2. `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.

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

```

See Also:

[pop](#), [top](#)

2.0.0.516 put

```

include std/map.e
public procedure put(map the_map_p, object the_key_p, object the_value_p, integer operation_p =

```

Adds or updates an entry on a map.

Parameters:

1. `the_map_p` : the map where an entry is being added or updated
2. `the_key_p` : an object, the `the_key_p` to look up
3. `the_value_p` : an object, the value to add, or to use for updating.
4. `operation` : an integer, indicating what is to be done with `the_value_p`. Defaults to PUT.
5. `trigger_p` : an integer. Default is 100. 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.

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
```

See Also:

[remove](#), [has](#), [nested_put](#)

2.0.0.517 put_integer16

```
include std/io.e
public procedure put_integer16(integer fh, atom val)
```

Write the supplied integer as two bytes to a file.

Parameters:

1. *fh* : an integer, the handle to an open file to write to.
2. *val* : an integer

Comments:

- This function is normally used with files opened in binary mode, "wb".

Example 1:

```
integer fn
fn = open("temp", "wb")

put_integer16(fn, 1234)
```

Parameters:

**See Also:**

[getc](#), [gets](#), [get_bytes](#), [get_dstring](#)

2.0.0.518 put_integer32

```
include std/io.e
public procedure put_integer32(integer fh, atom val)
```

Write the supplied integer as four bytes to a file.

Parameters:

1. `fh` : an integer, the handle to an open file to write to.
2. `val` : an integer

Comments:

- This function is normally used with files opened in binary mode, "wb".

Example 1:

```
integer fn
fn = open("temp", "wb")

put_integer32(fn, 1234)
```

See Also:

[getc](#), [gets](#), [get_bytes](#), [get_dstring](#)

2.0.0.519 put_screen_char

```
include std/console.e
public procedure put_screen_char(positive_atom line, positive_atom column, sequence char_attr)
```

Stores/displays a sequence of characters with attributes at a given location.

Parameters:

1. `line` : the 1-based line at which to start writing
2. `column` : the 1-based column at which to start writing

Parameters:

3. `char_attr` : a sequence of alternated characters and attribute codes.

Comments:

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

Errors:

The length of `char_attr` must be a multiple of 2.

Comments:

The attributes atom contains the foreground color, background color, and possibly other platform-dependent information controlling how the character is displayed on the screen. If `char_attr` has 0 length, nothing will be written to the screen. The characters are written to the *active page*. It's faster to write several characters to the screen with a single call to `put_screen_char()` than it is to write one character at a time.

Example 1:

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

See Also:

[get_screen_char](#), [display_text_image](#)

2.0.0.520 puts

```
<built-in> procedure puts(integer fn, object text)
```

Output, to a file or device, a single byte (atom) or sequence of bytes. The low order 8-bits of each value is actually sent out. If outputting to the screen you will see text characters displayed.

Parameters:

1. `fn` : an integer, the handle to an opened file or device
2. `text` : an object, either a single character or a sequence of characters.

Errors:

The target file or device must be open.

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.

Example 1:

```
puts(SCREEN, "Enter your first name: ")
```

Example 2:

```
puts(output, 'A') -- the single byte 65 will be sent to output
```

See Also:

[print](#)

2.0.0.521 quote

```
include std/text.e
public function quote(sequence text_in, object quote_pair = {"\"", "\""}, integer esc = - 1, t_
```

Return a quoted version of the first argument.

Parameters:

1. `text_in`: The string or set of strings to quote.
2. `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.
3. `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.
4. `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`.

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"'
```

See Also:

[escape](#)

2.0.0.522 rad2deg

```
include std/math.e
public function rad2deg(object x)
```

Convert an angle measured in radians to an angle measured in degrees

Parameters:

1. `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 $180/\text{PI}$.

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.

Example 1:

```
x = rad2deg(3.385938749)
-- x is 194
```

See Also:

[deg2rad](#)

2.0.0.523 ram_space

```
include std/eumem.e
export sequence ram_space
```

The (pseudo) RAM heap space. Use `malloc` to gain ownership to a heap location and `free` to release it back to the system.

2.0.0.524 rand

```
<built-in> function rand(object maximum)
```

Return a random positive integer.

Parameters:

1. `maximum`: an atom, a cap on the value to return.

Returns:

An **integer**, from 1 to `maximum`.

Errors:

If `ceil(maximum)` is not a positive integer ≤ 1073741823 , an error will occur. It must also be at least 1.

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.

Example 1:

```
s = rand({10, 20, 30})  
-- s might be: {5, 17, 23} or {9, 3, 12} etc.
```

See Also:

[set_rand](#), [ceil](#)

2.0.0.525 rand_range

```
include std/rand.e  
public function rand_range(integer lo, integer hi)
```

Return a random integer from a specified inclusive integer range.

Parameters:

1. `lo` : an integer, the lower bound of the range
2. `hi` : an integer, the upper bound of the range.

Returns:

An **integer**, randomly drawn between `lo` and `hi` inclusive.

Errors:

If `lo` is not less than `hi`, an error will occur.

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.

Example 1:

```
s = rand_range(18, 24)
-- s could be any of: 18, 19, 20, 21, 22, 23 or 24
```

See Also:

[rand](#), [set_rand](#), [rnd](#)

2.0.0.526 range

```
include std/sets.e
public function range(map f, set s)
```

Returns the set of all values taken by a map in some output set.

Parameters:

1. `f` : the map to inspect
2. `set` : the output set

Returns:

The **set**, of all $f(x)$.

Example 1:

```
map f = {3, 2, 5, 2, 4, 6}
set s = {"Albert", "Beatrix", "Conrad", "Doris", "Eugene", "Fabiola"}
set s1 = range(f, s)
-- s1 is now {"Beatrix", "Conrad", "Eugene"}
```

See Also:

[direct_map](#), [image](#)

2.0.0.527 range

```
include std/stats.e
public function range(object data_set)
```

Determines a number of *range* statistics for the data set.

Parameters:

1. `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}
```

See also:

[smallest](#) [largest](#)

Enums used to influence the results of some of these functions.

2.0.0.528 raw_frequency

```
include std/stats.e
public function raw_frequency(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the frequency of each unique item in the data set.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want the frequencies.
2. `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}  
}
```

2.0.0.529 read

```
include std/pipeio.e  
public function read(atom fd, integer bytes)
```

Read `bytes` bytes from handle `fd`

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 `bytes` to handle `fd`

Returns:

A **integer**, number of bytes written, or -1 on error

Example 1:

```
integer bytes_written = write(p[STDIN], "Hello World!")
```

2.0.0.530 read_bitmap

```
include std/image.e  
public function read_bitmap(sequence file_name)
```

Read a bitmap (.BMP) file into a 2-d sequence of sequences (image)

Parameters:

1. `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

```
public constant
    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.

Example 1:

```
x = read_bitmap("c:\\windows\\arcade.bmp")
```

Note:

double backslash needed to get single backslash in a string

See Also:

[save_bitmap](#)

2.0.0.531 read_file

```
include std/io.e
public function read_file(object file, integer as_text = BINARY_MODE)
```

Read the contents of a file as a single sequence of bytes.

Parameters:

1. `file` : an object, either a file path or the handle to an open file.
2. `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 Ctrl-J (NewLine) character, and the first byte value of 26 (Ctrl-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 (Ctrl-Z) will indicate the end of file (all data after this will not be returned to the caller.)

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##
```

See Also:

[write_file](#), [read_lines](#)

2.0.0.532 read_lines

```
include std/io.e
public function read_lines(object file)
```

Read the contents of a file as a sequence of lines.

Parameters:

`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:

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.

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"}
```

See Also:

[gets](#), [write_lines](#), [read_file](#)

2.0.0.533 receive

```
include std/socket.e
public function receive(socket sock, atom flags = 0)
```

Receive data from a bound socket.

Parameters:

1. `sock` : the socket to get data from
2. `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.

2.0.0.534 receive_from

```
include std/socket.e
public function receive_from(socket sock, atom flags = 0)
```

Receive a UDP packet from a given socket

Parameters:

1. `sock`: the server socket
2. `flags` : flags (see [Send Flags](#))

Returns:

A sequence containing { `client_ip`, `client_port`, `data` } or an atom error code.

See Also:

[send_to](#)

2.0.0.535 regex

```
include std/regex.e
public type regex(object o)
```

Regular expression type

2.0.0.536 register_block

```
include std/memory.e
public procedure register_block(atom block_addr, atom block_len, integer protection)
```

Description: Add a block of memory to the list of safe blocks maintained by safe.e (the debug version of memory.e). The block starts at address a. The length of the block is i bytes.

Parameters:

1. `block_addr` : an atom, the start address of the block
2. `block_len` : an integer, the size of the block.
3. `protection` : a constant integer, of the memory protection constants found in machine.e, that describes what access we have to the memory.

Comments:

In memory.e, this procedure does nothing. It is there to simplify switching between the normal and debug version of the library.

This routine is only meant to be used for debugging purposes. safe.e tracks the blocks of memory that your program is allowed to `peek()`, `poke()`, `mem_copy()` etc. These are normally just the blocks that you have allocated using Euphoria's `allocate()` routine, and which you have not yet freed using Euphoria's `free()`. In some cases, you may acquire additional, external, blocks of memory, perhaps as a result of calling a C routine.

If you are debugging your program using safe.e, you must register these external blocks of memory or safe.e will prevent you from accessing them. When you are finished using an external block you can unregister it using `unregister_block()`.

Example 1:

```
atom addr

addr = c_func(x, {})
register_block(addr, 5)
poke(addr, "ABCDE")
unregister_block(addr)
```

Parameters:

See Also:

[unregister_block](#), [safe.e](#)

2.0.0.537 register_block

```
include std/safe.e
public procedure register_block(machine_addr block_addr, positive_int block_len, valid_memory_p
```

2.0.0.538 rehash

```
include std/map.e
public procedure rehash(map the_map_p, integer requested_bucket_size_p = 0)
```

Changes the width, i.e. the number of buckets, of a map. Only effects *large* maps.

Parameters:

1. `m` : the map to resize
2. `requested_bucket_size_p` : a lower limit for the new size.

Comment:

If `requested_bucket_size_p` is not greater than zero, a new width is automatically derived from the current one.

See Also:

[statistics](#), [optimize](#)

2.0.0.539 remainder

```
<built-in> function remainder(object dividend, object divisor)
```

Compute the remainder of the division of two objects using truncated division.

Parameters:

1. `dividend` : any Euphoria object.
2. `divisor` : any Euphoria object.

Parameters:

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.

Errors:

1. If any atom in `divisor` is 0, this is an error condition as it amounts to an attempt to divide by zero.
2. If both `dividend` and `divisor` are sequences, they must be the same length as each other.

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/divisor` 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.

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}
```

See Also:

[mod](#), [Relational operators](#), [Operations on sequences](#)

2.0.0.540 remove

```
<built-in> function remove(sequence target, atom start, atom stop=start)
```

Remove an item, or a range of items from a sequence.

Parameters:

1. `target` : the sequence to remove from.
2. `start` : an atom, the (starting) index at which to remove
3. `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.

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}
```

See Also:

[replace](#), [insert](#), [splice](#), [remove_all](#)

2.0.0.541 remove

```
include std/map.e
public procedure remove(map the_map_p, object the_key_p)
```

Remove an entry with given key from a map.

Parameters:

1. `the_map_p` : the map to operate on
2. `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](#)

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
```

See Also:

[clear](#), [has](#)

2.0.0.542 remove_all

```
include std/sequence.e
public function remove_all(object needle, sequence haystack)
```

Parameters:

Removes all occurrences of some object from a sequence.

Parameters:

1. `needle` : the object to remove.
2. `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.

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."
```

See Also:

[remove](#), [replace](#)

2.0.0.543 `remove_directory`

```
include std/filesys.e
public function remove_directory(sequence dir_name, integer force = 0)
```

Remove a directory.

Parameters:

1. `name` : a sequence, the name of the directory to remove.
2. `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.

Example 1:

```
if not remove_directory("the_old_folder") then
    ("Fatal system problem - could not remove the old folder")
end if
```

See Also:

[create_directory](#), [chdir](#), [clear_directory](#)

2.0.0.544 remove_dups

```
include std/sequence.e
public function remove_dups(sequence source_data, integer proc_option = RD_PRESENTED)
```

Removes duplicate elements

Parameters:

1. `source_data` : A sequence that may contain duplicated elements
2. `proc_option` : One of `RD_INPLACE`, `RD_PRESENTED`, or `RD_SORT`.
 - ◆ `RD_INPLACE` removes items while preserving the original order of the unique items.
 - ◆ `RD_PRESENTED` 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_PRESENTED) --> {4, 7, 9, 7, 2, 5, 9, 0, 4, 5, 6, 5}
? remove_dups(sort(s), RD_PRESENTED) --> {0, 2, 4, 5, 6, 7, 9}
```

2.0.0.545 remove_from

```
include std/sets.e
public function remove_from(object x, set s)
```

Remove an object from a set.

Parameters:

1. `x` : the object to add
2. `S` : the set to remove from

Returns:

A **set**, which is a **copy** of `S`, with `x` removed if it was there.

Example 1:

```
set s0 = {1,2,3,5,7}
s0=remove_from(2,s0)  -- s0 is now {1,3,5,7}
```

See Also:

[remove_from](#), [belongs_to](#), [union](#)

2.0.0.546 remove_item

```
include std/sequence.e
public function remove_item(object needle, sequence haystack)
```

Removes an item from the sequence.

Parameters:

1. `needle` : object to remove.
2. `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}
```

2.0.0.547 remove_subseq

```
include std/sequence.e
public function remove_subseq(sequence source_list, object alt_value = SEQ_NOALT)
```

Removes all sub-sequences from the supplied sequence, optionally replacing them with a supplied alternative value. One common use is to remove all strings from a mixed set of numbers and strings.

Parameters:

1. `source_list` : A sequence from which sub-sequences are removed.
2. `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:

```
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.
```

2.0.0.548 rename_file

```
include std/filesys.e
public function rename_file(sequence old_name, sequence new_name, integer overwrite = 0)
```

Rename a file.

Parameters:

Parameters:

1. `old_name` : a sequence, the name of the file or directory to rename.
2. `new_name` : a sequence, the new name for the renamed file
3. `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](#)

2.0.0.549 repeat

```
<built-in> function repeat(object item, atom count)
```

Create a sequence whose all elements are identical, with given length.

Parameters:

1. `item` : an object, to which all elements of the result will be equal
2. `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`.

Errors:

`count` cannot be less than zero and cannot be greater than 1,073,741,823.

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.

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.
```

See Also:

[repeat_pattern](#), [linear](#)

2.0.0.550 repeat_pattern

```
include std/sequence.e
public function repeat_pattern(sequence pattern, integer count)
```

Returns a periodic sequence, given a pattern and a count.

Parameters:

1. `pattern` : the sequence whose elements are to be repeated
2. `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.

Example 1:

```
s = repeat_pattern({1,2,5},3)
-- s is {1,2,5,1,2,5,1,2,5}
```

See Also:

[repeat](#), [linear](#)

2.0.0.551 replace

<built-in> `function replace(sequence target, object replacement, integer start, integer stop=st`

Replace a slice in a sequence by an object.

Parameters:

1. `target` : the sequence in which replacement will be done.
2. `replacement` : an object, the item to replace with.
3. `start` : an integer, the starting index of the slice to replace.
4. `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.

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}}
```

See Also:

[splice](#), [remove](#), [remove_all](#)

2.0.0.552 replace_all

```
include std/sequence.e
public function replace_all(sequence source, object olddata, object newdata)
```

Replaces all occurrences of `olddata` with `newdata`

Parameters:

1. `source` : the sequence in which replacements will be done.
2. `olddata` : the sequence/item which is going to be replaced. If this is an empty sequence, the `source` is returned as is.
3. `newdata` : the sequence/item which will be the replacement.

Returns:

A **sequence**, which is made of `source` with all `olddata` occurrences replaced by `newdata`.

Comments:

This also removes all `olddata` occurrences when `newdata` is "".

Example:

```
s = replace_all("abracadabra", 'a', 'X')
-- s is now "XbrXcXdXbrX"
s = replace_all("abracadabra", "ra", 'X')
-- s is now "abXcadabX"
s = replace_all("abracadabra", "a", "aa")
-- s is now "aabraacaadaabraa"
s = replace_all("abracadabra", "a", "")
-- s is now "brcdbr"
```

See Also:

[replace](#), [remove_all](#)

2.0.0.553 restrict

```
include std/sets.e
public function restrict(map f, set source, set restriction)
```

Restricts `f` to the intersection of an input set and another set

Parameters:

1. `f` : the map to restrict
2. `source` : the initial source set for `f`
3. `restriction` : the set which will help forming a restricted source set.

Returns:

A **map**, defined on `difference(source, restriction)` which agrees with `f`.

Example 1:

```
set s1 = {1,3,5,7,9,11,13,17,19,23}
map f = [3,7,1,4,5,2,7,1,6,2,10,7}
set s0 = {3,11,13,19,29}
map f0 = restrict(f,s1,s0)
f0 is now: {7,2,7,6,4,7}
```

See Also:

[is_subset](#), [direct_map](#), [difference](#)

2.0.0.554 retain_all

```
include std/sequence.e
public function retain_all(object needles, sequence haystack)
```

Keeps all occurrences of a set of objects from a sequence and removes all others.

Parameters:

1. `needles` : the set of objects to retain.
2. `haystack` : the sequence to remove items not in `needles`.

Returns:

A **sequence** containing only those objects from `haystack` that are also in `needles`.

Example:

```
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"
```

See Also:

[remove](#), [replace](#), [remove_all](#)

2.0.0.555 reverse

```
include std/sequence.e
public function reverse(object target, integer pFrom = 1, integer pTo = 0)
```

Reverse the order of elements in a sequence.

Parameters:

1. `target` : the sequence to reverse.
2. `pFrom` : an integer, the starting point. Defaults to 1.
3. `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
```

2.0.0.556 reverse_map

```
include std/sets.e
public function reverse_map(map f, set s1, sequence s0, set s2)
```

Parameters:

Given a map between two sets, returns the smallest subset whose image contains the set of elements in a list.

