Part II - Library Routines

1. Introduction

A large number of library routines are provided. Some are built right into the interpreter, ex.exe, exw.exe or exu. Others are written in Euphoria and you must include one of the .e files in euphoria\include to use them. Where this is the case, the appropriate include file is noted in the "Syntax" part of the description. Of course an include file need only be included once in your program. The editor displays in magenta those routines that are built into the interpreter, and require no include file. You can override the definition of these built-in routines by defining your own routine with the same name. You will get a suppressible warning if you do this.

To indicate what kind of object may be passed in and returned, the following prefixes are used:

x - a general object (atom or sequence)

s - a sequence

a - an atom

i - an integer

fn - an integer used as a file number

st - a string sequence, or single-character atom

Some routines are only available on one or two of the three platforms. This is noted with "Platform: DOS32" or "Platform: WIN32" or "Platform: Linux" in the description of the routine, and with (DOS32) or (WIN32) or (Linux) in some other places.

A run-time error message will usually result if an illegal argument value is passed to any of these routines.

2. Routines by Application Area

2.1 Predefined Types

As well as declaring variables with these types, you can also call them just like ordinary functions, in order to test if a value is a certain type.

integer - test if an object is an integer

atom - test if an object is an atom

```
sequence        - test if an object is a sequence

object          - test if an object is an object (always true)
```

2.2 Sequence Manipulation

```
length          - return the length of a sequence

repeat          - repeat an object n times to form a sequence of length n

reverse         - reverse a sequence

append          - add a new element to the end of a sequence

prepend         - add a new element to the beginning of a sequence
```

2.3 Searching and Sorting

```
compare         - compare two objects

equal           - test if two objects are identical

find            - find an object in a sequence

match           - find a sequence as a slice of another sequence

sort            - sort the elements of a sequence into ascending order

custom_sort     - sort the elements of a sequence based on a compare function
                  that you supply
```

2.4 Pattern Matching

```
lower           - convert an atom or sequence to lower case

upper           - convert an atom or sequence to upper case

wildcard_match  - match a pattern containing ? and * wildcards

wildcard_file   - match a file name against a wildcard specification
```

2.5 Math

These routines can be applied to individual atoms or to sequences of values.
See Part I - Core Language - Operations on Sequences.

sqrt - calculate the square root of an object

rand - generate random numbers

sin - calculate the sine of an angle

arcsin - calculate the angle with a given sine

cos - calculate the cosine of an angle

arccos - calculate the angle with a given cosine

tan - calculate the tangent of an angle

arctan - calculate the arc tangent of a number

log - calculate the natural logarithm

floor - round down to the nearest integer

remainder - calculate the remainder when a number is divided by another

power - calculate a number raised to a power

PI - the mathematical value PI (3.14159...)

2.6 Bitwise Logical Operations

These routines treat numbers as collections of binary bits, and logical operations are performed on corresponding bits in the binary representation of the numbers. There are no routines for shifting bits left or right, but you can achieve the same effect by multiplying or dividing by powers of 2.

and_bits - perform logical AND on corresponding bits

or_bits - perform logical OR on corresponding bits

xor_bits - perform logical XOR on corresponding bits

not_bits - perform logical NOT on all bits

2.7 File and Device I/O

To do input or output on a file or device you must first open the file or device, then use the routines below to read or write to it, then close the file or device. open() will give you a file number to use as the first argument of the other I/O routines. Certain files/devices are opened for you automatically (as text files):

0 - standard input

1 - standard output
2 - standard error

Unless you redirect them on the command-line, standard input comes from the keyboard, standard output and standard error go to the screen. When you write something to the screen it is written immediately without buffering. If you write to a file, your characters are put into a buffer until there are enough of them to write out efficiently. When you close() or flush() the file or device, any remaining characters are written out. Input from files is also buffered. When your program terminates, any files that are still open will be closed for you automatically.

Note:

If a program (written in Euphoria or any other language) has a file open for writing, and you are forced to reboot your computer for any reason, you should immediately run scandisk to repair any damage to the file system that may have occurred.

open	- open a file or device
close	- close a file or device
flush	- flush out buffered data to a file or device
lock_file	- lock a file or device
unlock_file	- unlock a file or device
print	- print a Euphoria object on one line, with braces and commas {,,} to show the structure
pretty_print	- print a Euphoria object in a nice readable form, using multiple lines and appropriate indentation
? x	- shorthand for print(1, x)
sprint	- return a printed Euphoria object as a string sequence
printf	- formatted print to a file or device
sprintf	- formatted print returned as a string sequence
puts	- output a string sequence to a file or device
getc	- read the next character from a file or device
gets	- read the next line from a file or device
get_bytes	- read the next n bytes from a file or device
prompt_string	- prompt the user to enter a string
get_key	- check for key pressed by the user, don't wait
wait_key	- wait for user to press a key

get - read the representation of any Euphoria object from a file

prompt_number - prompt the user to enter a number

value - read the representation of any Euphoria object from a string

seek - move to any byte position within an open file

where - report the current byte position in an open file

current_dir - return the name of the current directory

chdir - change to a new current directory

dir - return complete info on all files in a directory

walk_dir - recursively walk through all files in a directory

allow_break - allow control-c/control-Break to terminate your program or not

check_break - check if user has pressed control-c or control-Break

2.8 Mouse Support (DOS32 and Linux)

get_mouse - return mouse "events" (clicks, movements)

mouse_events - select mouse events to watch for

mouse_pointer - display or hide the mouse pointer

2.9 Operating System

time - number of seconds since a fixed point in the past

tick_rate - set the number of clock ticks per second (DOS32)

date - current year, month, day, hour, minute, second etc.

command_line - command-line used to run this program

getenv - get value of an environment variable

system - execute an operating system command line

system_exec - execute a program and get its exit code

abort - terminate execution

sleep - suspend execution for a period of time

platform   - find out which operating system are we running on

2.10 Special Machine-Dependent Routines

machine_func   - specialized internal operations with a return value

machine_proc   - specialized internal operations with no return value

2.11 Debugging

trace   - dynamically turns tracing on or off

profile   - dynamically turns profiling on or off

2.12 Graphics & Sound

The following routines let you display information on the screen. In DOS, the PC screen can be placed into one of many graphics modes. See the top of include\graphics.e for a description of the modes. There are two basic types of graphics mode available. "Text" modes divide the screen up into lines, where each line has a certain number of characters. "Pixel-graphics" modes divide the screen up into many rows of dots, or "pixels". Each pixel can be a different color. In text modes you can display text only, with the choice of a foreground and a background color for each character. In pixel-graphics modes you can display lines, circles, dots, and also text. Any pixels that would be off the screen are safely clipped out of the image.