Parameters:

1. `f` : the map relative to which reverse images are to be taken
2. `source` : the source set
3. `elements` : the list of elements in `target` to lift to `source`
4. `target` : the target set

Returns:

A **set**, which is included in `source` and contains all antecedents of elements in `elements` by `f`.

Comments:

Elements which `f` does not hit are ignored.

Example 1:

```
set s1,s2
s1={5,7,9,11} s2={13,17,19,23,29}
sequence s0 = {23,13,17,23}
map f = {5,3,1,3,4,5}
set s = reverse_map(f,s1,s0,s2)
s is now {9}.
```

See Also:

[direct_map](#), [fiber_over](#)

2.0.0.557 rfind

```
include std/search.e
public function rfind(object needle, sequence haystack, integer start = length(haystack))
```

Find a needle in a haystack in reverse order.

Parameters:

1. `needle` : an object to search for
2. `haystack` : a sequence to search in
3. `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.

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
```

See Also:

[find](#), [rmatch](#)

2.0.0.558 rmatch

```
include std/search.e
public function rmatch(sequence needle, sequence haystack, integer start = length(haystack))
```

Try to match a needle against some slice of a haystack in reverse order.

Parameters:

1. `needle` : a sequence to search for
2. `haystack` : a sequence to search in
3. `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.

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')
```

See Also:

[rfind](#), [match](#)

2.0.0.559 rnd

```
include std/rand.e
public function rnd()
```

Return a random floating point number in the range 0 to 1.

Parameters:

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.

Example 1:

```
set_rand(1001)
s = rnd()
-- s is 0.2634879318
```

See Also:

[rand](#), [set_rand](#), [rand_range](#)

2.0.0.560 rnd_1

```
include std/rand.e
public function rnd_1()
```

Return a random floating point number in the range 0 to less than 1.

Parameters:

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.

Example 1:

```
set_rand(1001)
s = rnd_1()
-- s is 0.2634879318
```

See Also:

[rand](#), [set_rand](#), [rand_range](#)

2.0.0.561 roll

```
include std/rand.e
public function roll(object desired, integer sides = 6)
```

Simulates the probability of a dice throw.

Parameters:

1. `desired`: an object. One or more desired outcomes.
2. `sides`: an integer. The number of sides on the dice. Default is 6.

Returns:

an integer. 0 if none of the desired outcomes occurred, otherwise the face number that was rolled.

Comments:

The minimum number of sides is 2 and there is no maximum.

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.
```

See Also:

[rnd](#), [chance](#)

2.0.0.562 rotate

```
include std/sequence.e
public function rotate(sequence source, integer shift, integer start = 1, integer stop = length
```

Rotates a slice of a sequence.

Parameters:

1. `source`: sequence to be rotated
2. `shift`: direction and count to be shifted (ROTATE_LEFT or ROTATE_RIGHT)
3. `start`: starting position for shift, defaults 0 1
4. `stop`: stopping position for shift, defaults to `length(source)`

Parameters:

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.

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}
```

See Also:

[slice](#), [head](#), [tail](#)

2.0.0.563 rotate_bits

```
include std/math.e
public function rotate_bits(object source_number, integer shift_distance)
```

Rotates the bits in the input value by the specified distance.

Parameters:

1. `source_number` : **object**: value(s) whose bits will be rotated.
2. `shift_distance` : **integer**: number of bits to be moved by.

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.

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.

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 0xFFFFF0CF
? 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}
```

See Also:

[shift_bits](#)

Arithmetics

2.0.0.564 round

```
include std/math.e
public function round(object a, object precision = 1)
```

Return the argument's elements rounded to some precision

Parameters:

Parameters:

1. `value` : an object, each atom of which will be acted upon, no matter how deeply nested.
2. `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 $1/\text{precision}$.

Comments:

This function may be applied to an atom or to all elements of a sequence.

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
```

See Also:

[floor](#), [ceil](#)

2.0.0.565 routine_id

```
<built-in> function routine_id(sequence routine_name)
```

Return an integer id number for a user-defined Euphoria procedure or function.

Parameters:

1. `routine_name` : a string, the name of the procedure or function.

Returns:

An **integer**, known as a routine id, -1 if the named routine can't be found, else zero or more.

Errors:

`routine_name` should not exceed 1,024 characters.

Comments:

The id number can be passed to `call_proc()` or `call_func()`, to indirectly call the routine named by `routine_name`. This id depends on the internal process of parsing your code, not on `routine_name`.

The routine named `routine_name` must be visible, i.e. callable, at the place where `routine_id()` is used to get the id number. If it is not, -1 is returned.

Indirect calls to the routine can appear earlier in the program than the definition of the routine, but the id number can only be obtained in code that comes after the definition of the routine - see example 2 below.

Once obtained, a valid routine id can be used at any place in the program to call a routine indirectly via `call_proc()/call_func()`, including at places where the routine is no longer in scope.

Some typical uses of `routine_id()` are:

1. Creating a subroutine that takes another routine as a parameter. (See Example 2 below)
2. Using a sequence of routine id's to make a case (switch) statement. Using the `switch statement` is more efficient.
3. Setting up an Object-Oriented system.
4. Getting a routine id so you can pass it to `call_back()`. (See `Platform-Specific Issues`)
5. Getting a routine id so you can pass it to `task_create()`. (See `Multitasking in Euphoria`)
6. Calling a routine that is defined later in a program. This is no longer needed from v4.0 onward.

Note that C routines, callable by Euphoria, also have ids, but they cannot be used where routine ids are, because of the different type checking and other technical issues. See `define_c_proc()` and `define_c_func()`.

Example 1:

```
procedure foo()
    puts(1, "Hello World\n")
end procedure

integer foo_num
foo_num = routine_id("foo")

call_proc(foo_num, {}) -- same as calling foo()
```

Example 2:

```
function apply_to_all(sequence s, integer f)
    -- apply a function to all elements of a sequence
    sequence result
    result = {}
    for i = 1 to length(s) do
```

```
-- we can call add1() here although it comes later in the program
result = append(result, call_func(f, {s[i]}))
end for
return result
end function

function add1(atom x)
    return x + 1
end function

-- add1() is visible here, so we can ask for its routine id
? apply_to_all({1, 2, 3}, routine_id("add1"))
-- displays {2,3,4}
```

See Also:

[call_proc](#), [call_func](#), [call_back](#), [define_c_func](#), [define_c_proc](#), [task_create](#), [Platform-Specific Issues](#), [Indirect routine calling](#)

2.0.0.566 safe_address

```
include std/memory.e
public function safe_address(atom start, integer len, positive_int action)
```

Scans the list of registered blocks for any corruption.

Comments:

safe.e maintains a list of acquired memory blocks. Those gained through `allocate()` are automatically included. Any other block, for debugging purposes, must be registered by `register_block()` and unregistered by `unregister_block()`.

The list is scanned and, if any block shows signs of corruption, it is displayed on the screen and the program terminates. Otherwise, nothing happens.

In `memory.e`, this routine does nothing. It is there to make switching between debugged and normal version of your program easier.

See Also:

[register_block](#), [unregister_block](#)

2.0.0.567 safe_address

```
include std/safe.e
public function safe_address(machine_addr start, natural len, positive_int action)
```

2.0.0.568 safe_address_list

```
include std/safe.e
public sequence safe_address_list
```

2.0.0.569 sample

```
include std/rand.e
public function sample(sequence full_set, integer sample_size, integer return_remaining = 0)
```

Selects a random sample sub-set of items from a population set.

Parameters:

1. `full_set` : a sequence. The set of items from which to take a sample.
2. `sample_size`: an integer. The number of samples to take.
3. `return_remaining`: an integer. If non-zero, the sub-set not selected is also returned. If zero, the default, only the sampled set is returned.

Returns:

a sequence. When `return_remaining = 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:

- If `sample_size` is less than 1, an empty set is returned.
- If `sample_size` is greater than or equal to the population count, the entire population set is returned, but in a random order.

Example 1:

```
set_rand("example")
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxy", 1)}) --> "t"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxy", 5)}) --> "flukq"
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxy", -1)}) --> ""
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxy", 26)}) --> "kghrsxmjoeubaywzlzftcpivqnd"
```

```
printf(1, "%s\n", { sample("abcdefghijklmnopqrstuvwxy", 25)}) --> "omntrqsbjguaikzyvwxfldc"
```

Example 2:

```
-- Deal 4 hands of 5 cards from a standard deck of cards.
sequence theDeck
sequence hands = {}
sequence rt
function new_deck()
    sequence nd
    i = 1 to 4 do
        j = 1 to 13 do
            append(nd, {i,j}) nd =
                end for
        end for
    end for
    return nd
end function
theDeck = new_deck()
for i = 1 to 4 do
    rt = theDeck, 5, 1)
    theDeck = rt
    append(hands, rt[1])
end for
```

-
-
-
-
- *Linux* -- you need GPM server to be running
 - *Windows* -- not implemented yet for the text console
 - *FreeBSD* -- not implemented
 - *OS X* -- not implemented

The following constants can be used to identify and specify mouse events.

2.0.0.570 save_bitmap

```
include std/image.e
public function save_bitmap(two_seq palette_n_image, sequence file_name)
```

Create a .BMP bitmap file, given a palette and a 2-d sequence of sequences of colors.

Parameters:

1. `palette_n_image` : a {palette, image} pair, like `read_bitmap()` returns
2. `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:

```
public constant
    BMP_SUCCESS = 0,
    BMP_OPEN_FAILED = 1,
    BMP_INVALID_MODE = 4 -- invalid graphics mode
                        -- or invalid argument
```

`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.

Example 1:

```
code = save_bitmap({paletteData, imageData},
                  "c:\\example\\a1.bmp")
```

See Also:

[read_bitmap](#)

2.0.0.571 save_map

```
include std/map.e
public function save_map(map the_map_, object file_name_p, integer type_ = SM_TEXT)
```

2.0.0.572 save_text_image

```
include std/console.e
public function save_text_image(text_point top_left, text_point bottom_right)
```

Copy a rectangular block of text out of screen memory

Parameters:

1. `top_left` : the coordinates, given as a pair, of the upper left corner of the area to save.
2. `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 etc.

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-"}
```

See Also:

[display_text_image](#), [get_screen_char](#)

2.0.0.573 scroll

```
include std/graphics.e
public procedure scroll(integer amount, positive_int top_line, positive_int bottom_line)
```

Scroll a region of text on the screen.

Parameters:

1. `amount` : an integer, the number of lines by which to scroll. This is >0 to scroll up and <0 to scroll down.
2. `top_line` : the 1-based number of the topmost line to scroll.
3. `bottom_line` : the 1-based number of the bottom-most line to scroll.

Comments:

inclusive. New blank lines will appear at the top or bottom.

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.

Example 1:

```
bin/ed.ex
```

See Also:

[clear_screen](#), [text_rows](#)

2.0.0.574 section

```
include std/sets.e
public function section(map f)
```

Return a right, and left is possible, inverse of a map over its [range](#).

Parameters:

1. f : the map to invert.

Returns:

A **map**, g such that $f(g(y)) = y$ whenever y is hit by f . and If f is injective, it also holds that $g(f(x)) = x$.

Example 1:

```
map f = {2, 3, 1, 1, 2, 5, 3}, g = section(f)
-- g is now {3,1,2,3,5}.
```

See Also:

[reverse_map](#), [is_injective](#)

2.0.0.575 seek

```
include std/io.e
public function seek(file_number fn, file_position pos)
```

Seek (move) to any byte position in a file.

Parameters:

1. `fn` : an integer, the handle to the file or device to seek()
2. `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.

Errors:

The target file or device must be open.

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.

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
```

See Also:

[get_bytes](#), [puts](#), [where](#)

2.0.0.576 select

```
include std/socket.e
public function select(object sockets_read, object sockets_write, object sockets_err, integer t
```

Determine the read, write and error status of one or more sockets.

Using `select`, you can check to see if a socket has data waiting and is read to be read, if a socket can be written to and if a socket has an error status.

`select` allows for fine-grained control over your sockets, allow you to specify that a given socket only be checked for reading or for only reading and writing, etc.

Parameters:

1. `sockets_read` : either one socket or a sequence of sockets to check for reading.
2. `sockets_write` : either one socket or a sequence of sockets to check for writing.
3. `sockets_err` : either one socket or a sequence of sockets to check for errors.
4. `timeout` : maximum time to wait to determine a sockets status, seconds part
5. `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.

2.0.0.577 send

```
include std/socket.e
public function send(socket sock, sequence data, atom flags = 0)
```

Send TCP data to a socket connected remotely.

Parameters:

1. `sock` : the socket to send data to
2. `data` : a sequence of atoms, what to send
3. `flags` : flags (see [Send Flags](#))

Returns:

An **integer**, the number of characters sent, or -1 for an error.

2.0.0.578 send_to

```
include std/socket.e
public function send_to(socket sock, sequence data, sequence address, integer port = - 1, atom
```

Send a UDP packet to a given socket

Parameters:

1. `sock`: the server socket
2. `data`: the data to be sent
3. `ip`: the ip where the data is to be sent to (ip:port) is acceptable
4. `port`: the port where the data is to be sent on (if not supplied with the ip)
5. `flags` : flags (see [Send Flags](#))

Returns:

An integer status code.

See Also:

[receive_from](#)

2.0.0.579 sequence

```
<built-in> function sequence( object x)
```

Tests the supplied argument *x* to see if it is a sequence or not.

Returns:

1. An integer.
 - ◆ 1 if *x* is a sequence.
 - ◆ 0 if *x* is not an sequence.

Example 1:

```
? sequence(1) --> 0
? sequence({1}) --> 1
? sequence("1") --> 1
```

See Also:

[integer\(\)](#), [object\(\)](#), [atom\(\)](#)

2.0.0.580 sequence_array

```
include std/types.e
public type sequence_array(object x)
```

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
```

Parameters:

```
sequence_array({})          -- TRUE
```

2.0.0.581 sequence_to_set

```
include std/sets.e
public function sequence_to_set(sequence s)
```

Makes a set out of a sequence by sorting it and removing duplicate elements.

Parameters:

1. `s` : the sequence to transform.

Returns:

A **set**, which is the ordered list of distinct elements in `s`.

Example 1:

```
sequence s0 = {1,3,7,5,7,4,1}
set s1 = sequence_to_set(s0)  -- s1 is now {1,3,4,5,7}
```

See Also:

[set](#)

2.0.0.582 sequences_to_map

```
include std/sets.e
public function sequences_to_map(sequence mapped, sequence mapped_to, integer mode)
```

Returns a map which sends each element of some sequence to the corresponding one in another sequence.

Parameters:

1. `mapped` : the source sequence
2. `mapped_to` : the sequence it must map to.
3. `mode` : an integer, nonzero to also return the minimal sets the result map maps.

Returns:

A **sequence**,

- If mode is 0, a map which maps mapped to mapped_to, between the smallest possible sets.
- If mode is not zero, the sequence has length 3. The first element is the map above. The other two elements are the sets derived from the input sequences.

Comments:

Elements in excess in mapped_to are discarded.

If an element is repeated in mapped, only the mapping of the last occurrence is retained.

Example 1:

```
sequence s0, s1
s0 = {2, 3, 4, 1, 4}
s1 = {"aba", "aac", 3, "def"}

map f = sequences_to_map(s0,s1)
-- As a sequence, f is {3,2,1,4,4,4}
```

See Also:

[map](#), [define_map](#)

2.0.0.583 serialize

```
include std/serialize.e
public function serialize(object x)
```

Convert a standard Euphoria object in to a serialized version of it.

Parameters:

1. 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 be 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]
```

2.0.0.584 service_by_name

```
include std/socket.e
public function service_by_name(sequence name, object protocol = 0)
```

Get service information by name.

Parameters:

1. `name` : service name.
2. `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.

Example 1:

```
object result = getservbyname("http")
-- result = { "http", "tcp", 80 }
```

See Also:

[service_by_port](#)

2.0.0.585 service_by_port

```
include std/socket.e
public function service_by_port(integer port, object protocol = 0)
```

Get service information by port number.

Parameters:

1. `port` : port number.
2. `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.

Example 1:

```
object result = getservbyport(80)
-- result = { "http", "tcp", 80 }
```

See Also:

[service_by_name](#)

2.0.0.586 set

```
include std/locale.e
public function set(sequence new_locale)
```

Set the computer locale, and possibly load appropriate translation file.

Parameters:

1. `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` . This is required in some cases.

2.0.0.587 set

```
include std/sets.e
public type set(object s)
```

A set is a sequence in which each item is greater than the previous item.

See Also:

[compare](#)

2.0.0.588 set

```
include std/stack.e
public procedure set(stack sk, object val, integer idx = 1)
```

Update a value on the stack

Parameters:

1. `sk` : the stack being queried
2. `val` : an object, the value to place on the stack
3. `idx` : an integer, the place to inspect. The default is 1 (the top item)

Errors:

If the supplied value of `idx` does not correspond to an existing element, an error occurs.

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](#)

2.0.0.589 set_accumulate_summary

```
include std/unittest.e
public procedure set_accumulate_summary(integer accumulate)
```

Request the test report to save run stats in "unittest.dat" before exiting.

Parameters:

1. `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

2.0.0.590 set_charsets

```
include std/types.e
public procedure set_charsets(sequence charset_list)
```

Sets the definition for one or more defined character sets.

Parameters:

1. `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 , eg. `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.

Example 1:

```
set_charsets({{CS_Whitespace, " \t"}})
t_space('\n') --> FALSE

t_specword('$') --> FALSE
set_charsets({{CS_SpecWord, "_-#$$%"}})
t_specword('$') --> TRUE
```

See Also:

[get_charsets](#)

2.0.0.591 set_colors

```
include syncolor.e
public procedure set_colors(sequence pColorList)
```

2.0.0.592 set_decimal_mark

```
include std/convert.e
public function set_decimal_mark(integer new_mark)
```

Gets, and possibly sets, the decimal mark that [to_number\(\)](#) uses.