For DOS32 we've included a routine for making sounds on your PC speaker. To make more sophisticated sounds, get the Sound Blaster library developed by Jacques Deschenes. It's available on the Euphoria Web page.

The following routines work in all text and pixel-graphics modes:

clear_screen   - clear the screen

position   - set cursor line and column

get_position   - return cursor line and column

graphics_mode   - select a new pixel-graphics or text mode (DOS32)

video_config   - return parameters of current mode

scroll   - scroll text up or down

wrap   - control line wrap at right edge of screen

```
text_color        - set foreground text color

bk_color          - set background color

palette           - change color for one color number (DOS32)

all_palette       - change color for all color numbers (DOS32)

get_all_palette   - get the palette values for all colors (DOS32)

read_bitmap       - read a bitmap (.bmp) file and return a palette and a 2-d
                    sequence of pixels

save_bitmap       - create a bitmap (.bmp) file, given a palette and a 2-d
                    sequence of pixels

get_active_page   - return the page currently being written to (DOS32)

set_active_page   - change the page currently being written to (DOS32)

get_display_page  - return the page currently being displayed (DOS32)

set_display_page  - change the page currently being displayed (DOS32)

sound             - make a sound on the PC speaker (DOS32)


The following routines work in text modes only:
-----------------------------------------------

cursor              - select cursor shape

text_rows           - set number of lines on text screen

get_screen_char     - get one character from the screen (DOS32, Linux)

put_screen_char     - put one or more characters on the screen (DOS32, Linux)

save_text_image     - save a rectangular region from a text screen (DOS32,
                      Linux)

display_text_image  - display an image on the text screen (DOS32, Linux)


The following routines work in pixel-graphics modes only (DOS32):
-----------------------------------------------------------------

pixel           - set color of a pixel or set of pixels

get_pixel       - read color of a pixel or set of pixels

draw_line       - connect a series of graphics points with a line

polygon         - draw an n-sided figure

ellipse         - draw an ellipse or circle
```

save_screen - save the screen to a bitmap (.bmp) file

save_image - save a rectangular region from a pixel-graphics screen

display_image - display an image on the pixel-graphics screen

2.13 Machine Level Interface

We've grouped here a number of routines that you can use to access your machine at a low-level. With this low-level machine interface you can read and write to memory. You can also set up your own 386+ machine language routines and call them.

Some of the routines listed below are unsafe, in the sense that Euphoria can't protect you if you use them incorrectly. You could crash your program or even your system. Under DOS32, if you reference a bad memory address it will often be safely caught by the CauseWay DOS extender, and you'll get an error message on the screen plus a dump of machine-level information in the file cw.err. Under WIN32, the operating system will usually pop up a termination box giving a diagnostic message plus register information. Under Linux you'll typically get a segmentation violation.

Note:

> To assist programmers in debugging code involving these unsafe routines, we have supplied safe.e, an alternative to machine.e. If you copy euphoria\include\safe.e into the directory containing your program, and you rename safe.e as machine.e in that directory, your program will run using safer (but slower) versions of these low-level routines. safe.e can catch many errors, such as poking into a bad memory location. See the comments at the top of safe.e for instructions on how to use it and how to configure it optimally for your program.

These machine-level-interface routines are important because they allow Euphoria programmers to access low-level features of the hardware and operating system. For some applications this is essential.

Machine code routines can be written by hand, or taken from the disassembled output of a compiler for C or some other language. Pete Eberlein has written a "mini-assembler" for use with Euphoria. See the Archive. Remember that your machine code will be running in 32-bit protected mode. See demo\callmach.ex for an example.

peek - read one or more bytes from memory

peek4s - read 4-byte signed values from memory

peek4u - read 4-byte unsigned values from memory

poke - write one or more bytes to memory

poke4 - write 4-byte values into memory

```
mem_copy        - copy a block of memory

mem_set         - set a block of memory to a value

call            - call a machine language routine

dos_interrupt   - call a DOS software interrupt routine (DOS32)

allocate        - allocate a block of memory

free            - deallocate a block of memory

allocate_low    - allocate a block of low memory (address less than 1Mb)
                  (DOS32)

free_low        - free a block allocated with allocate_low (DOS32)

allocate_string - allocate a string of characters with 0 terminator

register_block  - register an externally-allocated block of memory

unregister_block- unregister an externally-allocated block of memory

get_vector      - return address of interrupt handler (DOS32)

set_vector      - set address of interrupt handler (DOS32)

lock_memory     - ensure that a region of memory will never be swapped out
                  (DOS32)

int_to_bytes    - convert an integer to 4 bytes

bytes_to_int    - convert 4 bytes to an integer

int_to_bits     - convert an integer to a sequence of bits

bits_to_int     - convert a sequence of bits to an integer

atom_to_float64 - convert an atom, to a sequence of 8 bytes in IEEE 64-bit
                  floating-point format

atom_to_float32 - convert an atom, to a sequence of 4 bytes in IEEE 32-bit
                  floating-point format

float64_to_atom - convert a sequence of 8 bytes in IEEE 64-bit floating-point
                  format, to an atom

float32_to_atom - convert a sequence of 4 bytes in IEEE 32-bit floating-point
                  format, to an atom

set_rand        - set the random number generator so it will generate a
                  repeatable series of random numbers

use_vesa        - force the use of the VESA graphics standard (DOS32)

crash_file      - specify the file for writing error diagnostics if Euphoria
                  detects an error in your program.
```

crash_message - specify a message to be printed if Euphoria detects an error in your program