Parameters:

1. `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.
-

2.0.0.593 set_def_lang

```
include std/locale.e
public procedure set_def_lang(object langmap)
```

Sets the default language (translation) map

Parameters:

Parameters:

1. `langmap` : A value returned by `lang_load()`, or zero to remove any default map.

Example:

```
set_def_lang( lang_load("appmsgs") )
```

2.0.0.594 set_default_charsets

```
include std/types.e
public procedure set_default_charsets()
```

Sets all the defined character sets to their default definitions.

Example 1:

```
set_default_charsets()
```

2.0.0.595 set_encoding_properties

```
include std/text.e
public procedure set_encoding_properties(sequence en = "", sequence lc = "", sequence uc = "")
```

Sets the table of lowercase and uppercase characters that is used by `lower` and `upper`

Parameters:

1. `en` : The name of the encoding represented by these character sets
2. `lc` : The set of lowercase characters
3. `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.

Example 1:

```
set_encoding_properties( "Elvish", "aeiouy", "AEIOUY")
```

Example 1:

```
set_encoding_properties ( "1251" ) -- Loads a predefined code page.
```

See Also:

[lower](#), [upper](#), [get_encoding_properties](#)

2.0.0.596 set_lang_path

```
include std/locale.e
public procedure set_lang_path(object pp)
```

Set the language path.

Parameters:

1. `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](#)

2.0.0.597 set_option

```
include std/socket.e
public function set_option(socket sock, integer level, integer optname, object val)
```

Set options for a socket.

Parameters:

1. `sock` : an atom, the socket id
2. `level` : an integer, the option level
3. `optname` : requested option (See [Socket Options](#))
4. `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](#)

2.0.0.598 set_rand

```
include std/rand.e
public procedure set_rand(object seed)
```

Reset the random number generator.

Parameters:

1. `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.

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.
```

See Also:

[rand](#)

2.0.0.599 set_sendheader

```
include std/net/http.e
public procedure set_sendheader(object whatheader, sequence whatdata)
```

Set an individual header field.

Parameters:

1. `whatheader` : an object, either an explicit name string or a `HTTP_HEADER_xxx` constant
2. `whatdata` : a string, the associated data

Comments:

If the requested field is not one of the default header fields, the field **MUST** be set by string. This will increase the length of the header overall.

Example 1:

```
set_sendheader("Referer", "search.yahoo.com")
```

See Also:

[get_sendheader](#)

2.0.0.600 set_sendheader_default

```
include std/net/http.e
public procedure set_sendheader_default()
```

Sets header elements to default values. The default User Agent is Opera (currently the most standards compliant). Before setting any header option individually, programs must call this procedure.

See Also:

[get_sendheader](#), [set_sendheader](#), [set_sendheader_useragent_msie](#)

2.0.0.601 set_sendheader_useragent_msie

```
include std/net/http.e
public procedure set_sendheader_useragent_msie()
```

Inform listener that user agent is Microsoft (R) Internet Explorer (TM).

Comments:

This is a convenience procedure to tell a website that a Microsoft Internet Explorer (TM) browser is requesting data. Because some websites format their response differently (or simply refuse data) depending on the browser, this procedure provides a quick means around that. For example, see:

<http://www.missporters.org/podium/nonsupport.aspx>

2.0.0.602 set_test_abort

```
include std/unittest.e
public function set_test_abort(integer abort_test)
```

Set behavior on test failure, and return previous value.

Parameters:

1. `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.

2.0.0.603 set_test_verbosity

```
include std/unittest.e
public procedure set_test_verbosity(atom verbosity)
```

Set the amount of information that is returned about passed and failed tests.

Parameters:

1. `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.

Parameters:

See Also:[test_report](#)

2.0.0.604 set_wait_on_summary

```
include std/unittest.e
public procedure set_wait_on_summary(integer to_wait)
```

Request the test report to pause before exiting.

Parameters:

1. `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](#)

2.0.0.605 setenv

```
include std/os.e
public function setenv(sequence name, sequence val, integer overwrite = 1)
```

Set an environment variable.

Parameters:

1. `name` : a string, the environment variable name
2. `val` : a string, the value to set to
3. `overwrite` : an integer, nonzero to overwrite an existing variable, 0 to disallow this.

Example 1:

```
? setenv("NAME", "John Doe")
? setenv("NAME", "Jane Doe")
? setenv("NAME", "Jim Doe", 0)
```

Parameters:

See Also:[getenv](#), [unsetenv](#)**2.0.0.606 shift_bits**

```
include std/math.e
public function shift_bits(object source_number, integer shift_distance)
```

Moves the bits in the input value by the specified distance.

Parameters:

1. `source_number` : object: The value(s) whose bits will be moved.
2. `shift_distance` : integer: number of bits to be moved by.

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.

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.

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 0xFFFFFC8
? 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
```

Parameters:

```
? shift_bits((-7, 3) --> 0x1FFFFFFF
? shift_bits({48, 121}, 2) --> {12, 30}
```

See Also:

[rotate_bits](#)

2.0.0.607 show_block

```
include std/safe.e
public procedure show_block(sequence block_info)
```

2.0.0.608 show_help

```
include std/cmdline.e
public procedure show_help(sequence opts, object add_help_rid = - 1, sequence cmds = command_line())
```

Show help message for the given opts.

Parameters:

1. `opts` : a sequence of options. See the [cmd_parse](#) for details.
2. `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.
3. `cmds` : a sequence of strings. By default this is the output from [command_line\(\)](#)

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({
```

Parameters:

```
{ "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. <<
```

2.0.0.609 shuffle

```
include std/sequence.e
public function shuffle(sequence seq)
```

Shuffle the elements of a sequence.

Parameters:

Parameters:

1. `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}
```

2.0.0.610 shutdown

```
include std/socket.e
public function shutdown(socket sock, atom method = SD_BOTH)
```

Partially or fully close a socket.

Parameters:

1. `sock` : the socket to shutdown
2. `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.

Parameters:

- **SD_BOTH** - shutdown both send and receive operations.

It may take several minutes for the OS to declare the socket as closed.

2.0.0.611 sign

```
include std/math.e
public function sign(object a)
```

Return -1, 0 or 1 for each element according to it being negative, zero or positive

Parameters:

1. `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)`.

Example 1:

```
i = sign(5)
i is 1

i = sign(0)
-- i is 0

i = sign(-2)
-- i is -1
```

See Also:

[compare](#)

2.0.0.612 sim_index

```
include std/sequence.e
public function sim_index(sequence A, sequence B)
```

Calculates the similarity between two sequences.

Parameters:

1. A : A sequence.
2. 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 beginning 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
```

2.0.0.613 sin

```
<built-in> function sin(object angle)
```

Return the sine of an angle expressed in radians

Parameters:

1. `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.

Example 1:

```
sin_x = sin({.5, .9, .11})
-- sin_x is {.479, .783, .110}
```

See Also:

[cos](#), [arcsin](#), [PI](#), [deg2rad](#)

2.0.0.614 sinh

```
include std/math.e
public function sinh(object a)
```

Computes the hyperbolic sine of an object.

Parameters:

1. `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, $\text{power}(\cosh(x), 2) - \text{power}(\sinh(x), 2) = 1$. Compare with ordinary trigonometry.

Example 1:

```
? sinh(LN2) -- prints out 0.75
```

See Also:

[cosh](#), [sin](#), [arcsinh](#)

2.0.0.615 size

```
include std/map.e
public function size(map the_map_p)
```

Return the number of entries in a map.

Parameters:

`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

Example 1:

```
map the_map_p
put(the_map_p, 1, "a")
put(the_map_p, 2, "b")
? size(the_map_p) -- outputs 2
```

See Also:

[statistics](#)

2.0.0.616 size

```
include std/stack.e
public function size(stack sk)
```

Returns how many elements a stack has.

Parameters:

1. `sk` : the stack being queried.

Returns:

An **integer**, the number of elements in `sk`.

2.0.0.617 skewness

```
include std/stats.e
public function skewness(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns a measure of the asymmetry of a data set. Usually the `data_set` is a probability 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.

Parameters:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `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:

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
```

See also:

[kurtosis](#)

2.0.0.618 sleep

```
include std/os.e
public procedure sleep(atom t)
```

Suspend thread execution. for `t` seconds.

Parameters:

1. `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()`.

Example:

```
puts(1, "Waiting 15 seconds and a quarter...\n")
sleep(15.25)
puts(1, "Done.\n")
```

See Also:

[task_schedule](#), [task_yield](#), [task_delay](#)

2.0.0.619 slice

```
include std/sequence.e
public function slice(sequence source, atom start = 1, atom stop = 0)
```

Return a portion of the supplied sequence.

Parameters:

1. `source` : the sequence from which to get a portion
2. `start` : an integer, the starting point of the portion. Default is 1.
3. `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.

Examples:

```
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}
```

See Also:

[head](#), [mid](#), [tail](#)

2.0.0.620 small

```
include std/stats.e
public function small(sequence data_set, integer ordinal_idx)
```

Determines the k-th smallest value from the supplied set of numbers.

Parameters:

1. `data_set` : The list of values from which the smallest value is chosen.
2. `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 than `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
```

2.0.0.621 smallest

```
include std/stats.e
public function smallest(object data_set)
```

Returns the smallest of the data points.

Parameters:

1. `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: {}
```

See also:

[range](#)

2.0.0.622 socket

```
include std/socket.e
public type socket(object o)
```

Socket type

Parameters:

2.0.0.623 sort

```
include std/sort.e
public function sort(sequence x, integer order = ASCENDING)
```

Sort the elements of a sequence into ascending order.

Parameters:

1. `x` : The sequence to be sorted.
2. `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", i.e. elements that are considered equal might change position relative to each other.

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}
```

See Also:

[compare](#), [custom_sort](#)

2.0.0.624 sort_columns

```
include std/sort.e
public function sort_columns(sequence x, sequence column_list)
```

Parameters:

Sort the rows in a sequence according to a user-defined column order.

Parameters:

1. `x` : a sequence, holding the sequences to be sorted.
2. `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", i.e. elements that are considered equal might change position relative to each other.

Example 1:

```
sequence dirlist
dirlist = dir("c:\\temp")
sequence sorted
-- Order is Size:descending, Name
sorted = sort_columns( dirlist, {-D_SIZE, D_NAME} )
```

See Also:

[compare](#), [sort](#)

2.0.0.625 splice

```
<built-in> function splice(sequence target, object what, integer index)
```

Inserts an object as a new slice in a sequence at a given position.

Parameters:

1. `target` : the sequence to insert into
2. `what` : the object to insert
3. `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.

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}
```

See Also:

[insert](#), [remove](#), [replace](#), &

2.0.0.626 split

```
include std/regex.e
public function split(regex re, string text, integer from = 1, option_spec options = DEFAULT)
```

Split a string based on a regex as a delimiter

Parameters:

1. `re` : a regex which will be used for matching
2. `text` : a string on which search and replace will apply
3. `from` : optional start position
4. `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"
-- }
```

2.0.0.627 split

```
include std/sequence.e
public function split(sequence st, object delim = ' ', integer no_empty = 0, integer limit = 0)
```

Split a sequence on separator delimiters into a number of sub-sequences.

Parameters:

1. `source` : the sequence to split.
2. `delim` : an object (default is `' '`). The delimiter that separates items in `source`.
3. `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.
4. `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 be created, otherwise there is no limit.

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"}
```

See Also:

[split_any](#), [breakup](#), [join](#)

2.0.0.628 split_any

```
include std/sequence.e
public function split_any(sequence source, object delim, integer limit = 0, integer no_empty =
```

Split a sequence by any of the separators in the list of delimiters.

If `limit` is > 0 then limit the number of tokens that will be split to `limit`.

Parameters:

1. `source` : the sequence to split.
2. `delim` : a list of delimiters to split by.
3. `limit` : an integer (default is 0). The maximum number of sub-sequences to create. If zero, there is no limit.
4. `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.

Example 1:

```
result = split_any("One,Two|Three.Four", ",.|")
-- result is {"One", "Two", "Three", "Four"}
result = split_any(",One,,Two|.Three||.Four", ",.|",,1) -- No Empty option
-- result is {"One", "Two", "Three", "Four"}
```

See Also:

[split](#), [breakup](#), [join](#)

2.0.0.629 split_limit

```
include std/regex.e
public function split_limit(regex re, string text, integer limit = 0, integer from = 1, option_
```

2.0.0.630 sprint

```
include std/text.e
public function sprint(object x)
```

Returns the representation of any Euphoria object as a string of characters.

Parameters:

1. `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 10 significant digits, just as with `print()`.

Example 1:

```
s = sprint(12345)
-- s is "12345"
```

Example 2:

```
s = sprint({10,20,30}+5)
-- s is "{15,25,35}"
```

See Also:

[sprintf](#), [printf](#)

2.0.0.631 sprintf

```
<built-in> function sprintf(sequence format, object values)
```

This is exactly the same as `printf()`, except that the output is returned as a sequence of characters, rather than being sent to a file or device.

Parameters:

1. `format` : a sequence, the text to print. This text may contain format specifiers.
2. `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:

1. Converting numbers to strings.
2. Creating strings to pass to `system()`.
3. Creating formatted error messages that can be passed to a common error message handler.

Example 1:

```
s = sprintf("%08d", 12345)
-- s is "00012345"
```

See Also:

[printf](#), [sprintf](#), [format](#)

2.0.0.632 sqrt

```
<built-in> function sqrt(object value)
```

Calculate the square root of a number.

Parameters:

1. `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`.

Errors:

If any atom in `value` is less than zero, an error will occur, as no squared real can be less than zero.

Comments:

This function may be applied to an atom or to all elements of a sequence.

Example 1:

```
r = sqrt(16)
-- r is 4
```

See Also:

[power](#), [Operations on sequences](#)

2.0.0.633 stack

```
include std/stack.e
public type stack(object obj_p)
```

A stack is a sequence of objects with some internal data.

2.0.0.634 statistics

```
include std/map.e
public function statistics(map the_map_p)
```

2.0.0.635 std_library_address

```
include std/machine.e
public type std_library_address(atom addr)
```

Type for memory addresses

an address returned from `allocate()` or `allocate_protect()` or `allocate_code()` or the value 0.

Return Value:

An **integer**, The type will return 1 if the parameter was returned from one of these functions (and has not yet been freed)

Comments:

This type is equivalent to `atom` unless `SAFE` is defined. Only values that satisfy this type may be passed into `free` or `free_code`.

2.0.0.636 stdev

```
include std/stats.e
public function stdev(sequence data_set, object subseq_opt = ST_ALLNUM, integer population_type
```

Returns the standard deviation based of the population.

Parameters:

1. `data_set` : a list of 1 or more numbers for which you want the estimated standard deviation.
2. `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.
3. `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.

The equation for standard deviation is:

$$\text{stdev}(X) ==> \text{SQRT}(\text{SUM}(\text{SQ}(X\{1..N\} - \text{MEAN})) / (N))$$

Example 1:

```
? stdev( {4,5,6,7,5,4,3,7} )           -- Ans: 1.457737974
? stdev( {4,5,6,7,5,4,3,7} ,, ST_FULLLPOP) -- 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_FULLLPOP ) -- 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_FULLLPOP ) -- Ans: 1.984313483
```

See also:

[average](#), [avedev](#)

2.0.0.637 store

```
include std/sequence.e
public function store(sequence target, sequence indexes, object x)
```

Stores something at a location nested arbitrarily deep into a sequence.

Parameters:

1. `target` : the sequence in which to store something
2. `indexes` : a sequence of integers, the path to follow to reach the place where to store
3. `x` : the object to store.

Returns:

A **sequence**, a **copy** of `target` with the specified place `indexes` modified by storing `x` into it.

Errors:

If the path to storage location cannot be followed to its end, or an index is not what one would expect or is not valid, an error about illegal sequence operations will occur.

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}`..

Parameters:

In Euphoria, you can never modify an object by passing it to a routine. You have to get a modified copy and then assign it back to the original.

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}
```

See Also:

[fetch](#), [Subscripting of Sequences](#)

2.0.0.638 string

```
include std/types.e
public type string(object x)
```

Returns:

TRUE if argument is a sequence that only contains zero or more byte characters.

Comment:

A byte 'character' is defined as a integer in the range [0 to 255].

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'}) -- FALSE (contains a negative integer)
string({})           -- TRUE
```

2.0.0.639 subsets

```
include std/sets.e
public function subsets(set s)
```

Returns the list of all subsets of the input set.

Parameters:

1. `s` : the set to enumerate the subsets of.

Returns:

A **sequence**, containing all the subsets of the input set.

Comments:

`s` must not have more than 29 elements, as the length of the output sequence is $\text{power}(2, \text{length}(s))$, which rapidly grows out of integer range. The order in which the subsets are output is implementation dependent.

Example 1:

```
set s0 = {1,3,5,7}
s0 = subsets(s0)  -- s0 is now:
  {}, {1}, {3}, {5}, {7}, {1,3}, {1,5}, {1,7}, {3,5}, {3,7}, {5,7}, {1,3,5}, {1,3,7}, {1,5,7}, {3,5,7}, {1,3,
```

See Also:

[is_subset](#)

2.0.0.640 subtract

```
include std/datetime.e
public function subtract(datetime dt, atom qty, integer interval)
```

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

Parameters:

1. `dt` : the base datetime
2. `qty` : the number of *intervals* to subtract. It should be positive.
3. `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

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
```

See Also:

[add](#), [diff](#)

2.0.0.641 sum

```
include std/math.e
public function sum(object a)
```

Compute the sum of all atoms in the argument, no matter how deeply nested

Parameters:

1. `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

Example 1:

```
a = sum({10, 20, 30})
-- a is 60

a = sum({10.5, {11.2} , 8.1})
-- a is 29.8
```

Parameters:

See Also:

[can_add](#), [product](#), [or_all](#)

2.0.0.642 sum

```
include std/stats.e
public function sum(object data_set, object subseq_opt = ST_ALLNUM)
```

Returns the sum of all the atoms in an object.

Parameters:

1. `data_set` : Either an atom or a list of numbers to sum.
2. `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
```

See also:

[average](#)

2.0.0.643 sum_central_moments

```
include std/stats.e
public function sum_central_moments(object data_set, integer order_mag = 1, object subseq_opt =
```

Returns sum of the central moments of each item in a data set.

Parameters:

1. `data_set` : a list of 1 or more numbers whose mean is used.
2. `order_mag`: An integer. This is the order of magnitude required. Usually a number from 1 to 4, but can be anything.
3. `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
```

See also:

[central_moment](#), [average](#)

2.0.0.644 swap

```
include std/stack.e
public procedure swap(stack sk)
```

Swap the top two elements of a stack

Parameters:

1. `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.

Errors:

If the stack has less than two elements, an error occurs.

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)  -- ""
```

Parameters:

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
```

2.0.0.645 system

<built-in> `procedure system(sequence command, integer mode=0)`

Pass a command string to the operating system command interpreter.

Parameters:

1. `command` : a string to be passed to the shell
2. `mode` : an integer, indicating the manner in which to return from the call.

Errors:

`command` should not exceed 1,024 characters.

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`.

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
```

See Also:

[system_exec](#), [command_line](#), [current_dir](#), [getenv](#)

2.0.0.646 system_exec

<built-in> `function system_exec(sequence command, integer mode=0)`

Try to run the a shell executable command

Parameters:

1. `command` : a string to be passed to the shell, representing an executable command
2. `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.

Errors:

`command` should not exceed 1,024 characters.

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 *WIN32*, `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 *WIN32*, `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.

Example 1:

```
integer exit_code
exit_code = system_exec("xcopy templ.dat temp2.dat", 2)

if exit_code = -1 then
    puts(2, "\n couldn't run xcopy.exe\n")
elseif 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
```

See Also:

[system](#), [abort](#)

2.0.0.647 t_alnum

```
include std/types.e
public type t_alnum(object test_data)
```

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)
```

2.0.0.648 t_alpha

```
include std/types.e
public type t_alpha(object test_data)
```

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)
```

2.0.0.649 t_ascii

```
include std/types.e
public type t_ascii(object test_data)
```

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)
```

2.0.0.650 t_boolean

```
include std/types.e
public type t_boolean(object test_data)
```

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)
```

2.0.0.651 t_bytearray

```
include std/types.e
public type t_bytearray(object test_data)
```

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)
```

2.0.0.652 t_cntrl

```
include std/types.e
public type t_cntrl(object test_data)
```

Returns TRUE if argument is an Control character or if every element of the argument is an Control character.

Returns FALSE if the argument is an empty sequence, or contains sequences, or contains non-Control elements

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)
```

2.0.0.653 t_consonant

```
include std/types.e
public type t_consonant(object test_data)
```

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)
```

2.0.0.654 t_digit

```
include std/types.e
public type t_digit(object test_data)
```

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)

```

2.0.0.655 t_display

```

include std/types.e
public type t_display(object test_data)

```

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)

```

2.0.0.656 t_graph

```
include std/types.e
public type t_graph(object test_data)
```

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)
```

2.0.0.657 t_identifier

```
include std/types.e
public type t_identifier(object test_data)
```

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)
```

Parameters:

```

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)

```

2.0.0.658 t_lower

```

include std/types.e
public type t_lower(object test_data)

```

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)

```

2.0.0.659 t_print

```

include std/types.e
public type t_print(object test_data)

```

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)

```

2.0.0.660 t_punct

```

include std/types.e
public type t_punct(object test_data)

```

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)

```

2.0.0.661 t_space

```
include std/types.e
public type t_space(object test_data)
```

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)
```

2.0.0.662 t_specword

```
include std/types.e
public type t_specword(object test_data)
```

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)

```

2.0.0.663 t_text

```

include std/types.e
public type t_text(object x)

```

Returns:

TRUE if argument is a sequence that only contains zero or more characters.

Comment:

A 'character' is defined as a positive integer or zero. This is a broad definition that may be refined once proper UNICODE support is implemented.

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

```

2.0.0.664 t_upper

```

include std/types.e
public type t_upper(object test_data)

```

Parameters:

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)
```

2.0.0.665 t_vowel

```
include std/types.e
public type t_vowel(object test_data)
```

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)
```

2.0.0.666 t_xdigit

```
include std/types.e
public type t_xdigit(object test_data)
```

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)
```

2.0.0.667 tail

```
<built-in> function tail(sequence source, atom size=length(source) - 1)
```

Return the last `size` item(s) of a sequence.

Parameters:

1. `source` : the sequence to get the tail of.
2. `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.

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}
```

See Also:

[head](#), [mid](#), [slice](#)

2.0.0.668 tan

<built-in> `function tan(object angle)`

Return the tangent of an angle, or a sequence of angles.

Parameters:

1. `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.

Errors:

If any atom in `angle` is an odd multiple of $\pi/2$, an error occurs, as its tangent would be infinite.

Comments:

This function may be applied to an atom or to all elements of a sequence of arbitrary shape, recursively.

Example 1:

```
t = tan(1.0)
-- t is 1.55741
```

See Also:

[sin](#), [cos](#), [arctan](#)

2.0.0.669 tanh

```
include std/math.e
public function tanh(object a)
```

Computes the hyperbolic tangent of an object.

Parameters:

1. 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.

Example 1:

```
? tanh(LN2) -- prints out 0.6
```

See Also:

[cosh](#), [sinh](#), [tan](#), [arctanh](#)

2.0.0.670 task_clock_start

```
<built-in> procedure task_clock_start()
```

Restart the clock used for scheduling real-time tasks.

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.

Example 1:

```
-- freeze the game while the player answers the phone
task_clock_stop()
while get_key() = -1 do
end while
task_clock_start()
```

See Also:

[task_clock_stop](#), [task_schedule](#), [task_yield](#), [task_suspend](#), [task_delay](#)

2.0.0.671 task_clock_stop

```
<built-in> procedure task_clock_stop()
```

Stop the scheduling of real-time tasks.

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.

Parameters:

Scheduling will resume when `task_clock_start()` is called.

Time-shared tasks can continue. The current task can also continue, unless it's 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](#)

2.0.0.672 `task_create`

```
<built-in> function task_create(integer rid, sequence args)
```

Create a new task, given a home procedure and the arguments passed to it.

Parameters:

1. `rid`: an integer, the `routine_id` of a user-defined Euphoria procedure.
2. `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.

Errors:

There must be at most 12 parameters in `args`.

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).

Example 1:

```
mytask = task_create(routine_id("myproc"), {5, 9, "ABC"})
```

See Also:

[task_schedule](#), [task_yield](#), [task_suspend](#), [task_self](#)

2.0.0.673 task_delay

```
include std/task.e
public procedure task_delay(atom delaytime)
```

Suspends a task for a short period, allowing other tasks to run in the meantime.

Parameters:

1. `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](#)

2.0.0.674 task_list

```
<built-in> function task_list()
```

Get a sequence containing the task id's for all active or suspended tasks.

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.

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
```

See Also:

[task_status](#), [task_create](#), [task_schedule](#), [task_yield](#), [task_suspend](#)

2.0.0.675 task_schedule

<built-in> `procedure task_schedule(atom task_id, object schedule)`

Schedule a task to run using a scheduling parameter.

Parameters:

1. `task_id`: an atom, the identifier of a task that did not terminate yet.
2. `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.

A task can switch back and forth between real-time and time-shared. It all depends on the last call to `task_schedule()` for that task. The scheduler never runs a real-time task before the start of its time frame (min value in the {`min`, `max`} pair), and it tries to avoid missing the task's deadline (max value).

For precise timing, you can specify the same value for min and max. However, by specifying a range of times, you give the scheduler some flexibility. This allows it to schedule tasks more efficiently, and avoid non-productive delays. When the scheduler must delay, it calls `sleep()`, unless the required delay is very short. `sleep()` lets the operating system run other programs.

The min and max values can be fractional. If the min value is smaller than the resolution of the scheduler's clock (currently 0.01 seconds on *Windows* or *Unix*) then accurate time scheduling cannot be performed, but the scheduler will try to run the task several times in a row to approximate what is desired.

For example, if you ask for a min time of 0.002 seconds, then the scheduler will try to run your task $.01/0.002 = 5$ times in a row before waiting for the clock to "click" ahead by .01. During the next 0.01 seconds it will run your task (up to) another 5 times etc. provided your task can be completed 5 times in one clock period.

At program start-up there is a single task running. Its task id is 0, and initially it's a time-shared task allowed 1 run per `task_yield()`. No other task can run until task 0 executes a `task_yield()`.

If task 0 (top-level) runs off the end of the main file, the whole program terminates, regardless of what other tasks may still be active.

If the scheduler finds that no task is active, i.e. no task will ever run again (not even task 0), it terminates the program with a 0 exit code, similar to `abort(0)`.

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})
```

See Also:

[task_create](#), [task_yield](#), [task_suspend](#)

2.0.0.676 task_self

```
<built-in> function task_self()
```

Return the task id of the current task.

Comments:

This value may be needed, if a task wants to schedule or suspend itself.

Example 1:

```
-- schedule self
task_schedule(task_self(), {5.9, 6.0})
```

See Also:

[task_create](#), [task_schedule](#), [task_yield](#), [task_suspend](#)

2.0.0.677 task_status

```
<built-in> function task_status(atom task_id)
```

Return the status of a task.

Parameters:

1. `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.

Example 1:

```
integer s

s = task_status(tid)
if s = 1 then
    puts(1, "ACTIVE\n")
elseif s = 0 then
    puts(1, "SUSPENDED\n")
else
    puts(1, "DOESN'T EXIST\n")
end if
```

See Also:

[task_list](#), [task_create](#), [task_schedule](#), [task_suspend](#)

2.0.0.678 task_suspend

```
<built-in> procedure task_suspend(atom task_id)
```

Suspend execution of a task.

Parameters:

1. `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.

Example 1:

```
-- suspend task 15
task_suspend(15)

-- suspend current task
task_suspend(task_self())
```

See Also:

[task_create](#), [task_schedule](#), [task_self](#), [task_yield](#)

2.0.0.679 task_yield

```
<built-in> procedure task_yield()
```

Yield control to the scheduler. The scheduler can then choose another task to run, or perhaps let the current task continue running.

Comments:

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's 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.

Parameters:

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.

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
```

See Also:

[task_create](#), [task_schedule](#), [task_suspend](#)

2.0.0.680 temp_file

```
include std/filesys.e
public function temp_file(sequence temp_location = "", sequence temp_prefix = "", sequence temp
```

Returns a file name that can be used as a temporary file.

Parameters:

1. `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.
 - ◇ If it doesn't exist, the directory name portion is used.
2. `temp_prefix`: A sequence: The is prepended to the start of the generated file name. The default is "".
3. `temp_extn`: A sequence: The is a file extension used in the generated file. The default is "_T_".
4. `reserve_temp`: An integer: If not zero an empty file is created using the generated name. The default is not to reserve (create) the file.

Returns:

A **sequence**, A generated file name.

Comments:**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_
```

2.0.0.681 test_equal

```
include std/unittest.e
public procedure test_equal(sequence name, object expected, object outcome)
```

Records whether a test passes by comparing two values.

Parameters:

1. `name`: a string, the name of the test
2. `expected`: an object, the expected outcome of some action
3. `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](#)

2.0.0.682 test_exec

```
include std/memconst.e
export function test_exec(valid_memory_protection_constant protection)
```

2.0.0.683 test_fail

```
include std/unittest.e
public procedure test_fail(sequence name)
```

Records that a test failed.

Parameters:

1. `name` : a string, the name of the test

See Also:

[test_equal](#), [test_not_equal](#), [test_true](#), [test_false](#), [test_pass](#)

2.0.0.684 test_false

```
include std/unittest.e
public procedure test_false(sequence name, object outcome)
```

Records whether a test passes by comparing two values.

Parameters:

Parameters:

1. `name` : a string, the name of the test
2. `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](#)

2.0.0.685 test_not_equal

```
include std/unittest.e
public procedure test_not_equal(sequence name, object a, object b)
```

Records whether a test passes by comparing two values.

Parameters:

1. `name` : a string, the name of the test
2. `expected` : an object, the expected outcome of some action
3. `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](#)

2.0.0.686 test_pass

```
include std/unittest.e
public procedure test_pass(sequence name)
```

Records that a test passed.

Parameters:

1. name : a string, the name of the test

See Also:

[test_equal](#), [test_not_equal](#), [test_true](#), [test_false](#), [test_fail](#)

Possible style values for message_box() style sequence

2.0.0.687 test_read

```
include std/memconst.e
export function test_read(valid_memory_protection_constant protection)
```

2.0.0.688 test_report

```
include std/unittest.e
public procedure test_report()
```

Output test report

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](#)

2.0.0.689 test_true

```
include std/unittest.e
public procedure test_true(sequence name, object outcome)
```

Records whether a test passes.

Parameters:

1. name : a string, the name of the test
2. 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](#)

2.0.0.690 test_write

```
include std/memconst.e
export function test_write(valid_memory_protection_constant protection)
```

2.0.0.691 text_color

```
include std/graphics.e
public procedure text_color(color c)
```

Set the foreground text color.

Parameters:

1. 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.

Example:

```
text_color(BRIGHT_BLUE)
```

See Also:

`bk_color` , `clear_screen`

2.0.0.692 text_rows

```
include std/console.e
public function text_rows(positive_int rows)
```

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

Parameters:

1. `rows` : an integer, the desired number of rows.

Platforms:

Not *Unix*

Returns:

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

Comments:

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

See Also:

[graphics_mode](#), [video_config](#)

2.0.0.693 threshold

```
include std/map.e
public function threshold(integer new_value_p = 0)
```

Gets or Sets the threshold value that determines at what point a small map converts into a large map structure. Initially this has been set to 50, meaning that maps up to 50 elements use the *small map* structure.

Parameters:

1. `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`.

2.0.0.694 time

```
<built-in> function time()
```

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

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.

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
```

See Also:

[date](#), [now](#)

2.0.0.695 to_integer

```
include std/convert.e
public function to_integer(object data_in, integer def_value = 0)
```

Converts an object into a integer.

Parameters:

1. `data_in`: Any Euphoria object.
2. `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.

Examples:

```

? 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)

```

These are returned from `get` and `value`.

2.0.0.696 to_number

```

include std/convert.e
public function to_number(sequence text_in, integer return_bad_pos = 0)

```

Converts the text into a number.

Parameters:

1. `text_in`: A string containing the text representation of a number.
2. `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:

1. 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.
2. 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()`.
3. You can supply a single leading or trailing sign. Either a minus (-) or plus (+).
4. 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 $(3845 / 1000) ==> 3.845$.
5. You can have single currency symbol before the first digit or after the last digit. A currency symbol is any character of the string: "\$£¤¥ ".
6. You can have any number of whitespace characters before the first digit and after the last digit.
7. 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.
8. 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.
9. 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 IEEE Floating point format.

Examples:

```
object val
  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
```

2.0.0.697 to_unix

```
include std/datetime.e
public function to_unix(datetime dt)
```

Parameters:

Convert a datetime value to the unix numeric format (seconds since EPOCH_1970)

Parameters:

1. dt : a datetime to be queried.

Returns:

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

Example 1:

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

See Also:

[from_unix](#), [format](#)

2.0.0.698 top

```
include std/stack.e  
public function top(stack sk)
```

Retrieve the top element on a stack.

Parameters:

1. sk : the stack to inspect.

Returns:

An **object**, the top element on a stack.

Comments:

This call is equivalent to `at (sk, 1)`.

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
```

See Also:

[at](#), [pop](#), [peek_top](#), [last](#)

2.0.0.699 trailer

```
include std/memory.e
export constant trailer
```

2.0.0.700 trailer

```
include std/safe.e
export constant trailer
```

2.0.0.701 transform

```
include std/sequence.e
public function transform(sequence source_data, object transformer_rids)
```

Transforms the input sequence by using one or more user-supplied transformers.

Parameters:

1. `source_data` : A sequence to be transformed.
2. `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` parameter 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.

Examples:

```
res = transform(" hello   ", {
    {routine_id("trim"), " ",0},
    routine_id("upper"),
    {routine_id("replace_all"), "O", "A"}
})
--> "HELLA"
```

2.0.0.702 translate

```
include std/locale.e
public function translate(sequence word, object langmap = 0, object defval = "", integer mode =
```

Translates a word, using the current language file.

Parameters:

1. `word` : a sequence, the word to translate.
2. `langmap` : Either a value returned by `lang_load()` or zero to use the default language map
3. `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.
4. `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.

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))
```

See Also:

[set](#), [lang_load](#)

2.0.0.703 transmute

```
include std/sequence.e
public function transmute(sequence source_data, sequence current_items, sequence new_items, int
```

Replaces all instances of any element from the `current_items` sequence that occur in the `source_data` sequence with the corresponding item from the `new_items` sequence.

Parameters:

1. `source_data` : a sequence, the data that might contain elements from `current_items`
2. `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`.
3. `new_items` : a sequence, the set of replacement data for any matches found.
4. `start` : an integer, the starting point of the search. Defaults to 1.
5. `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`.

For example, you can find all occurrences of 'h', 's', and 't' in a string and replace them with '1', '2', and '3' respectively.