2.14 Dynamic Calls

These routines let you call Euphoria procedures and functions using a unique integer known as a routine identifier, rather than by specifying the name of the routine.

routine_id - get a unique identifying number for a Euphoria routine

call_proc - call a Euphoria procedure using a routine id

call_func - call a Euphoria function using a routine id

2.15 Calling C Functions (WIN32 and Linux)

See platform.doc for a description of WIN32 and Linux programming in Euphoria.

open_dll - open a Windows dynamic link library (.dll file) or Linux shared library (.so file)

define_c_proc - define a C function that is VOID (no value returned), or whose value your program will ignore

define_c_func - define a C function that returns a value that your program will use

define_c_var - get the memory address of a C variable.

c_proc - call a C function, ignoring any return value

c_func - call a C function and get the return value

call_back - get a 32-bit machine address for a Euphoria routine for use as a call-back address

message_box - pop up a small window to get a Yes/No/Cancel response from the user

free_console - delete the console text window

instance - get the instance handle for the current program

3. Alphabetical Listing of all Routines

----------<?>----------

Syntax: ? x

Description: This is just a shorthand way of saying: pretty_print(1, x, {}) - i.e. printing the value of an expression to the standard output, with braces and indentation to show the structure.

Example:

```
? {1, 2} + {3, 4}  -- will display {4, 6}
```

See Also: pretty_print, print

----------<abort>----------

Syntax: abort(i)

Description: Abort execution of the program. The argument i is a small integer status value to be returned to the operating system. A value of 0 generally indicates successful completion of the program. Other values can indicate various kinds of errors. DOS batch (.bat) programs can read this value using the errorlevel feature. A Euphoria program can read this value using system_exec().

Comments: 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(), ex.exe/exw.exe/exu 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:

```
if x = 0 then
    puts(ERR, "can't divide by 0 !!!\n")
    abort(1)
else
    z = y / x
end if
```

See Also: crash_message, system_exec

----------<all_palette>----------

Platform: DOS32

Syntax: include graphics.e

```
            all_palette(s)

Description: Specify new color intensities for the entire set of colors in the
            current graphics mode. s is a sequence of the form:

            {{r,g,b}, {r,g,b}, ..., {r,g,b}}

            Each element specifies a new color intensity {red, green, blue}
            for the corresponding color number, starting with color number 0.
            The values for red, green and blue must be in the range 0 to 63.

Comments:   This executes much faster than if you were to use palette() to
            set the new color intensities one by one. This procedure can be
            used with read_bitmap() to quickly display a picture on the
            screen.

Example Program: demo\dos32\bitmap.ex

See Also:   get_all_palette, palette, read_bitmap, video_config,
            graphics_mode


----------------------------------<allocate>-----------------------------------

Syntax:     include machine.e
            a = allocate(i)

Description: Allocate i contiguous bytes of memory. Return the address of the
            block of memory, or return 0 if the memory can't be allocated.
            The address returned will be at least 4-byte aligned.

Comments:   When you are finished using the block, you should pass the
            address of the block to free(). This will free the block and make
            the memory available for other purposes. Euphoria will never free
            or reuse your block until you explicitly call free(). When your
            program terminates, the operating system will reclaim all memory
            for use with other programs.

Example:

            buffer = allocate(100)
            for i = 0 to 99 do
                poke(buffer+i, 0)
            end for


See Also:   free, allocate_low, peek, poke, call


--------------------------------<allocate_low>---------------------------------

Platform:   DOS32

Syntax:     include machine.e
            i2 = allocate_low(i1)

Description: Allocate i1 contiguous bytes of low memory, i.e. conventional
```

memory (address below 1 megabyte). Return the address of the block of memory, or return 0 if the memory can't be allocated.

Comments: Some DOS software interrupts require that you pass one or more addresses in registers. These addresses must be conventional memory addresses for DOS to be able to read or write to them.

Example Program: demo\dos32\dosint.ex

See Also: dos_interrupt, free_low, allocate, peek, poke

<allocate_string>

Syntax:
```
include machine.e
a = allocate_string(s)
```

Description: Allocate space for string sequence s. Copy s into this space along with a 0 terminating character. This is the format expected for C strings. The memory address of the string will be returned. If there is not enough memory available, 0 will be returned.

Comments: To free the string, use free().

Example:

```
atom title

title = allocate_string("The Wizard of Oz")
```

Example Program: demo\win32\window.exw

See Also: allocate, free

<allow_break>

Syntax:
```
include file.e
allow_break(i)
```

Description: When i is 1 (true) control-c and control-Break can terminate your program when it tries to read input from the keyboard. When i is 0 (false) your program will not be terminated by control-c or control-Break.

Comments: DOS will display ^C on the screen, even when your program cannot be terminated.

Initially your program can be terminated at any point where it tries to read from the keyboard. It could also be terminated by other input/output operations depending on options the user has set in his config.sys file. (Consult an MS-DOS manual for the BREAK command.) For some types of program this sudden termination could leave things in a messy state and might result in loss of data. allow_break(0) lets you avoid this situation.

You can find out if the user has pressed control-c or control-Break by calling check_break().

Example:

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

See Also: check_break

<and_bits>

Syntax: x3 = and_bits(x1, x2)

Description: Perform the logical AND operation on corresponding bits in x1 and x2. A bit in x3 will be 1 only if the corresponding bits in x1 and x2 are both 1.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply.

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().