```
transmute(SomeString, "hts", "123")
```

However, the routine can also be used to scan for sub-sequences and/or replace matches with sequences rather than single elements. This is done by making the first element in `current_items` and/or `new_items` an empty sequence.

For example, to find all occurrences 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"
```

The similar syntax is used to indicates that replacements are sequences and not single elements.

```
res = transmute("the school shoes", {{}, "sh", "th", "sch"}, {{}, "SH", "TH", "SCH"})
-- res becomes "THE SCHool SHoes"
```

Using this option also allows you to remove matching data.

```
res = transmute("the school shoes", {{}, "sh", "th", "sch"}, {{}, "", "", ""})
-- res becomes "e ool oes"
```

Another thing to note is that when using this syntax, you can still mix together atoms and sequences.

```
res = transmute("the school shoes", {{}, "sh", 't', "sch"}, {{}, 'x', "TH", "SCH"})
-- res becomes "THhe SCHool xoes"
```

Example 1:

```
res = transmute("John Smith enjoys uncooked apples.", "aeiouy", "YUOIEA")
-- res is "JIhn SmOth UnjIAs EncIIkUd YpplUs."
```

See Also:

[find](#), [match](#), [replace](#), [mapping](#)

2.0.0.704 trim

```
include std/text.e
public function trim(sequence source, object what = "\t\r\n", integer ret_index = 0)
```

Trim all items in the supplied set from both the left end (head/start) and right end (tail/end) of a sequence.

Parameters:

1. `source` : the sequence to trim.
2. `what` : the set of item to trim from `source` (defaults to "`\t\r\n`").
3. `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}`.

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}
```

See Also:

[trim_head](#), [trim_tail](#)

2.0.0.705 trim_head

```
include std/text.e
public function trim_head(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

Trim all items in the supplied set from the leftmost (start or head) of a sequence.

Parameters:

1. `source` : the sequence to trim.
2. `what` : the set of item to trim from `source` (defaults to "`\t\r\n`").
3. `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`.

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
```

See Also:

[trim_tail](#), [trim](#), [pad_head](#)

2.0.0.706 trim_tail

```
include std/text.e
public function trim_tail(sequence source, object what = " \t\r\n", integer ret_index = 0)
```

Trim all items in the supplied set from the rightmost (end or tail) of a sequence.

Parameters:

1. `source` : the sequence to trim.
2. `what` : the set of item to trim from `source` (defaults to "`\t\r\n`").
3. `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`.

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
```

See Also:

[trim_head](#), [trim](#), [pad_tail](#)

2.0.0.707 trsprintf

```
include std/locale.e
public function trsprintf(sequence fmt, sequence data, object langmap = 0)
```

Returns a formatted string with automatic translation performed on the parameters.

Parameters:

1. `fmt` : A sequence. Contains the formatting string. see `printf()` for details.
2. `data` : A sequence. Contains the data that goes into the formatted result. see `printf` for details.
3. `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.

Examples:

```
-- 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?"
```

2.0.0.708 true_color

```
include std/graphcst.e
export constant true_color
```

Parameters:

2.0.0.709 trunc

```
include std/math.e
public function trunc(object x)
```

Return the integer portion of a number.

Parameters:

1. value : any Euphoria object.

Returns:

An **object**, the shape of which depends on `value`'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`

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}
```

See Also:

[floor](#) [frac](#)

2.0.0.710 type_of

```
include std/map.e
public function type_of(map the_map_p)
```

Parameters:

Determines the type of the map.

Parameters:

1. *m* : A map

Returns:

An **integer**, Either *SMALLMAP* or *LARGEMAP*

2.0.0.711 uname

```
include std/os.e
public function uname()
```

Retrieves the name of the host OS.

Returns:

A **sequence**, starting with the OS name. If identification fails, returns an OS name of UNKNOWN. Extra information depends on the OS.

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 occurred.

Comments:

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.

2.0.0.712 union

```
include std/sets.e
public function union(set S1, set S2)
```

Returns the set of elements belonging to any of two sets.

Parameters:

1. S1: one of the sets to merge
2. S2: the other set.

Returns:

The **set** of all elements belonging to S1 or S2, and possibly to both.

Example 1:

```
set s0, s1, s2
s1={1, 3, 5, 7} s2={-1, 2, 3, 7, 11}
s0=union(s1, s2)  -- s0 is now {-1, 1, 2, 3, 5, 7, 11}.
```

See Also:

[is_subset](#), [subsets](#), [belongs_to](#)

2.0.0.713 unlock_file

```
include std/io.e
public procedure unlock_file(file_number fn, byte_range r = {})
```

Unlock (a portion of) an open file.

Parameters:

1. `fn` : an integer, the handle to the file or device to (partially) lock.
2. `r` : a sequence, defining a section of the file to be locked, or `{ }` for the whole file (the default).

Errors:

The target file or device must be open.

Comments:

You must have previously locked the file using `lock_file()`. On *WIN32* 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](#)

2.0.0.714 unregister_block

```
include std/memory.e
public procedure unregister_block(atom block_addr)
```

Remove a block of memory from the list of safe blocks maintained by `safe.e` (the debug version of `memory.e`).

Parameters:

1. `block_addr` : an atom, the start address of the block

Comments:

In `memory.e`, this procedure does nothing. It is there to simplify switching between the normal and debug version of the library.

This routine is only meant to be used for debugging purposes. Use it to unregister blocks of memory that you have previously registered using `register_block()`. By unregistering a block, you remove it from the list of safe blocks maintained by `safe.e`. This prevents your program from performing any further reads or writes of memory within the block.

See `register_block()` for further comments and an example.

See Also:

[register_block](#), [safe.e](#)

2.0.0.715 unregister_block

```
include std/safe.e
public procedure unregister_block(machine_addr block_addr)
```

2.0.0.716 unsetenv

```
include std/os.e
public function unsetenv(sequence env)
```

Unset an environment variable

Parameters:

1. name : name of environment variable to unset

Example 1:

```
? unsetenv("NAME")
```

See Also:

[setenv](#), [getenv](#)

2.0.0.717 upper

```
include std/text.e
public function upper(object x)
```

Convert an atom or sequence to upper case.

Parameters:

1. 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 non-Windows, 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.

Example 1:

```
s = upper("Euphoria")
-- s is "EUPHORIA"

a = upper('b')
-- a is 'B'

s = upper({"Euphoria", "Programming"})
-- s is {"EUPHORIA", "PROGRAMMING"}
```

See Also:

[lower](#), [proper](#), [set_encoding_properties](#), [get_encoding_properties](#)

2.0.0.718 valid

```
include std/eumem.e
export function valid(object mem_p, object mem_struct_p = 1)
```

Validates a block of (pseudo) memory

Parameters:

1. `mem_p` : The handle to a previously acquired [ram_space](#) location.
2. `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
. . . do some processing . .
if valid(my_spot, length(my_data)) then
    free(my_spot)
end if
```

2.0.0.719 valid_index

```
include std/sequence.e
public function valid_index(sequence st, object x)
```

Checks whether an index exists on a sequence.

Parameters:

1. *s* : the sequence for which to check
2. *x* : an object, the index to check.

Returns:

An **integer**, 1 if *s* [*x*] makes sense, else 0.

Example 1:

```
i = valid_index({51,27,33,14},2)
-- i is 1
```

See Also:

[Subscripting of Sequences](#)

2.0.0.720 valid_memory_protection_constant

```
include std/machine.e
public type valid_memory_protection_constant(integer x)
```

protection constants type

2.0.0.721 valid_memory_protection_constant

```
include std/memconst.e
export type valid_memory_protection_constant(integer x)
```

2.0.0.722 valid_wordsize

```
include std/memconst.e
export type valid_wordsize(integer i)
```

2.0.0.723 value

```
include std/get.e
public function value(sequence st, integer start_point = 1, integer answer = GET_SHORT_ANSWER)
```

Read, from a string, a human-readable string of characters representing a Euphoria object. Convert the string into the numeric value of that object.

Parameters:

1. `st` : a sequence, from which to read text
2. `offset` : an integer, the position at which to start reading. Defaults to 1.
3. `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.

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}
```

See Also:

[get](#)

2.0.0.724 values

```
include std/map.e
public function values(map the_map, object keys = 0, object default_values = 0)
```

Return values, without their keys, from a map.

Parameters:

Parameters:

1. `the_map` : the map being queried
2. `keys` : optional, key list of values to return.
3. `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.

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 }
```

See Also:

[get](#), [keys](#), [pairs](#)

2.0.0.725 version

```
include info.e
public function version()
```

Get the version, as an integer, of the host Euphoria

Returns:

An **integer**, representing Major, Minor and Patch versions. Version 4.0.0 will return 40000, 4.0.1 will return 40001, 5.6.2 will return 50602, 5.12.24 will return 512624, etc...

2.0.0.726 version_major

```
include info.e
public function version_major()
```

Get the major version of the host Euphoria

Returns:

An **integer**, representing the Major version number. Version 4.0.0 will return 4, version 5.6.2 will return 5, etc...

2.0.0.727 version_minor

```
include info.e
public function version_minor()
```

Get the minor version of the hosting Euphoria

Returns:

An **integer**, representing the Minor version number. Version 4.0.0 will return 0, 4.1.0 will return 1, 5.6.2 will return 6, etc...

2.0.0.728 version_patch

```
include info.e
public function version_patch()
```

Get the patch version of the hosting Euphoria

Returns:

An **integer**, representing the Path version number. Version 4.0.0 will return 0, 4.0.1 will return 1, 5.6.2 will return 2, etc...

2.0.0.729 version_revision

```
include info.e
public function version_revision()
```

Get the source code revision of the hosting Euphoria

Returns:

A text **sequence**, containing the source code management system's revision number that the executing Euphoria was built from.

2.0.0.730 version_string

```
include info.e
public function version_string()
```

Get a normal version string

Returns:

A **#sequence**, representing the Major, Minor, Patch, Type and Revision all in one string.

Example return values:

- "4.0.0 alpha 3 (r1234)"
 - "4.0.0 release (r271)"
 - "4.0.2 beta 1 (r2783)"
-

2.0.0.731 version_string_long

```
include info.e
public function version_string_long()
```

Get a long version string

Returns:

Same **value**, as **version_string** with the addition of the platform name.

Example return values:

- "4.0.0 alpha 3 for Windows"
 - "4.0.0 release for Linux"
 - "5.6.2 release for OS X"
-
-

2.0.0.732 version_string_short

```
include info.e
public function version_string_short()
```

Get a short version string

Returns:

A **sequence**, representing the Major, Minor and Patch all in one string.

Example return values:

- "4.0.0"
 - "4.0.2"
 - "5.6.2"
-

2.0.0.733 version_type

```
include info.e
public function version_type()
```

Parameters:

Get the type version of the hosting Euphoria

Returns:

A **sequence**, representing the Type version string. Version 4.0.0 alpha 1 will return `alpha 1`. 4.0.0 beta 2 will return `beta 2`. 4.0.0 final, or release, will return `release`.

2.0.0.734 video_config

```
include std/graphcst.e
public function video_config()
```

Return a description of the current video configuration:

Returns:

A **sequence**, of 10 non-negative integers, laid out as follows:

1. color monitor? -- 1 0 if monochrome, 1 otherwise
2. current video mode
3. number of text rows in console buffer
4. number of text columns in console buffer
5. screen width in pixels
6. screen height in pixels
7. number of colors
8. number of display pages
9. number of text rows for current screen size
10. 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_LINES
- VC_COLUMNS

Parameters:



- VC_SCRNLINES
- VC_SCRNCOLS

This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

Example:

```
vc = video_config()
-- vc could be {1, 3, 300, 132, 0, 0, 32, 8, 37, 90}
```

See Also:

[graphics_mode](#)

2.0.0.735 vlookup

```
include std/search.e
public function vlookup(object find_item, sequence grid_data, integer source_col, integer target_col, object def_value)
```

If the supplied item is in a source grid column, this returns the corresponding element from the target column.

Parameters:

1. `find_item`: an object that might exist in `source_col`.
2. `grid_data`: a 2D grid sequence that might contain `pItem`.
3. `source_col`: an integer. The column number to look for `find_item`.
4. `target_col`: an integer. The column number from which the corresponding item will be returned.
5. `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.

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.

Returns:

an object

- If `find_item` is found in the `source_col` column then this is the corresponding element from the `target_col` column.

Examples:

```
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, "?") --> "?"
```

2.0.0.736 vslice

```
include std/sequence.e
public function vslice(sequence source, atom colno, object error_control = 0)
```

Perform a vertical slice on a nested sequence

Parameters:

1. `source` : the sequence to take a vertical slice from
2. `colno` : an atom, the column number to extract (rounded down)
3. `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]`.

Errors:

If an element is not defined and `error_control` is 0, an error occurs. If `colno` is less than 1, it cannot be any valid column, and an error occurs.

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. A short vertical slice is returned if `error_control` is exhausted.

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}
```

See Also:

[slice](#), [project](#)

2.0.0.737 w32_name_canonical

```
include std/localeconv.e
public constant w32_name_canonical
```

Canonical locale names for *WIN32*:

Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Afrikaans_South Africa	Afrikaans_South Africa	Afrikaans_South Africa
Basque_Spain	Basque_Spain	Belarusian_Belarus
Belarusian_Belarus	Belarusian_Belarus	Belarusian_Belarus

Parameters:

Belarusian_Belarus	Belarusian_Belarus	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Catalan_Spain	Catalan_Spain	Catalan_Spain
Danish_Denmark	Danish_Denmark	Danish_Denmark
Danish_Denmark	Danish_Denmark	English_Australia
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
English_United States	English_United States	English_United States
Finnish_Finland	French_France	French_France
French_France	French_France	French_France
Hungarian_Hungary	Hungarian_Hungary	Hungarian_Hungary
Hungarian_Hungary	Hungarian_Hungary	Hungarian_Hungary
Hungarian_Hungary	Italian_Italy	Italian_Italy
Italian_Italy	Romanian_Romania	Romanian_Romania
Russian_Russia	Russian_Russia	Russian_Russia
Russian_Russia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia

Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia	Serbian (Cyrillic)_Serbia
Serbian (Cyrillic)_Serbia	Slovak_Slovakia	Estonian_Estonia
Estonian_Estonia	Estonian_Estonia	Swedish_Sweden
Swedish_Sweden	Swedish_Sweden	Ukrainian_Ukraine
Ukrainian_Ukraine	Ukrainian_Ukraine	Ukrainian_Ukraine

2.0.0.738 w32_names

```
include std/localeconv.e
public constant w32_names
```

2.0.0.739 wait_key

```
include std/console.e
public function wait_key()
```

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

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](#)

2.0.0.740 walk_dir

```
include std/filesys.e
public function walk_dir(sequence path_name, object your_function, integer scan_subdirs = FALSE)
```

Generalized Directory Walker

Parameters:

1. `path_name` : a sequence, the name of the directory to walk through
2. `your_function` : the routine id of a function that will receive each path returned from the result of `dir_source`, one at a time.
3. `scan_subdirs` : an optional integer, 1 to also walk though subfolders, 0 (the default) to skip them all.
4. `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, 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.

Parameters:

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`.

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
    return 0 -- Ignore directories
  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)
  dobject
  (path)dir
  atom(d)if then
    d      return
  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"))
```

See Also:

[dir](#), [sort](#), [sort_columns](#)

2.0.0.741 warning

<built-in> `procedure warning(sequence message)`

Causes the specified warning message to be displayed as a regular warning.

Parameters:

1. `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 didn't write. It may be desirable then to influence, from inside the library, that code you didn't 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 `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.

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...
```

See Also:

[warning_file](#)

2.0.0.742 warning_file

```
include std/error.e
public procedure warning_file(object file_path)
```

Parameters:

Specify a file path where to output warnings.

Parameters:

1. `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.

Example 1:

```
warning_file("warnings.lst")
-- some code
warning_file(0)
-- changed opinion: warnings will go to standard error as usual
```

See Also:

[without warning](#), [warning](#)

2.0.0.743 weeks_day

```
include std/datetime.e
public function weeks_day(datetime dt)
```

Get the day of week of the datetime dt.

Parameters:

1. `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.
```

2.0.0.744 where

```
include std/io.e
public function where(file_number fn)
```

Retrieves the current file position for an opened file or device.

Parameters:

1. `fn` : an integer, the handle to the file or device to query.

Returns:

An **atom**, the current byte position in the file.

Errors:

The target file or device must be open.

Comments:

The file position 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.

2.0.0.745 which_bit

```
include std/flags.e
public function which_bit(object theValue)
```

Tests if the supplied value has only a single bit on in its representation.

Parameters:

1. `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.

Examples:

```
? 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
```

2.0.0.746 wildcard_file

```
include std/wildcard.e
public function wildcard_file(sequence pattern, sequence filename)
```

Determine whether a file name matches a wildcard pattern.

Parameters:

1. `pattern` : a string, the pattern to match
2. `filename` : the string to be matched against

Returns:

An **integer**, TRUE if `filename` matches `pattern`, else FALSE.

Comments:

* matches any 0 or more characters, ? matches any single character. On *Unix* the character comparisons are case sensitive. On Windows they are not.

You might use this function to check the output of the `dir()` routine for file names that match a pattern supplied by the user of your program.

Example 1:

```
i = wildcard_file("AB*CD.?", "aB123cD.e")
-- i is set to 1 on Windows, 0 on Linux or FreeBSD
```

Example 2:

```
i = wildcard_file("AB*CD.?", "abcd.ex")
-- i is set to 0 on all systems,
-- because the file type has 2 letters not 1
```

Example 3:

```
bin/search.ex
```

See Also:

[is_match](#), [dir](#)

2.0.0.747 Header

- byte 0: magic number for this file-type: 77
- byte 1: version number (major)
- byte 2: version number (minor)
- byte 3: 4-byte pointer to block of table headers
- byte 7: number of free blocks
- byte 11: 4-byte pointer to block of free blocks

2.0.0.748 Block of table headers

- -4: allocated size of this block (for possible reallocation)
- 0: number of table headers currently in use
- 4: table header1
- 16: table header2
- 28: etc.

2.0.0.749 Table header

- 0: pointer to the name of this table
- 4: total number of records in this table
- 8: number of blocks of records

- 12: pointer to the index block for this table

There are two levels of pointers. The logical array of key pointers is split up across many physical blocks. A single index block is used to select the correct small block. This allows inserts and deletes to be made without having to shift a large number of key pointers. Only one small block needs to be adjusted. This is particularly helpful when the table contains many thousands of records.