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

----------<append>----------

Syntax: s2 = append(s1, x)

Description: Create a new sequence identical to s1 but with x added on the end as the last element. The length of s2 will be length(s1) + 1.

Comments: If x is an atom this is equivalent to s2 = s1 & 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 s1 and s2 are actually the same variable (as in Example 1 below) is highly optimized.

Example 1: You can use append() to dynamically grow a sequence, e.g.

```
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: Any kind of Euphoria object can be appended to a sequence, e.g.

```
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, concatenation operator &, sequence-formation operator

----------<arccos>----------

Syntax:
```
include misc.e
x2 = arccos(x1)
```

Description: Return an angle with cosine equal to x1.

Comments: The argument, x1, must be in the range -1 to +1 inclusive.

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:

```
s = arccos({-1,0,1})
-- s is {3.141592654, 1.570796327, 0}
```

See Also: cos, arcsin, arctan

----------<arcsin>----------

Syntax:
```
include misc.e
x2 = arcsin(x1)
```

Description: Return an angle with sine equal to x1.

Comments: The argument, x1, must be in the range -1 to +1 inclusive.

A value between -PI/2 and +PI/2 (radians) 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:

```
s = arcsin({-1,0,1})
-- s is {-1.570796327, 0, 1.570796327}
```

See Also: sin, arccos, arctan

----------<arctan>----------

Syntax:
```
x2 = arctan(x1)
```

Description: Return an angle with tangent equal to x1.

Comments: A value between -PI/2 and PI/2 (radians) will be returned.

This function may be applied to an atom or to all elements of a sequence.

arctan() is faster than arcsin() or arccos().

Example:

```
s = arctan({1,2,3})
-- s is {0.785398, 1.10715, 1.24905}
```

See Also: tan, arcsin, arccos

----------<atom>----------

Syntax: i = atom(x)

Description: Return 1 if x is an atom else return 0.

Comments: This serves to define the atom type. You can also call it like an ordinary function to determine if an object is an atom.

Example 1:

```
atom a
a = 5.99
```

Example 2:

```
object line

line = gets(0)
if atom(line) then
    puts(SCREEN, "end of file\n")
end if
```

See Also: sequence, object, integer, atoms and sequences

----------<atom_to_float32>----------

Syntax:
```
include machine.e
s = atom_to_float32(a1)
```

Description: Convert a Euphoria atom to a sequence of 4 single-byte values. These 4 bytes contain the representation of an IEEE floating-point number in 32-bit format.

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.

Example:

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

See Also: atom_to_float64, float32_to_atom

---------------------------------<atom_to_float64>---------------------------------

Syntax: include machine.e
s = atom_to_float64(a1)

Description: Convert a Euphoria atom to a sequence of 8 single-byte values. These 8 bytes contain the representation of an IEEE floating-point number in 64-bit format.

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: atom_to_float32, float64_to_atom

---------------------------------<bits_to_int>---------------------------------

Syntax: include machine.e
a = bits_to_int(s)

Description: Convert a sequence of binary 1's and 0's into a positive number. The least-significant bit is s[1].

Comments: 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:

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

See Also: int_to_bits, operations on sequences

---------------------------------<bk_color>---------------------------------

Syntax: include graphics.e
bk_color(i)

Description: Set the background color to one of the 16 standard colors. In pixel-graphics modes the whole screen is affected immediately. In text modes any new characters that you print will have the new

background color.

Comments: The 16 standard colors are defined as constants in graphics.e

In pixel-graphics modes, color 0 which is normally BLACK, will be set to the same {r,g,b} palette value as color number i.

In some pixel-graphics modes, there is a border color that appears at the edges of the screen. In 256-color modes, this is the 17th color in the palette. You can control it as you would any other color.

In text modes, 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, palette

<bytes_to_int>

Syntax:
```
include machine.e
a = bytes_to_int(s)
```

Description: Convert a 4-element sequence of byte values to an atom. The elements of s are in the order expected for a 32-bit integer on the 386+, i.e. least-significant byte first.

Comments: The result could be greater than the integer type allows, so you should assign it to an atom.

s would normally contain positive values that have been read using peek() from 4 consecutive memory locations.

Example:

```
atom int32

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

See Also: int_to_bytes, bits_to_int, peek, peek4s, peek4u, poke

---

<call>

Syntax: call(a)

Description: Call a machine language routine that starts at address a. This routine must execute a RET instruction #C3 to return control to Euphoria. The routine should save and restore any registers that it uses.

Comments: You can allocate a block of memory for the routine and then poke in the bytes of machine code. You might allocate other blocks of memory for data and parameters that the machine code can operate on. The addresses of these blocks could be poked into the machine code.

Example Program: demo\callmach.ex

See Also: allocate, free, peek, poke, poke4, c_proc, define_c_proc

----------<call_back>----------

Platform: WIN32, Linux, FreeBSD

Syntax: include dll.e
a = call_back(i)
or
a = call_back({i1, i})

Description: Get a machine address for the Euphoria routine with routine id i. This address can be used by Windows, or an external C routine in a Windows .dll or Linux/FreeBSD shared library (.so), as a 32-bit "call-back" address for calling your Euphoria routine. On Windows, you can specify i1, which determines the C calling convention that can be used to call your routine. If i1 is '+', then your routine will work with the cdecl calling convention. By default it will work with the stdcall convention. On Linux and FreeBSD you should only use the first form, as there is just one standard calling convention

Comments: 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 exw. This is because the Watcom C compiler (used to build exw) has a non-standard way of handling cdecl floating-point return values.

Example Program: demo\win32\window.exw, demo\linux\qsort.exu

See Also: routine_id, platform.doc

----------<c_func>----------

Platform: WIN32, Linux, FreeBSD

Syntax: a = c_func(i, s)

Description: Call the C function, or machine code routine, with routine id i. i must be a valid routine id returned by define_c_func(). s is a sequence of argument values of length n, where n is the number of arguments required by the function. a will be the result returned by the C function.

Comments: If the C function does not take any arguments then s 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 C function could be part of a .dll created by the Euphoria To C Translator.

Example:

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

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

-- the C function BeginPaint takes a C int argument and
-- a C pointer, and returns a C int as a result:
BeginPaint = define_c_func(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_func, open_dll, platform.doc

----------<c_proc>----------

Platform: WIN32, Linux, FreeBSD

Syntax: c_proc(i, s)

Description: Call the C function, or machine code routine, with routine id i. i must be a valid routine id returned by define_c_proc(). s is a sequence of argument values of length n, where n is the number of arguments required by the function.

Comments: If the C function does not take any arguments then s 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 C function could be part of a .dll created by the Euphoria To C Translator.

Example:

```
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, call, define_c_proc, open_dll, platform.doc

---

<call_func>

Syntax: x = call_func(i, s)

Description: Call the user-defined Euphoria function with routine id i. i must be a valid routine id returned by routine_id(). s must be a sequence of argument values of length n, where n is the number of arguments required by function i. x will be the result returned by function i.

Comments: If function i does not take any arguments then s should be {}.

Example Program: demo\csort.ex

See Also: call_proc, routine_id

<call_proc>

Syntax: call_proc(i, s)

Description: Call the user-defined Euphoria procedure with routine id i. i must be a valid routine id returned by routine_id(). s must be a sequence of argument values of length n, where n is the number of arguments required by procedure i.

Comments: If procedure i does not take any arguments then s should be {}.

Example:

```
global integer foo_id

procedure x()
    call_proc(foo_id, {1, "Hello World\n"})
end procedure

procedure foo(integer a, sequence s)
    puts(a, s)
end procedure

foo_id = routine_id("foo")

x()
```

See Also: call_func, routine_id

----------<chdir>----------

Syntax:
```
include file.e
i = chdir(s)
```

Description: Set the current directory to the path given by sequence s. s must name an existing directory on the system. If successful, chdir() returns 1. If unsuccessful, chdir() returns 0.

Comments: By setting the current directory, you can refer to files in that directory using just the file name.

The function current_dir() will return the name of the current directory.

On DOS32 and WIN32 the current directory is a global property shared by all the processes running under one shell. On Linux/FreeBSD, a subprocess can change the current directory for itself, but this won't affect the current directory of its parent process.

Example:

```
if chdir("c:\\euphoria") then
    f = open("readme.doc", "r")
else
    puts(1, "Error: No euphoria directory?\n")
```

```
            end if

See Also:   current_dir


---------------------------------<check_break>---------------------------------

Syntax:     include file.e
            i = check_break()

Description: Return the number of times that control-c or control-Break have
            been pressed since the last call to check_break(), or since the
            beginning of the program if this is the first call.

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

            Neither control-c nor control-Break will be returned as input
            characters when you read the keyboard. You can only detect them
            by calling check_break().

Example:

            k = get_key()
            if check_break() then
                temp = graphics_mode(-1)
                puts(1, "Shutting down...")
                save_all_user_data()
                abort(1)
            end if


See Also:   allow_break, get_key


--------------------------------<clear_screen>--------------------------------

Syntax:     clear_screen()

Description: Clear the screen using the current background color (may be set
            by bk_color()).

Comments:   This works in all text and pixel-graphics modes.

See Also:   bk_color, graphics_mode


-----------------------------------<close>-----------------------------------

Syntax:     close(fn)

Description: Close a file or device and flush out any still-buffered
            characters.
```

Comments: Any still-open files will be closed automatically when your program terminates.

See Also: open, flush

----------<command_line>----------

Syntax: s = command_line()

Description: Return a sequence of strings, where each string is a word from the command-line that started your program. The first word will be the path to either the Euphoria executable, ex.exe, exw.exe or exu, or to your bound executable file. The next word is either the name of your Euphoria main file, or (again) the path to your bound executable file. After that will come any extra words typed by the user. You can use these words in your program.

Comments: The Euphoria interpreter itself does not use any command-line options. You are free to use any options for your own program.

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

If you bind your program you will find that all command-line arguments remain the same, except for the first two, even though your user no longer types "ex" on the command-line (see examples below).

Example 1:

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

cmd = command_line()

-- cmd will be:
      {"C:\EUPHORIA\BIN\EX.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", -- spacer
        "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 as a .exe or not.
```

See Also: getenv

----------<compare>----------

Syntax: i = compare(x1, x2)

Description: Return 0 if objects x1 and x2 are identical, 1 if x1 is greater than x2, -1 if x1 is less than x2. Atoms are considered to be less than sequences. Sequences are compared "alphabetically" starting with the first element until a difference is found.

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({12345, 99999, -1, 700, 2},
            {12345, 99999, -1, 699, 3, 0})
-- x will be 1 because 700 > 699
```

Example 4:

```
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

----------<cos>----------

Syntax: x2 = cos(x1)

Description: Return the cosine of x1, where x1 is in radians.

Comments: This function may be applied to an atom or to all elements of a sequence.

Example:

```
x = cos({.5, .6, .7})

-- x is {0.8775826, 0.8253356, 0.7648422}
```

See Also: sin, tan, log, sqrt

<crash_file>

Syntax:
```
include machine.e
crash_file(s)
```

Description: Specify a file name, s, for holding error diagnostics if Euphoria must stop your program due to a compile-time or run-time error.

Comments: Normally Euphoria prints a diagnostic message such as "syntax error" or "divide by zero" on the screen, as well as dumping debugging information into ex.err in the current directory. By calling crash_file() you can control the directory and file name where the debugging information will be written.

s may be empty, i.e. "". In this case no diagnostics or debugging information will be written to either a file or the screen. s might also be "NUL" or "/dev/null", in which case diagnostics will be written to the screen, but the ex.err information will be discarded.

You can call crash_file() as many times as you like from different parts of your program. The file specified by the last call will be the one used.

Example:

```
crash_file("\\tmp\\mybug")
```

See Also: abort, crash_message, debugging and profiling

<crash_message>

Syntax:
```
include machine.e
crash_message(s)
```

Description: Specify a string, s, to be printed on the screen in the event that Euphoria must stop your program due to a compile-time or run-time error.

Comments: Normally Euphoria prints a diagnostic message such as "syntax error" or "divide by zero" on the screen, as well as dumping debugging information into ex.err. Euphoria's error messages will

not be meaningful for your users unless they happen to be Euphoria programmers. By calling crash_message() you can control the message that will appear on the screen. Debugging information will still be stored in ex.err. You won't lose any information by doing this.

s may contain '\n', new-line characters, so your message can span several lines on the screen. Euphoria will switch to the top of a clear text-mode screen before printing your message.

You can call crash_message() as many times as you like from different parts of your program. The message specified by the last call will be the one displayed.

Example:

```
crash_message("An unexpected error has occurred!\n" &
              "Please contact john_doe@whoops.com\n" &
              "Do not delete the file \"ex.err\".\n")
```

See Also: abort, crash_file, debugging and profiling

----------<current_dir>----------

Syntax:
```
include file.e
s = current_dir()
```

Description: Return the name of the current working directory.

Example:

```
sequence s
s = current_dir()
-- s would have "C:\EUPHORIA\DOC" if you were in that directory
```

See Also: dir, chdir, getenv

----------<cursor>----------

Syntax:
```
include graphics.e
cursor(i)
```

Description: Select a style of cursor. graphics.e contains:

```
    global constant NO_CURSOR = #2000,
             UNDERLINE_CURSOR = #0607,
        THICK_UNDERLINE_CURSOR = #0507,
            HALF_BLOCK_CURSOR = #0407,
                 BLOCK_CURSOR = #0007
```

The second and fourth hex digits (from the left) determine the top and bottom rows 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.

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

Example:

             cursor(BLOCK_CURSOR)


See Also:    graphics_mode, text_rows


---------------------------------<custom_sort>---------------------------------

Syntax:      include sort.e
             s2 = custom_sort(i, s1)

Description: Sort the elements of sequence s1, using a compare function with
             routine id i.

Comments:    Your compare function must be a function of two arguments similar
             to Euphoria's compare(). It will compare two objects and return
             -1, 0 or +1.

Example Program: demo\csort.ex

See Also:    sort, compare, routine_id


-----------------------------------<date>-------------------------------------

Syntax:      s = date()

Description: Return a sequence with the following information:

                          { year,  -- since 1900
                           month,  -- January = 1
                             day,  -- day of month, starting at 1
                            hour,  -- 0 to 23
                          minute,  -- 0 to 59
                          second,  -- 0 to 59
                      day of the week,  -- Sunday = 1
                      day of the year}  -- January 1st = 1

Example:

             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


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 will be 100. In 2001 it will be 101, etc.

See Also:    time
```

<define_c_func>

Syntax:

```
include dll.e
i1 = define_c_func(x1, x2, s1, i2)
```

Description: Define the characteristics of either a C function, or a machine-code routine that returns a value. A small integer, i1, known as a routine id, will be returned. Use this routine id as the first argument to c_func() when you wish to call the function from Euphoria.

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open_dll(). If the C function can't be found, -1 will be returned as the routine id. 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.

s1 is a list of the parameter types for the function. i2 is the return type of the function. A list of C types is contained in dll.e, and these can be used to define machine code parameters as well:

```
global constant C_CHAR = #01000001,
                C_UCHAR = #02000001,
                C_SHORT = #01000002,
               C_USHORT = #02000002,
                  C_INT = #01000004,
                 C_UINT = #02000004,
                 C_LONG = C_INT,
                C_ULONG = C_UINT,
              C_POINTER = C_ULONG,
                C_FLOAT = #03000004,
               C_DOUBLE = #03000008
```

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:

```
global constant
       E_INTEGER = #06000004,
       E_ATOM    = #07000004,
       E_SEQUENCE= #08000004,
       E_OBJECT  = #09000004
```

Comments: 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. 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.

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 exw 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 exw) 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.

ex.exe uses calls to WATCOM floating-point routines (which then use hardware floating-point instructions if available), so floating-point values are generally passed and returned in integer register-pairs rather than floating-point registers. You'll have to disassemble some Watcom code to see how it works.

Example:

```
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: euphoria\demo\callmach.ex, c_func, define_c_proc, c_proc,

open_dll, platform.doc

<define_c_proc>

Syntax:

```
include dll.e
i1 = define_c_proc(x1, x2, s1)
```

Description: 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. A small integer, known as a routine id, will be returned. Use this routine id as the first argument to c_proc() when you wish to call the routine from Euphoria.

When defining a C function, x1 is the address of the library containing the C function, while x2 is the name of the C function. x1 is a value returned by open_dll(). If the C function can't be found, -1 will be returned as the routine id. On Windows, you can add a '+' character as a prefix to the function 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, 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.

s1 is a list of the parameter types for the function. A list of C types is contained in dll.e, and shown above. These can 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 contained in dll.e, and shown above.

Comments: 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.

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:

```
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, platform.doc

----------<define_c_var>----------

Platform: WIN32, Linux, FreeBSD

Syntax:
```
            include dll.e
            a1 = define_c_var(a2, s)
```

Description: a2 is the address of a Linux or FreeBSD shared library, or Windows .dll, as returned by open_dll(). s is the name of a global C variable defined within the library. a1 will be the memory address of variable s.

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.

Example Program: euphoria/demo/linux/mylib.exu

See Also: c_proc, define_c_func, c_func, open_dll, platform.doc

----------<dir>----------

Syntax:
```
            include file.e
            x = dir(st)
```

Description: Return directory information for the file or directory named by st. If there is no file or directory with this name then -1 is returned. On Windows and DOS st can contain * and ? wildcards to select multiple files.