2.0.0.750 Index block

one per table

- -4: allocated size of index block
- 0: number of records in 1st block of key pointers
- 4: pointer to 1st block
- 8: number of records in 2nd " "
- 12: pointer to 2nd block
- 16: etc.

2.0.0.751 Block of key pointers

many per table

- -4: allocated size of this block in bytes
- 0: key pointer 1
- 4: key pointer 2
- 8: etc.

2.0.0.752 Free list

in ascending order of address

- -4: allocated size of block of free blocks
- 0: address of 1st free block
- 4: size of 1st free block
- 8: address of 2nd free block
- 12: size of 2nd free block
- 16: etc.

The key value and the data value for a record are allocated space as needed. A pointer to the data value is stored just before the key value. Euphoria objects, key and data, are stored in a compact form.

All allocated blocks have the size of the block in bytes, stored in the four bytes just before the address.

2.0.0.753 wrap

```
include std/graphics.e
public procedure wrap(boolean on)
```

Determine whether text will wrap when hitting the rightmost column.

Parameters:

1. on : a boolean, 0 to truncate text, nonzero 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.

Example:

```
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'
```

See Also:

[puts](#), [position](#)

2.0.0.754 write

```
include std/pipeio.e
public function write(atom fd, sequence str)
```

2.0.0.755 write_file

```
include std/io.e
public function write_file(object file, sequence data, integer as_text = BINARY_MODE)
```

Write a sequence of bytes to a file.

Parameters:

1. `file`: an object, either a file path or the handle to an open file.
2. `data`: the sequence of bytes to write
3. `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 NewLine to be written out according to the operating system's end of line convention. In Unix this is Ctrl-J and in Windows this is the pair {Ctrl-L, Ctrl-J}.
 - ◆ **UNIX_TEXT** ensures that lines are written out with unix style line endings (Ctrl-J).
 - ◆ **DOS_TEXT** ensures that lines are written out with Windows style line endings {Ctrl-L, Ctrl-J}.

Returns:

An **integer**, 1 on success, -1 on failure.

Errors:

If `puts` cannot write `data`, a runtime error will occur.

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 any of the text modes, the file is truncated at the first Ctrl-Z character in the input data.

Example 1:

```
if write_file("data.txt", "This is important data\nGoodbye") = -1 then
    puts(STDERR, "Failed to write data\n")
end if
```

See Also:

[read_file](#), [write_lines](#)

2.0.0.756 write_lines

```
include std/io.e
public function write_lines(object file, sequence lines)
```

Parameters:

Write a sequence of lines to a file.

Parameters:

1. `file` : an object, either a file path or the handle to an open file.
2. `lines` : the sequence of lines to write

Returns:

An **integer**, 1 on success, -1 on failure.

Errors:

If `puts()` cannot write some line of text, a runtime error will occur.

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.

Example 1:

```
if write_lines("data.txt", {"This is important data", "Goodbye"}) != -1 then
    puts(STDERR, "Failed to write data\n")
end if
```

See Also:

[read_lines](#), [write_file](#), [puts](#)

2.0.0.757 `writef`

```
include std/io.e
public procedure writef(object fm, object data = {}, object fn = 1, object data_not_string = 0)
```

Write formatted text to a file..

Parameters:

There are two ways to pass arguments to this function,

1. Traditional way with first arg being a file handle.
 1. : integer, The file handle.
 2. : sequence, The format pattern.
 3. : object, The data that will be formatted.
 4. `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.
2. Alternative way with first argument being the format pattern.
 1. : sequence, Format pattern.
 2. : sequence, The data that will be formatted,
 3. : object, The file to receive the formatted output. Default is to the STDOUT device (console).
 4. `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 third 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,

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", "
-- 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"
```

See Also:

[text:format](#), [writefln](#), [write_lines](#)

2.0.0.758 writefln

```
include std/io.e
public procedure writefln(object fm, object data = {}, object fn = 1, object data_not_string =
```

Write formatted text to a file, ensuring that a new line is also output.

Parameters:

1. `fm` : sequence, Format pattern.
2. `data` : sequence, The data that will be formatted,
3. `fn` : object, The file to receive the formatted output. Default is to the STDOUT device (console).
4. `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](#).

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",
```

See Also:

[text:format](#), [writef](#), [write_lines](#)

2.0.0.759 xor_bits

```
<built-in> function xor_bits(object a, object b)
```

Perform the logical XOR operation on corresponding bits in two objects. A bit in the result will be 1 only if the corresponding bits in both arguments are different.

Parameters:

1. a : one of the objects involved
2. b : the second object

Returns:

An **object**, whose shape depends on the shape of both arguments. Each atom in this object is obtained by logical 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.

Example 1:

```
a = xor_bits(#0110, #1010)
-- a is #1100
```

See Also:

[and_bits](#), [or_bits](#), [not_bits](#), [int_to_bits](#)

2.0.0.760 years_day

```
include std/datetime.e
public function years_day(datetime dt)
```

Parameters:

Get the Julian day of year of the supplied date.

Parameters:

1. 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
```

safe.e This file is not normally included directly. The normal approach is to include `std/machine.e`, which will automatically include either this file or `std/safe.e` if the `SAFE` symbol has been defined.

Warning: Some of these routines require a knowledge of machine-level programming. You could crash your system!

These routines, along with `peek()`, `poke()` and `call()`, let you access all of the features of your computer. You can read and write to any memory location, and you can create and execute machine code subroutines.

If you are manipulating 32-bit addresses or values, remember to use variables declared as `atom`. The integer type only goes up to 31 bits.

Writing characters to screen memory with `poke()` is much faster than using `puts()`. Address of start of text screen memory:

- mono: #B0000
- color: #B8000

If you choose to call `machine_proc()` or `machine_func()` directly (to save a bit of overhead) you *must* pass valid arguments or Euphoria could crash.

Some example programs to look at:

- `demo/callmach.exe` -- calling a machine language routine

See also `include/safe.e`. It's a safe, debugging version of this file.

These constant names are taken right from Microsoft's Memory Protection constants.

1 Constants

2 Routines

Subject and Routine Index

O A B C D E F G H
I J K L M N O P Q
R S T U V W X Y

0

0.0.0.1 ? ()

A

A_EXECUTE (Constants)

A_WRITE (Constants)

abort (Routines)

abs (Routines)

absolute_path (Routines)

accept (Routines)

add (Routines)

ADD (Constants)

ADD_APPEND (Constants)

add_item (Routines)

ADD_PREPEND (Constants)

ADD_SORT_DOWN (Constants)

ADD_SORT_UP (Constants)

add_to (Routines)

ADDR_ADDRESS (Constants)

ADDR_FAMILY (Constants)

ADDR_FLAGS (Constants)

ADDR_PROTOCOL (Constants)

ADDR_TYPE (Constants)

ADLER32 (Constants)

AF_APPLETALK (Constants)

AF_BTH (Constants)

AF_INET (Constants)

AF_INET6 (Constants)

AF_UNIX (Constants)

AF_UNSPEC (Constants)

all_copyrights (Routines)

all_left_units (Routines)

all_matches (Routines)

all_right_units (Routines)

MB_APPLMODAL (Constants)

MB_DEFAULT_DESKTOP_ONLY (Constants)

MB_DEFBUTTON1 (Constants)

MB_DEFBUTTON2 (Constants)

MB_DEFBUTTON3 (Constants)

MB_DEFBUTTON4 (Constants)

MB_HELP (Constants)

MB_ICONASTERISK (Constants)

MB_ICONERROR (Constants)

MB_ICONEXCLAMATION (Constants)

MB_ICONHAND (Constants)

MB_ICONINFORMATION (Constants)

MB_ICONQUESTION (Constants)

MB_ICONSTOP (Constants)

MB_ICONWARNING (Constants)

MB_OK (Constants)

MB_OKCANCEL (Constants)

MB_RETRYCANCEL (Constants)

MB_RIGHT (Constants)

MB_RTLREADING (Constants)

MB_SERVICE_NOTIFICATION (Constants)

MB_SETFOREGROUND (Constants)

MB_SYSTEMMODAL (Constants)

MB_TASKMODAL (Constants)

MB_YESNO (Constants)

MB_YESNOCANCEL (Constants)

MD5 (Constants)

median (Routines)

MEM_COMMIT (Constants)

mem_copy (Routines)

mem_copy (Routines)

MEM_RELEASE (Constants)

MEM_RESERVE (Constants)

allocate (Routines)
allocate_code (Routines)
allocate_data (Routines)
allocate_pointer_array (Routines)
allocate_protect (Routines)
allocate_string (Routines)
allocate_string_pointer_array (Routines)
allocate_wstring (Routines)
allocations (Routines)
allow_break (Routines)
amalgamated_sum (Routines)
ampm (Routines)
ANCHORED (Constants)
and_bits (Routines)
any_key (Routines)
ANY_UP (Constants)
append (Routines)
APPEND (Constants)
append_lines (Routines)
apply (Routines)
approx (Routines)
arccos (Routines)
arccosh (Routines)
arcsin (Routines)
arcsinh (Routines)
arctan (Routines)
arctanh (Routines)
ASCENDING (Constants)
ascii_string (Routines)
assert (Routines)
at (Routines)
AT_EXPANSION (Constants)
atan2 (Routines)
atom (Routines)
atom_to_float32 (Routines)
atom_to_float64 (Routines)
attr_to_colors (Routines)
AUTO_CALLOUT (Constants)
avedev (Routines)
average (Routines)
MEM_RESET (Constants)
mem_set (Routines)
mem_set (Routines)
memDLL_id (Routines)
memory_used (Routines)
merge (Routines)
message_box (Routines)
mid (Routines)
MIDDLE_DOWN (Constants)
MIDDLE_UP (Constants)
min (Routines)
MIN_ASCII (Constants)
MINF (Constants)
minsize (Routines)
MISSING_END (Constants)
mixture (Routines)
mlock (Routines)
mmap (Routines)
mod (Routines)
mode (Routines)
money (Routines)
month_abbrs (Routines)
month_names (Routines)
mouse_events (Routines)
mouse_pointer (Routines)
movavg (Routines)
MOVE (Constants)
move_file (Routines)
mprotect (Routines)
MSG_CONFIRM (Constants)
MSG_CTRUNC (Constants)
MSG_DONTROUTE (Constants)
MSG_DONTWAIT (Constants)
MSG_EOR (Constants)
MSG_ERRQUEUE (Constants)
MSG_FIN (Constants)
MSG_MORE (Constants)
MSG_NOSIGNAL (Constants)
MSG_OOB (Constants)
MSG_PEEK (Constants)
MSG_PROXY (Constants)

B

BAD_FILE (Constants)
 BAD_RECNO (Constants)
 BAD_SEEK (Constants)
 begins (Routines)
 belongs_to (Routines)
 BINARY_MODE (Constants)
 binary_search (Routines)
 bind (Routines)
 bits_to_int (Routines)
 bk_color (Routines)
 BK_LEN (Constants)

 BK_PIECES (Constants)

 BLACK (Constants)
 BLINKING (Constants)
 Block of key pointers (Routines)
 Block of table headers (Routines)
 BLOCK_CURSOR (Constants)
 BLUE (Constants)
 BMP_INVALID_MODE (Constants)
 BMP_OPEN_FAILED (Constants)
 BMP_SUCCESS (Constants)
 BMP_UNEXPECTED_EOF (Constants)
 BMP_UNSUPPORTED_FORMAT (Constants)
 boolean (Routines)
 BORDER_SPACE (Constants)
 BORDER_SPACE (Constants)
 bordered_address (Routines)
 bordered_address (Routines)
 breakup (Routines)
 BRIGHT_BLUE (Constants)
 BRIGHT_CYAN (Constants)
 BRIGHT_GREEN (Constants)
 BRIGHT_MAGENTA (Constants)
 BRIGHT_RED (Constants)
 BRIGHT_WHITE (Constants)
 BROWN (Constants)
 BSR_ANYCRLF (Constants)
 BSR_UNICODE (Constants)

MSG_RST (Constants)
 MSG_SYN (Constants)
 MSG_TRUNC (Constants)
 MSG_TRYHARD (Constants)
 MSG_WAITALL (Constants)
 MULTILINE (Constants)
 MULTIPLE (Constants)
 MULTIPLY (Constants)
 munlock (Routines)
 munmap (Routines)
 my_dir (Routines)

N

NESTED_ALL (Constants)
 NESTED_ANY (Constants)
 NESTED_BACKWARD (Constants)
 nested_get (Routines)
 NESTED_INDEX (Constants)
 nested_put (Routines)
 NETBSD (Constants)
 new (Routines)
 new (Routines)
 new (Routines)
 new (Routines)
 new (Routines)
 new_extra (Routines)
 new_from_kvpairs (Routines)
 new_from_string (Routines)
 new_time (Routines)
 NEWLINE_ANY (Constants)
 NEWLINE_ANYCRLF (Constants)
 NEWLINE_CR (Constants)
 NEWLINE_CRLF (Constants)
 NEWLINE_LF (Constants)
 next_prime (Routines)
 NO_AT_EXPANSION (Constants)
 NO_AUTO_CAPTURE (Constants)
 NO_CASE (Constants)
 NO_CURSOR (Constants)

build_commandline (Routines)
 build_list (Routines)
 builtins (Routines)
 byte_range (Routines)
 BYTES_PER_CHAR (Constants)
 BYTES_PER_SECTOR (Constants)
 bytes_to_int (Routines)

C

C_BOOL (Constants)
 C_BYTE (Constants)
 C_CHAR (Constants)
 C_DOUBLE (Constants)
 C_DWORD (Constants)
 C_DWORDLONG (Constants)
 C_FLOAT (Constants)
 c_func (Routines)
 c_func (Routines)
 C_HANDLE (Constants)
 C_HRESULT (Constants)
 C_HWND (Constants)
 C_INT (Constants)
 C_LONG (Constants)
 C_LPARAM (Constants)
 C_POINTER (Constants)
 c_proc (Routines)
 c_proc (Routines)
 C_SHORT (Constants)
 C_SIZE_T (Constants)
 C_UBYTE (Constants)
 C_UCHAR (Constants)
 C_UINT (Constants)
 C_ULONG (Constants)
 C_USHORT (Constants)
 C_WORD (Constants)
 C_WPARAM (Constants)

calc_hash (Routines)
 calc_primes (Routines)
 call (Routines)
 call (Routines)

NO_DATABASE (Constants)
 NO_HELP (Constants)
 NO_PARAMETER (Constants)
 NO_ROUTINE_ID (Constants)
 NO_TABLE (Constants)
 NO_UTF8_CHECK (Constants)
 NO_VALIDATION (Constants)
 NO_VALIDATION_AFTER_FIRST_EXTRA (Constants)
 NORMAL_ORDER (Constants)
 not_bits (Routines)
 NOTBOL (Constants)
 NOTEMPTY (Constants)
 NOTEOL (Constants)
 Notes (Routines)
 now (Routines)
 now_gmt (Routines)
 NS_C_ANY (Constants)
 NS_C_IN (Constants)
 NS_KT_DH (Constants)
 NS_KT_DSA (Constants)
 NS_KT_PRIVATE (Constants)
 NS_KT_RSA (Constants)
 NS_T_A (Constants)
 NS_T_A6 (Constants)
 NS_T_AAAA (Constants)
 NS_T_ANY (Constants)
 NS_T_MX (Constants)
 NS_T_NS (Constants)
 NS_T_PTR (Constants)
 NULL (Constants)
 NULLDEVICE (Constants)
 NUM_ENTRIES (Constants)
 number (Routines)
 number_array (Routines)
 NUMBER_OF_FREE_CLUSTERS (Constants)

O

OBJ_ATOM (Constants)
 OBJ_INTEGER (Constants)
 OBJ_SEQUENCE (Constants)

call_back (Routines)
 call_func (Routines)
 call_proc (Routines)
 can_add (Routines)
 canon2win (Routines)
 canonical (Routines)
 canonical_path (Routines)
 cardinal (Routines)
 CASELESS (Constants)
 ceil (Routines)
 central_moment (Routines)
 chance (Routines)
 change_target (Routines)
 char_test (Routines)
 chdir (Routines)
 check_all_blocks (Routines)
 check_all_blocks (Routines)
 check_break (Routines)
 check_calls (Routines)
 check_calls (Routines)

 check_free_list (Routines)

 checksum (Routines)
 CHILD (Constants)
 clear (Routines)
 clear (Routines)
 clear_directory (Routines)
 clear_screen (Routines)
 close (Routines)
 close (Routines)
 close (Routines)
 cmd_parse (Routines)
 CMD_SWITCHES (Constants)
 color (Routines)
 Colors (Constants)
 colors_to_attr (Routines)
 columnize (Routines)
 combine (Routines)
 combine_maps (Routines)
 COMBINE_SORTED (Constants)
 COMBINE_UNSORTED (Constants)

 OBJ_UNASSIGNED (Constants)
 object (Routines)
 OK (Constants)
 ONCE (Constants)
 open (Routines)
 open_dll (Routines)
 OPENBSD (Constants)
 operation (Routines)
 OPT_CNT (Constants)
 OPT_IDX (Constants)
 OPT_REV (Constants)
 OPT_VAL (Constants)
 optimize (Routines)
 option_spec (Routines)
 option_spec_to_string (Routines)
 option_switches (Routines)
 OPTIONAL (Constants)
 or_all (Routines)
 or_bits (Routines)
 OSX (Constants)

P
 pad_head (Routines)
 pad_tail (Routines)
 page_aligned_address (Routines)
 PAGE_EXECUTE (Constants)
 PAGE_EXECUTE_READ (Constants)
 PAGE_EXECUTE_READWRITE (Constants)
 PAGE_EXECUTE_WRITECOPY (Constants)
 PAGE_NOACCESS (Constants)
 PAGE_NONE (Constants)
 PAGE_READ (Constants)
 PAGE_READ_EXECUTE (Constants)
 PAGE_READ_WRITE (Constants)
 PAGE_READ_WRITE_EXECUTE (Constants)
 PAGE_READONLY (Constants)
 PAGE_READWRITE (Constants)
 PAGE_SIZE (Constants)
 PAGE_WRITE_COPY (Constants)
 PAGE_WRITE_EXECUTE_COPY (Constants)
 PAGE_WRITECOPY (Constants)

command_line (Routines)
 compare (Routines)
 compare (Routines)
 Compile Time and Match Time (Routines)
 compose_map (Routines)
 CONCAT (Constants)
 connect (Routines)
 Constants
 copy (Routines)
 copy_file (Routines)
 cos (Routines)
 cosh (Routines)
 count (Routines)
 COUNT_DIRS (Constants)
 COUNT_FILES (Constants)
 COUNT_SIZE (Constants)
 COUNT_TYPES (Constants)
 crash (Routines)
 crash_file (Routines)
 crash_message (Routines)
 crash_routine (Routines)
 create (Routines)
 create (Routines)
 create_directory (Routines)
 create_file (Routines)
 CS_FIRST (Constants)
 curdir (Routines)
 current_dir (Routines)
 cursor (Routines)
 custom_sort (Routines)
 CYAN (Constants)

D

D_ALTNAME (Constants)
 D_ATTRIBUTES (Constants)
 D_DAY (Constants)
 D_HOUR (Constants)
 D_MILLISECOND (Constants)
 D_MINUTE (Constants)
 D_MONTH (Constants)
 D_NAME (Constants)

pairs (Routines)
 PARENT (Constants)
 parse (Routines)
 parse (Routines)
 parse_commandline (Routines)
 parse_ip_address (Routines)
 parse_querystring (Routines)
 parse_recvheader (Routines)
 parse_url (Routines)
 PARTIAL (Constants)
 patch (Routines)
 PATH_BASENAME (Constants)
 PATH_DIR (Constants)
 PATH_DRIVEID (Constants)
 PATH_FILEEXT (Constants)
 PATH_FILENAME (Constants)
 pathinfo (Routines)
 pathname (Routines)
 PATHSEP (Constants)
 PAUSE_MSG (Constants)
 pcre_copyright (Routines)
 peek (Routines)
 peek (Routines)
 peek2s (Routines)
 peek2s (Routines)
 peek2u (Routines)
 peek2u (Routines)
 peek4s (Routines)
 peek4s (Routines)
 peek4u (Routines)
 peek4u (Routines)

peek_end (Routines)
 peek_string (Routines)
 peek_string (Routines)
 peek_top (Routines)
 peek_wstring (Routines)
 peeks (Routines)
 peeks (Routines)
 PHI (Constants)
 PI (Constants)

D_SECOND (Constants)
D_SIZE (Constants)
D_YEAR (Constants)
date (Routines)
datetime (Routines)
datetime (Routines)
day_abbrs (Routines)
day_names (Routines)
days_in_month (Routines)
days_in_year (Routines)
db_cache_clear (Routines)
db_clear_table (Routines)
db_close (Routines)
db_compress (Routines)
db_create (Routines)
db_create_table (Routines)
db_current (Routines)
db_current_table (Routines)
db_delete_record (Routines)
db_delete_table (Routines)
db_dump (Routines)
DB_EXISTS_ALREADY (Constants)
DB_FATAL_FAIL (Constants)
db_fatal_id (Routines)
db_fetch_record (Routines)
db_find_key (Routines)
db_get_errors (Routines)
db_get_recid (Routines)
db_insert (Routines)
DB_LOCK_EXCLUSIVE (Constants)
DB_LOCK_FAIL (Constants)
DB_LOCK_NO (Constants)
DB_LOCK_SHARED (Constants)
DB_OK (Constants)
db_open (Routines)
DB_OPEN_FAIL (Constants)
db_record_data (Routines)
db_record_key (Routines)
db_record_recid (Routines)
db_rename_table (Routines)
db_replace_data (Routines)
PID (Constants)
PINF (Constants)
PISQR (Constants)
pivot (Routines)
platform (Routines)
platform_locale (Routines)
platform_name (Routines)
poke (Routines)
poke (Routines)
poke2 (Routines)
poke2 (Routines)
poke4 (Routines)
poke4 (Routines)
poke_string (Routines)
poke_wstring (Routines)
pop (Routines)
position (Routines)
positive_int (Routines)
positive_int (Routines)
positive_int (Routines)
posix_names (Routines)
power (Routines)
powof2 (Routines)
prepare_block (Routines)
prepare_block (Routines)
prepend (Routines)
PRETTY_DEFAULT (Constants)
pretty_print (Routines)
pretty_sprint (Routines)
prime_list (Routines)
print (Routines)
printf (Routines)
process (Routines)
process_lines (Routines)
product (Routines)
product (Routines)
product_map (Routines)
project (Routines)
prompt_number (Routines)
prompt_string (Routines)
proper (Routines)

db_replace_recid (Routines)
 db_select (Routines)
 db_select_table (Routines)
 db_set_caching (Routines)
 db_table_list (Routines)
 db_table_size (Routines)
 deallocate (Routines)
 deallocate (Routines)
 decanonical (Routines)
 decode (Routines)
 DEFAULT (Constants)

 defaulted_value (Routines)

 defaulttext (Routines)
 define_c_func (Routines)

 define_c_proc (Routines)

 define_c_var (Routines)
 define_map (Routines)
 define_operation (Routines)
 deg2rad (Routines)
 DEGREES_TO_RADIANS (Constants)
 delete (Routines)
 delete_file (Routines)
 delete_routine (Routines)
 delta (Routines)
 DEP_on (Constants)
 DEP_really_works (Constants)
 dep_works (Routines)
 dep_works (Routines)
 dequote (Routines)
 DESCENDING (Constants)
 deserialize (Routines)
 DFA_RESTART (Constants)
 DFA_SHORTEST (Constants)
 diagram_commutates (Routines)
 diff (Routines)
 difference (Routines)
 dir (Routines)
 dir_size (Routines)
 direct_map (Routines)

PROT_EXEC (Constants)
 PROT_NONE (Constants)
 PROT_READ (Constants)
 PROT_WRITE (Constants)
 push (Routines)
 PUT (Constants)
 put (Routines)
 put_integer16 (Routines)
 put_integer32 (Routines)
 put_screen_char (Routines)
 puts (Routines)

Q

QUARTPI (Constants)
 quote (Routines)

R

rad2deg (Routines)
 RADIANS_TO_DEGREES (Constants)
 ram_space (Routines)
 rand (Routines)
 rand_range (Routines)
 range (Routines)
 range (Routines)
 raw_frequency (Routines)
 RD_INPLACE (Constants)
 read (Routines)
 read_bitmap (Routines)
 read_file (Routines)
 read_lines (Routines)
 receive (Routines)
 receive_from (Routines)
 RED (Constants)
 regex (Routines)
 register_block (Routines)
 register_block (Routines)
 rehash (Routines)
 remainder (Routines)
 remove (Routines)
 remove (Routines)
 remove_all (Routines)

dirname (Routines)
 disk_metrics (Routines)
 disk_size (Routines)
 display (Routines)
 DISPLAY_ASCII (Constants)
 display_text_image (Routines)
 distributes_over (Routines)
 DIVIDE (Constants)
 DNS_QUERY_ACCEPT_TRUNCATED_RESPONSE (Constants)
 DNS_QUERY_BYPASS_CACHE (Constants)
 DNS_QUERY_DONT_RESET_TTL_VALUES (Constants)
 DNS_QUERY_NO_HOSTS_FILE (Constants)
 DNS_QUERY_NO_LOCAL_NAME (Constants)
 DNS_QUERY_NO_NETBT (Constants)
 DNS_QUERY_NO_RECURSION (Constants)
 DNS_QUERY_NO_WIRE_QUERY (Constants)
 DNS_QUERY_RESERVED (Constants)
 DNS_QUERY_RETURN_MESSAGE (Constants)
 DNS_QUERY_STANDARD (Constants)
 DNS_QUERY_TREAT_AS_FQDN (Constants)
 DNS_QUERY_USE_TCP_ONLY (Constants)
 DNS_QUERY_WIRE_ONLY (Constants)
 DOLLAR_ENDONLY (Constants)
 DOS_TEXT (Constants)
 DOTALL (Constants)
 driveid (Routines)
 dump (Routines)
 dup (Routines)
 DUP_TABLE (Constants)
 DUPNAMES (Constants)

E

E (Constants)
 E_ATOM (Constants)
 E_INTEGER (Constants)
 E_OBJECT (Constants)
 E_SEQUENCE (Constants)
 edges_only (Routines)
 edges_only (Routines)

remove_directory (Routines)
 remove_dups (Routines)
 remove_from (Routines)
 remove_item (Routines)
 remove_subseq (Routines)
 rename_file (Routines)
 repeat (Routines)
 repeat_pattern (Routines)
 replace (Routines)
 replace_all (Routines)
 restrict (Routines)
 retain_all (Routines)
 reverse (Routines)
 reverse_map (Routines)
 REVERSE_ORDER (Constants)
 rfind (Routines)
 RIGHT_DOWN (Constants)
 RIGHT_UP (Constants)
 rmatch (Routines)
 rnd (Routines)
 rnd_1 (Routines)
 roll (Routines)
 rotate (Routines)
 rotate_bits (Routines)
 ROTATE_LEFT (Constants)
 ROTATE_RIGHT (Constants)
 round (Routines)
 routine_id (Routines)
 Routines

S

safe_address (Routines)
 safe_address (Routines)
 safe_address_list (Routines)
 sample (Routines)
 save_bitmap (Routines)
 save_map (Routines)
 save_text_image (Routines)
 SCREEN (Constants)

embed_union (Routines)
embedding (Routines)
emovavg (Routines)
encode (Routines)
ends (Routines)
ensure_in_list (Routines)
ensure_in_range (Routines)
EOF (Constants)
EOL (Constants)
EOLSEP (Constants)
equal (Routines)
ERR_ACCESS (Constants)
ERR_ADDRINUSE (Constants)
ERR_ADDRNOTAVAIL (Constants)
ERR_AFNOSUPPORT (Constants)
ERR_AGAIN (Constants)
ERR_ALREADY (Constants)
ERR_CLOSE_CHAR (Constants)
ERR_CONNABORTED (Constants)
ERR_CONNREFUSED (Constants)
ERR_CONNRESET (Constants)
ERR_DECIMAL (Constants)
ERR_DESTADDRREQ (Constants)
ERR_EOF (Constants)
ERR_EOF_STRING (Constants)
ERR_EOL_CHAR (Constants)
ERR_EOL_STRING (Constants)
ERR_ESCAPE (Constants)
ERR_FAULT (Constants)
ERR_HEX (Constants)