This information is similar to what you would get from the DOS DIR command. A sequence is returned where each element is a sequence that describes one file or subdirectory.

If st names a directory you may have entries for "." and "..", just as with the DOS DIR command. If st names a file then x will have just one entry, i.e. length(x) will be 1. If st contains wildcards you may have multiple entries.

Each entry contains the name, attributes and file size as well as the year, month, day, hour, minute and second of the last modification. You can refer to the elements of an entry with the following constants defined in file.e:

```
global constant D_NAME = 1,
                D_ATTRIBUTES = 2,
                      D_SIZE = 3,

                       D_YEAR = 4,
                      D_MONTH = 5,
                        D_DAY = 6,

                       D_HOUR = 7,
                     D_MINUTE = 8,
                     D_SECOND = 9
```

The attributes element is a string sequence containing characters chosen from:

```
'd' -- directory
'r' -- read only file
'h' -- hidden file
's' -- system file
'v' -- volume-id entry
'a' -- archive file
```

A normal file without special attributes would just have an empty string, "", in this field.

Comments: 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, st can have a long file or directory name anywhere in the path.

Under Linux/FreeBSD, the only attribute currently available is 'd'.

DOS32: The file name returned in D_NAME will be a standard DOS 8.3 name. (See Archive Web page for a better solution).

WIN32: The file name returned in D_NAME will be a long file name.

Example:

```
d = dir(current_dir())

-- d might have:
  {
    {".",    "d",     0  1994, 1, 18,  9, 30, 02},
    {"..",   "d",     0  1994, 1, 18,  9, 20, 14},
    {"fred", "ra", 2350, 1994, 1, 22, 17, 22, 40},
    {"sub",  "d" ,    0, 1993, 9, 20,  8, 50, 12}
```

```
  }

d[3][D_NAME] would be "fred"
```

Example Programs: bin\search.ex, bin\install.ex

See Also: wildcard_file, current_dir, open

<display_image>

Platform: DOS32

Syntax:
```
include image.e
display_image(s1, s2)
```

Description: Display at point s1 on a pixel-graphics screen the 2-d sequence of pixels contained in s2. s1 is a two-element sequence {x, y}. s2 is a sequence of sequences, where each sequence is one horizontal row of pixel colors to be displayed. The first pixel of the first sequence is displayed at s1. It is the top-left pixel. All other pixels appear to the right or below of this point.

Comments: s2 might be the result of a previous call to save_image(), or read_bitmap(), or it could be something you have created.

The sequences (rows) of the image do not have to all be the same length.

Example:

```
display_image({20,30}, {{7,5,9,4,8},
                        {2,4,1,2},
                        {1,0,1,0,4,6,1},
                        {5,5,5,5,5,5}})
-- This will display a small image containing 4 rows of
-- pixels. The first pixel (7) of the top row will be at
-- {20,30}. The top row contains 5 pixels. The last row
-- contains 6 pixels ending at {25,33}.
```

Example Program: demo\dos32\bitmap.ex

See Also: save_image, read_bitmap, display_text_image

<display_text_image>

Platform: DOS32, Linux, FreeBSD

Syntax:
```
include image.e
display_text_image(s1, s2)
```

Description: Display the 2-d sequence of characters and attributes contained in s2 at line s1[1], column s1[2]. s2 is a sequence of sequences,

where each sequence is a string of characters and attributes to be displayed. The top-left character is displayed at s1. Other characters appear to the right or below this position. The attributes indicate the foreground and background color of the preceding character. On DOS32, the attribute should consist of the foreground color plus 16 times the background color.

Comments: s2 would normally be the result of a previous call to save_text_image(), although you could construct it yourself.

This routine 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.

The sequences of the text image do not have to all be the same length.

Example:

```
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, display_image, put_screen_char

<dos_interrupt>

Platform: DOS32

Syntax:
```
include machine.e
s2 = dos_interrupt(i, s1)
```

Description: Call DOS software interrupt number i. s1 is a 10-element sequence of 16-bit register values to be used as input to the interrupt routine. s2 is a similar 10-element sequence containing output register values after the call returns. machine.e has the following declaration which shows the order of the register values in the input and output sequences.

```
    global constant REG_DI = 1,
                    REG_SI = 2,
                    REG_BP = 3,
```

```
                                REG_BX = 4,
                                REG_DX = 5,
                                REG_CX = 6,
                                REG_AX = 7,
                                REG_FLAGS = 8,
                                REG_ES = 9,
                                REG_DS = 10
```

Comments: The register values returned in s2 are always positive values between 0 and #FFFF (65535).

The flags value in s1[REG_FLAGS] is ignored on input. On output the least significant bit of s2[REG_FLAGS] has the carry flag, which usually indicates failure if it is set to 1.

Certain interrupts require that you supply addresses of blocks of memory. These addresses must be conventional, low-memory addresses. You can allocate/deallocate low-memory using allocate_low() and free_low().

With DOS software interrupts you can perform a wide variety of specialized operations, anything from formatting your floppy drive to rebooting your computer. For documentation on these interrupts consult a technical manual such as Peter Norton's "PC Programmer's Bible", or download Ralf Brown's Interrupt List from the Web:

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/ralf/pub/WWW/files.html

Example:

```
sequence registers

registers = repeat(0, 10)  -- no registers need to be set

-- call DOS interrupt 5: Print Screen
registers = dos_interrupt(#5, registers)
```

Example Program: demo\dos32\dosint.ex

See Also: allocate_low, free_low

----------<draw_line>----------

Platform: DOS32

Syntax:
```
include graphics.e
draw_line(i, s)
```

Description: Draw a line on a pixel-graphics screen connecting two or more points in s, using color i.

Example:

```
draw_line(WHITE, {{100, 100}, {200, 200}, {900, 700}})
```

```
             -- This would connect the three points in the sequence using
             -- a white line, i.e. a line would be drawn from {100, 100} to
             -- {200, 200} and another line would be drawn from {200, 200} to
             -- {900, 700}.