ERR_HEX_STRING (Constants)
ERR_HOSTUNREACH (Constants)
ERR_INPROGRESS (Constants)
ERR_INTR (Constants)
ERR_INVALID (Constants)
ERR_IO (Constants)
ERR_ISCONN (Constants)
ERR_ISDIR (Constants)
ERR_LOOP (Constants)
ERR_MFILE (Constants)
ERR_MSGSIZE (Constants)
scroll (Routines)
SD_BOTH (Constants)
SD_RECEIVE (Constants)
SD_SEND (Constants)
section (Routines)
SECTORS_PER_CLUSTER (Constants)
seek (Routines)
select (Routines)
SELECT_IS_ERROR (Constants)
SELECT_IS_READABLE (Constants)
SELECT_IS_WRITABLE (Constants)
SELECT_SOCKET (Constants)
send (Routines)
send_to (Routines)
SEQ_NOALT (Constants)
sequence (Routines)
sequence_array (Routines)
sequence_to_set (Routines)
sequences_to_map (Routines)
serialize (Routines)
service_by_name (Routines)
service_by_port (Routines)
set (Routines)
set (Routines)
set (Routines)
set_accumulate_summary (Routines)
set_charsets (Routines)
set_colors (Routines)
set_decimal_mark (Routines)
set_def_lang (Routines)
set_default_charsets (Routines)
set_encoding_properties (Routines)
set_lang_path (Routines)
set_option (Routines)
set_rand (Routines)
set_sendheader (Routines)
set_sendheader_default (Routines)
set_sendheader_useragent_msie (Routines)
set_test_abort (Routines)
set_test_verbosity (Routines)
set_wait_on_summary (Routines)

ERR_NAMETOOLONG (Constants)
 ERR_NETDOWN (Constants)
 ERR_NETRESET (Constants)
 ERR_NETUNREACH (Constants)
 ERR_NFILE (Constants)
 ERR_NOBUFS (Constants)
 ERR_NOENT (Constants)
 ERR_NOTCONN (Constants)
 ERR_NOTDIR (Constants)
 ERR_NOTINITIALISED (Constants)
 ERR_NOTSOCK (Constants)
 ERR_OPEN (Constants)
 ERR_OPNOTSUPP (Constants)
 ERR_PROTONOSUPPORT (Constants)
 ERR_PROTOTYPE (Constants)
 ERR_ROFS (Constants)
 ERR_SHUTDOWN (Constants)
 ERR_SOCKETNOSUPPORT (Constants)
 ERR_TIMEDOUT (Constants)
 ERR_UNKNOWN (Constants)
 ERR_WOULDBLOCK (Constants)
 ERROR_BADCOUNT (Constants)
 ERROR_BADMAGIC (Constants)
 ERROR_BADNEWLINE (Constants)
 ERROR_BADOPTION (Constants)
 ERROR_BADPARTIAL (Constants)
 ERROR_BADUTF8 (Constants)
 ERROR_BADUTF8_OFFSET (Constants)
 ERROR_CALLOUT (Constants)
 error_code (Routines)
 ERROR_DFA_RECURSE (Constants)
 ERROR_DFA_UCOND (Constants)
 ERROR_DFA_UITEM (Constants)
 ERROR_DFA_UMLIMIT (Constants)
 ERROR_DFA_WSSIZE (Constants)
 ERROR_INTERNAL (Constants)
 ERROR_MATCHLIMIT (Constants)
 error_message (Routines)
 error_names (Routines)
 error_no (Routines)
 ERROR_NOMATCH (Constants)
 setenv (Routines)
 SHA256 (Constants)
 SHARED_LIB_EXT (Constants)
 shift_bits (Routines)
 show_block (Routines)
 show_help (Routines)
 SHOW_ONLY_OPTIONS (Constants)
 shuffle (Routines)
 shutdown (Routines)
 SIDE_NONE (Constants)
 sign (Routines)
 sim_index (Routines)
 sin (Routines)
 sinh (Routines)
 size (Routines)
 size (Routines)
 skewness (Routines)
 SLASH (Constants)
 SLASHES (Constants)
 sleep (Routines)
 slice (Routines)
 SM_TEXT (Constants)
 small (Routines)
 smallest (Routines)
 SMALLMAP (Constants)
 SND_ASTERISK (Constants)
 SND_DEFAULT (Constants)
 SND_EXCLAMATION (Constants)
 SND_QUESTION (Constants)
 SND_STOP (Constants)
 SO_ACCEPTCONN (Constants)
 SO_BINDTODEVICE (Constants)
 SO_BROADCAST (Constants)
 SO_CONNDATA (Constants)
 SO_CONNDATALEN (Constants)
 SO_CONNOPT (Constants)
 SO_CONNOPTLEN (Constants)
 SO_DEBUG (Constants)
 SO_DISCDATA (Constants)
 SO_DISCDATALEN (Constants)
 SO_DISCOPT (Constants)

ERROR_NOMEMORY (Constants)
 ERROR_NOSUBSTRING (Constants)
 ERROR_NULL (Constants)
 ERROR_NULLWSLIMIT (Constants)
 ERROR_PARTIAL (Constants)
 ERROR_RECURSIONLIMIT (Constants)
 error_to_string (Routines)
 ERROR_UNKNOWN_NODE (Constants)
 ERROR_UNKNOWN_OPCODE (Constants)
 escape (Routines)
 escape (Routines)
 ET_ERR_COLUMN (Constants)
 ET_ERR_LINE (Constants)
 ET_ERROR (Constants)
 et_error_string (Routines)
 et_keep_blanks (Routines)
 et_keep_comments (Routines)
 et_string_numbers (Routines)

 et_tokenize_file (Routines)

 et_tokenize_string (Routines)
 ET_TOKENS (Constants)
 EULER_GAMMA (Constants)
 euphoria_copyright (Routines)
 exec (Routines)
 exp (Routines)
 EXT_COUNT (Constants)
 EXT_NAME (Constants)
 EXT_SIZE (Constants)
 EXTENDED (Constants)
 EXTRA (Constants)
 extract (Routines)

F
 FALSE (Constants)
 fetch (Routines)
 fib (Routines)
 fiber_over (Routines)
 fiber_product (Routines)
 FIFO (Constants)

 SO_DISCOPTLEN (Constants)
 SO_DONTLINGER (Constants)
 SO_DONTROUTE (Constants)
 SO_ERROR (Constants)
 SO_KEEPALIVE (Constants)
 SO_LINGER (Constants)
 SO_MAXDG (Constants)
 SO_MAXPATHDG (Constants)
 SO_OOBLINE (Constants)
 SO_OPENTYPE (Constants)
 SO_PASSCRED (Constants)
 SO_PEERCREC (Constants)
 SO_RCVBUF (Constants)
 SO_RCVLOWAT (Constants)
 SO_RCVTIMEO (Constants)
 SO_REUSEADDR (Constants)
 SO_REUSEPORT (Constants)
 SO_SECURITY_AUTHENTICATION (Constants)
 SO_SECURITY_ENCRYPTION_NETWORK (Constants)
 SO_SECURITY_ENCRYPTION_TRANSPORT (Constants)
 SO_SNDBUF (Constants)
 SO_SNDLOWAT (Constants)
 SO_SNDTIMEO (Constants)
 SO_SYNCHRONOUS_ALERT (Constants)
 SO_SYNCHRONOUS_NONALERT (Constants)
 SO_TYPE (Constants)
 SO_USELOOPBACK (Constants)
 SOCK_DGRAM (Constants)
 SOCK_RAW (Constants)
 SOCK_RDM (Constants)
 SOCK_SEQPACKET (Constants)

 SOCK_STREAM (Constants)
 socket (Routines)
 SOCKET_SOCKETADDR_IN (Constants)
 SOCKET_SOCKET (Constants)
 SOL_SOCKET (Constants)
 sort (Routines)
 sort_columns (Routines)

file_exists (Routines)	splice (Routines)
file_length (Routines)	split (Routines)
file_number (Routines)	split (Routines)
file_position (Routines)	split_any (Routines)
file_timestamp (Routines)	split_limit (Routines)
file_type (Routines)	sprint (Routines)
filebase (Routines)	sprintf (Routines)
fileext (Routines)	sqrt (Routines)
filename (Routines)	SQRT2 (Constants)
FILETYPE_DIRECTORY (Constants)	SQRT3 (Constants)
FILETYPE_FILE (Constants)	SQRT5 (Constants)
FILETYPE_NOT_FOUND (Constants)	SQRTE (Constants)
FILETYPE_UNDEFINED (Constants)	ST_ALLNUM (Constants)
filter (Routines)	ST_FULLPOP (Constants)
find (Routines)	ST_IGNSTR (Constants)
find (Routines)	ST_SAMPLE (Constants)
find_all (Routines)	ST_ZEROSTR (Constants)
find_all (Routines)	stack (Routines)
find_any (Routines)	START_COLUMN (Constants)
find_each (Routines)	statistics (Routines)
find_from (Routines)	std_library_address (Routines)
find_nested (Routines)	STDERR (Constants)
find_replace (Routines)	STDERR (Constants)
find_replace (Routines)	stdev (Routines)
find_replace_callback (Routines)	STDFLTR_ALPHA (Constants)
find_replace_limit (Routines)	STDIN (Constants)
FIRSTLINE (Constants)	STDIN (Constants)
flags_to_string (Routines)	STDOUT (Constants)
flatten (Routines)	STDOUT (Constants)
FLETCHER32 (Constants)	store (Routines)
float32_to_atom (Routines)	string (Routines)
float64_to_atom (Routines)	STRING_OFFSETS (Constants)
floor (Routines)	subsets (Routines)
flush (Routines)	SUBTRACT (Constants)
for_each (Routines)	subtract (Routines)
format (Routines)	sum (Routines)
format (Routines)	sum (Routines)
FP_FORMAT (Constants)	sum_central_moments (Routines)
frac (Routines)	SUNOS (Constants)
free (Routines)	swap (Routines)
free (Routines)	SyntaxColor (Constants)

Free list (Routines)

FREE_BYTES (Constants)

free_code (Routines)

free_code (Routines)

free_code (Routines)

free_console (Routines)

free_pointer_array (Routines)

FREE_RID (Constants)

FREEBSD (Constants)

from_date (Routines)

from_unix (Routines)

G

gcd (Routines)

geomean (Routines)

get (Routines)

get (Routines)

get (Routines)

get_active_id (Routines)

get_bytes (Routines)

get_charsets (Routines)

get_def_lang (Routines)

get_dstring (Routines)

get_encoding_properties (Routines)

GET_EOF (Constants)

GET_FAIL (Constants)

get_http (Routines)

get_http_use_cookie (Routines)

get_integer16 (Routines)

get_integer32 (Routines)

get_key (Routines)

get_key (Routines)

get_lang_path (Routines)

GET_LONG_ANSWER (Constants)

get_mouse (Routines)

GET_NOTHING (Constants)

get_option (Routines)

get_ovector_size (Routines)

get_page_size (Routines)

get_pid (Routines)

system (Routines)

system_exec (Routines)

T

t_alnum (Routines)

t_alpha (Routines)

t_ascii (Routines)

T_BLANK (Constants)

t_boolean (Routines)

t_bytearray (Routines)

T_CHAR (Constants)

t_cntrl (Routines)

T_COLON (Constants)

T_COMMA (Constants)

T_COMMENT (Constants)

T_CONCAT (Constants)

T_CONCATEQ (Constants)

t_consonant (Routines)

T_DELIMITER (Constants)

t_digit (Routines)

t_display (Routines)

T_DIVIDE (Constants)

T_DIVIDEEQ (Constants)

T_DOLLAR (Constants)

T_DOUBLE_OPS (Constants)

T_EOF (Constants)

T_EQ (Constants)

t_graph (Routines)

T_GT (Constants)

T_GTEQ (Constants)

t_identifier (Routines)

T_IDENTIFIER (Constants)

T_KEYWORD (Constants)

T_LBRACE (Constants)

T_LBRACKET (Constants)

t_lower (Routines)

T_LPAREN (Constants)

T_LT (Constants)

T_LTEQ (Constants)

T_MINUS (Constants)

[get_position \(Routines\)](#)
[get_recvheader \(Routines\)](#)
[get_screen_char \(Routines\)](#)
[get_sendheader \(Routines\)](#)
[GET_SHORT_ANSWER \(Constants\)](#)
[GET_SUCCESS \(Constants\)](#)
[get_text \(Routines\)](#)
[get_url \(Routines\)](#)
[getc \(Routines\)](#)
[getenv \(Routines\)](#)
[GetLastError_rid \(Constants\)](#)
[gets \(Routines\)](#)
[GetSystemInfo_rid \(Constants\)](#)
[graphics_mode \(Routines\)](#)
[graphics_point \(Routines\)](#)
[GRAY \(Constants\)](#)
[GREEN \(Constants\)](#)

H

[HALF_BLOCK_CURSOR \(Constants\)](#)
[HALFPI \(Constants\)](#)
[HALFSQRT2 \(Constants\)](#)
[harmean \(Routines\)](#)
[has \(Routines\)](#)
[HAS_CASE \(Constants\)](#)
[has_inverse \(Routines\)](#)
[has_match \(Routines\)](#)
[HAS_PARAMETER \(Constants\)](#)
[has_unit \(Routines\)](#)
[hash \(Routines\)](#)
[head \(Routines\)](#)
[Header \(Routines\)](#)
[HELP \(Constants\)](#)
[HELP_RID \(Constants\)](#)
[hex_text \(Routines\)](#)
[HOST_ALIASES \(Constants\)](#)
[host_by_addr \(Routines\)](#)
[host_by_name \(Routines\)](#)
[HOST_IPS \(Constants\)](#)
[HOST_OFFICIAL_NAME \(Constants\)](#)
[HOST_TYPE \(Constants\)](#)
[T_MINUSEQ \(Constants\)](#)
[T_MULTIPLY \(Constants\)](#)
[T_MULTIPLYEQ \(Constants\)](#)
[T_NOT \(Constants\)](#)
[T_NOTEQ \(Constants\)](#)
[T_NULL \(Constants\)](#)
[T_NUMBER \(Constants\)](#)
[T_PERIOD \(Constants\)](#)
[T_PLUS \(Constants\)](#)
[T_PLUSEQ \(Constants\)](#)
[t_print \(Routines\)](#)
[t_punct \(Routines\)](#)
[T_QPRINT \(Constants\)](#)
[T_RBRACE \(Constants\)](#)
[T_RBRACKET \(Constants\)](#)
[T_RPAREN \(Constants\)](#)
[T_SHBANG \(Constants\)](#)
[T_SINGLE_OPS \(Constants\)](#)
[T_SLICE \(Constants\)](#)
[t_space \(Routines\)](#)
[t_specword \(Routines\)](#)
[T_STRING \(Constants\)](#)
[t_text \(Routines\)](#)
[t_upper \(Routines\)](#)
[t_vowel \(Routines\)](#)
[t_xdigit \(Routines\)](#)
[Table header \(Routines\)](#)
[tail \(Routines\)](#)
[tan \(Routines\)](#)
[tanh \(Routines\)](#)
[task_clock_start \(Routines\)](#)
[task_clock_stop \(Routines\)](#)
[task_create \(Routines\)](#)
[task_delay \(Routines\)](#)
[task_list \(Routines\)](#)
[task_schedule \(Routines\)](#)
[task_self \(Routines\)](#)
[task_status \(Routines\)](#)
[task_suspend \(Routines\)](#)
[task_yield \(Routines\)](#)

HSIEH32 (Constants)
 HTTP_HEADER_ACCEPT (Constants)
 HTTP_HEADER_ACCEPTCHARSET (Constants)
 HTTP_HEADER_ACCEPTENCODING (Constants)
 HTTP_HEADER_ACCEPTLANGUAGE (Constants)
 HTTP_HEADER_ACCEPTRANGES (Constants)
 HTTP_HEADER_AUTHORIZATION (Constants)
 HTTP_HEADER_CACHECONTROL (Constants)
 HTTP_HEADER_CONNECTION (Constants)
 HTTP_HEADER_CONTENTLENGTH (Constants)
 HTTP_HEADER_CONTENTTYPE (Constants)
 HTTP_HEADER_DATE (Constants)
 HTTP_HEADER_FROM (Constants)
 HTTP_HEADER_GET (Constants)
 HTTP_HEADER_HOST (Constants)
 HTTP_HEADER_HTTPVERSION (Constants)
 HTTP_HEADER_IFMODIFIEDSINCE (Constants)
 HTTP_HEADER_KEEPALIVE (Constants)
 HTTP_HEADER_POST (Constants)
 HTTP_HEADER_POSTDATA (Constants)
 HTTP_HEADER_REFERER (Constants)
 HTTP_HEADER_USERAGENT (Constants)

I

IDABORT (Constants)
 IDCANCEL (Constants)
 IDIGNORE (Constants)
 IDNO (Constants)
 IDOK (Constants)
 IDRETRY (Constants)
 IDYES (Constants)
 iff (Routines)
 image (Routines)
 include_paths (Routines)
 INDENT (Constants)
 Index block (Routines)
 info (Routines)
 init_class (Routines)
 init_curdir (Routines)
 insert (Routines)
 INSERT_FAILED (Constants)

TDATA (Constants)
 temp_file (Routines)
 test_equal (Routines)
 test_exec (Routines)
 test_fail (Routines)
 test_false (Routines)
 test_not_equal (Routines)
 test_pass (Routines)
 TEST_QUIET (Constants)
 test_read (Routines)
 test_report (Routines)
 TEST_SHOW_ALL (Constants)
 TEST_SHOW_FAILED_ONLY (Constants)
 test_true (Routines)
 test_write (Routines)
 text_color (Routines)
 TEXT_MODE (Constants)
 text_rows (Routines)
 TF_ATOM (Constants)
 TF_HEX (Constants)
 TF_INT (Constants)
 TFORM (Constants)

THICK_UNDERLINE_CURSOR (Constants)
 threshold (Routines)
 time (Routines)
 TLNUM (Constants)
 TLPOS (Constants)
 to_integer (Routines)
 to_number (Routines)
 to_unix (Routines)
 top (Routines)
 TOTAL_BYTES (Constants)
 TOTAL_NUMBER_OF_CLUSTERS (Constants)
 trailer (Routines)
 trailer (Routines)
 transform (Routines)
 translate (Routines)
 transmute (Routines)
 trim (Routines)
 trim_head (Routines)

[insertion_sort \(Routines\)](#)
[instance \(Routines\)](#)
[INT_FORMAT \(Constants\)](#)
[int_to_bits \(Routines\)](#)
[int_to_bytes \(Routines\)](#)
[intdiv \(Routines\)](#)
[integer \(Routines\)](#)
[integer_array \(Routines\)](#)

[intersection \(Routines\)](#)

[INVALID_ROUTINE_ID \(Constants\)](#)
[INVLN10 \(Constants\)](#)
[INVLN2 \(Constants\)](#)
[INVSQ2PI \(Constants\)](#)
[is_associative \(Routines\)](#)
[is_bijective \(Routines\)](#)
[is_DEP_supported \(Routines\)](#)
[is_empty \(Routines\)](#)
[is_even \(Routines\)](#)
[is_even_obj \(Routines\)](#)
[is_in_list \(Routines\)](#)
[is_in_range \(Routines\)](#)
[is_inetaddr \(Routines\)](#)
[is_injective \(Routines\)](#)
[is_leap_year \(Routines\)](#)
[is_match \(Routines\)](#)
[is_match \(Routines\)](#)
[is_page_aligned_address \(Routines\)](#)
[is_subset \(Routines\)](#)
[is_surjective \(Routines\)](#)
[is_symmetric \(Routines\)](#)
[is_unit \(Routines\)](#)
[is_using_DEP \(Routines\)](#)
[is_valid_memory_protection_constant \(Routines\)](#)
[is_win_nt \(Routines\)](#)

J

[join \(Routines\)](#)

K

[kernel_dll \(Routines\)](#)

[trim_tail \(Routines\)](#)
[trsprintf \(Routines\)](#)
[TRUE \(Constants\)](#)
[true_color \(Routines\)](#)
[trunc \(Routines\)](#)
[TTYTYPE \(Constants\)](#)
[TWOPI \(Constants\)](#)
[type_of \(Routines\)](#)

U

[uname \(Routines\)](#)
[UNDERLINE_CURSOR \(Constants\)](#)
[UNGREEDY \(Constants\)](#)
[union \(Routines\)](#)
[UNIX_TEXT \(Constants\)](#)
[unlock_file \(Routines\)](#)
[unregister_block \(Routines\)](#)
[unregister_block \(Routines\)](#)
[unsetenv \(Routines\)](#)
[upper \(Routines\)](#)
[URL_ENTIRE \(Constants\)](#)
[URL_HOSTNAME \(Constants\)](#)
[URL_HTTP_DOMAIN \(Constants\)](#)
[URL_HTTP_PATH \(Constants\)](#)
[URL_HTTP_QUERY \(Constants\)](#)
[URL_MAIL_ADDRESS \(Constants\)](#)
[URL_MAIL_DOMAIN \(Constants\)](#)
[URL_MAIL_QUERY \(Constants\)](#)
[URL_MAIL_USER \(Constants\)](#)
[URL_PASSWORD \(Constants\)](#)
[URL_PATH \(Constants\)](#)
[URL_PORT \(Constants\)](#)
[URL_PROTOCOL \(Constants\)](#)
[URL_PROTOCOL \(Constants\)](#)
[URL_QUERY_STRING \(Constants\)](#)

[URL_USER \(Constants\)](#)

[USED_BYTES \(Constants\)](#)

[UTF8 \(Constants\)](#)

V

keys (Routines)
 keyvalues (Routines)
 keywords (Routines)
 kill (Routines)
 kurtosis (Routines)

L

lang_load (Routines)
 largest (Routines)
 last (Routines)
 LAST_ERROR_CODE (Constants)
 LEAVE (Constants)
 LEFT_DOWN (Constants)
 LEFT_UP (Constants)
 length (Routines)
 LINE_BREAKS (Constants)
 linear (Routines)
 LINUX (Constants)
 listen (Routines)
 LN10 (Constants)
 LN2 (Constants)
 load (Routines)
 load_map (Routines)
 locale_canonical (Routines)
 locate_file (Routines)
 LOCK_EXCLUSIVE (Constants)
 lock_file (Routines)
 LOCK_SHARED (Constants)
 lock_type (Routines)
 log (Routines)
 log10 (Routines)
 lookup (Routines)
 lower (Routines)

M

M_ALLOC (Constants)
 M_FREE (Constants)
 machine_addr (Routines)
 machine_addr (Routines)
 machine_func (Routines)

valid (Routines)
 valid_index (Routines)
 valid_memory_protection_constant (Routines)
 valid_memory_protection_constant (Routines)
 valid_wordsize (Routines)

VALIDATE_ALL (Constants)

value (Routines)
 values (Routines)
 VC_COLOR (Constants)
 VC_COLUMNS (Constants)
 VC_LINES (Constants)
 VC_MODE (Constants)
 VC_NCOLORS (Constants)
 VC_PAGES (Constants)
 VC_SCRNCOLS (Constants)
 VC_SCRNLINES (Constants)
 VC_XPIXELS (Constants)
 VC_YPIXELS (Constants)
 version (Routines)
 version_major (Routines)
 version_minor (Routines)
 version_patch (Routines)
 version_revision (Routines)
 version_string (Routines)
 version_string_long (Routines)
 version_string_short (Routines)
 version_type (Routines)
 video_config (Routines)
 VirtualAlloc_rid (Constants)
 VirtualFree_rid (Constants)
 VirtualFree_rid (Constants)
 VirtualLock_rid (Constants)
 VirtualProtect_rid (Constants)
 VirtualUnlock_rid (Constants)
 vlookup (Routines)
 vslice (Routines)

W

w32_name_canonical (Routines)

machine_proc (Routines)
MAGENTA (Constants)
malloc (Routines)
Managing Records (Routines)
Managing tables (Routines)
MANDATORY (Constants)
map (Routines)
map (Routines)
MAP_ANONYMOUS (Constants)
MAP_FILE (Constants)
MAP_FIXED (Constants)
MAP_PRIVATE (Constants)
MAP_SHARED (Constants)
MAP_TYPE (Constants)
mapping (Routines)
match (Routines)
match_all (Routines)
match_any (Routines)
match_from (Routines)

match_replace (Routines)
matches (Routines)
max (Routines)
MAX_ASCII (Constants)
MAX_LINES (Constants)
maybe_any_key (Routines)
MB_ABORTRETRYIGNORE (Constants)

w32_names (Routines)
W_BAD_PATH (Constants)
wait_key (Routines)
walk_dir (Routines)
warning (Routines)
warning_file (Routines)
weeks_day (Routines)
where (Routines)
which_bit (Routines)
WHITE (Constants)
wildcard_file (Routines)
WIN32 (Constants)
wrap (Routines)
WRAP (Constants)
write (Routines)
write_file (Routines)
write_lines (Routines)
writef (Routines)
writefln (Routines)

X
xor_bits (Routines)

Y
YEAR (Constants)
YEARS (Constants)
years_day (Routines)
YELLOW (Constants)