See Also:    polygon, ellipse, pixel




-----------------------------------<ellipse>-----------------------------------

Platform:    DOS32

Syntax:      include graphics.e
             ellipse(i1, i2, s1, s2)

Description: Draw an ellipse with color i1 on a pixel-graphics screen. The
             ellipse will neatly fit inside the rectangle defined by diagonal
             points s1 {x1, y1} and s2 {x2, y2}. If the rectangle is a square
             then the ellipse will be a circle. Fill the ellipse when i2 is 1.
             Don't fill when i2 is 0.

Example:

             ellipse(MAGENTA, 0, {10, 10}, {20, 20})

             -- This would make a magenta colored circle just fitting
             -- inside the square:
             --        {10, 10}, {10, 20}, {20, 20}, {20, 10}.


Example Program: demo\dos32\sb.ex

See Also:    polygon, draw_line


-----------------------------------<equal>-------------------------------------

Syntax:      i = equal(x1, x2)

Description: Compare two Euphoria objects to see if they are the same. Return
             1 (true) if they are the same. Return 0 (false) if they are
             different.

Comments:    This is equivalent to the expression: compare(x1, x2) = 0

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, equals operator (=)

----------<find>----------

Syntax: i = find(x, s)

Description: Find x as an element of s. If successful, return the index of the first element of s that matches. If unsuccessful return 0.

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: match, compare

----------<float32_to_atom>----------

Syntax:
```
include machine.e
a1 = float32_to_atom(s)
```

Description: Convert a sequence of 4 bytes to an atom. These 4 bytes must contain an IEEE floating-point number in 32-bit format.

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

Example:

```
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, atom_to_float32

---------------------------------<float64_to_atom>---------------------------------

Syntax: include machine.e
a1 = float64_to_atom(s)

Description: Convert a sequence of 8 bytes to an atom. These 8 bytes must contain an IEEE floating-point number in 64-bit format.

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

Example:

```
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, atom_to_float64

-----------------------------------<floor>-----------------------------------

Syntax: x2 = floor(x1)

Description: Return the greatest integer less than or equal to x1. (Round down to an integer.)

Comments: This function may be applied to an atom or to all elements of a sequence.

Example:

```
y = floor({0.5, -1.6, 9.99, 100})
-- y is {0, -2, 9, 100}
```

See Also: remainder

-----------------------------------<flush>-----------------------------------

Syntax: include file.e
flush(fn)

Description: 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.

Comments: 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 aren't ready to close the file yet.

Example:

```
f = open("logfile", "w")
puts(f, "Record#1\n")
puts(1, "Press Enter when ready\n")

flush(f)  -- This forces "Record #1" into "logfile" on disk.
          -- Without this, "logfile" will appear to have
          -- 0 characters when we stop for keyboard input.

s = gets(0) -- wait for keyboard input
```

See Also: close, lock_file

----------<free>----------

Syntax:
```
include machine.e
free(a)
```

Description: Free up a previously allocated block of memory by specifying the address of the start of the block, i.e. the address that was returned by allocate().

Comments: Use free() to recycle blocks of memory during execution. This will reduce the chance of running out of memory or getting into excessive virtual memory swapping to disk. Do not reference a block of memory that has been freed. When your program terminates, all allocated memory will be returned to the system.

Do not use free() to deallocate memory that was allocated using allocate_low(). Use free_low() for this purpose.

Example Program: demo\callmach.ex

See Also: allocate, free_low

----------<free_console>----------

Platform: WIN32, Linux, FreeBSD

Syntax:
```
include dll.e
free_console()
```

Description: 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 (similar to a DOS-prompt window). 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 Linux or FreeBSD, free_console() will set the terminal parameters back to normal, undoing the effect that curses has on the screen.

In a Linux or FreeBSD 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(), because it lets the interpreter know that there is no longer a console.

See Also: clear_screen, platform.doc

----------<free_low>----------

Platform: DOS32

Syntax:
```
include machine.e
free_low(i)
```

Description: Free up a previously allocated block of conventional memory by specifying the address of the start of the block, i.e. the address that was returned by allocate_low().

Comments: Use free_low() to recycle blocks of conventional memory during execution. This will reduce the chance of running out of conventional memory. Do not reference a block of memory that has been freed. When your program terminates, all allocated memory will be returned to the system.

Do not use free_low() to deallocate memory that was allocated using allocate(). Use free() for this purpose.

Example Program: demo\dos32\dosint.ex

See Also: allocate_low, dos_interrupt, free

<get>

Syntax: include get.e
s = get(fn)

Description: Input, from file fn, a human-readable string of characters representing a Euphoria object. Convert the string into the numeric value of that object. s will be a 2-element sequence: {error status, value}. Error status codes are:

```
GET_SUCCESS -- object was read successfully
GET_EOF     -- end of file before object was read
GET_FAIL    -- object is not syntactically correct
```

get() can read arbitrarily complicated Euphoria objects. You could have a long sequence of values in braces and separated by commas, e.g. {23, {49, 57}, 0.5, -1, 99, 'A', "john"}. A single call to get() will read in this entire sequence and return it's value as a result.

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.

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). Whitespace is not necessary within a top-level object. A call to get() will read one entire top-level object, plus one additional (whitespace) character.

Comments: 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().

The value returned is not meaningful unless you have a GET_SUCCESS status.

Example: Suppose your program asks the user to enter a number from the keyboard.

-- If he types 77.5, get(0) would return:

```
             {GET_SUCCESS, 77.5}

             -- whereas gets(0) would return:

             "77.5\n"


Example Program: demo\mydata.ex

See Also:    print, value, gets, getc, prompt_number, prompt_string


--------------------------------<get_active_page>--------------------------------

Platform:    DOS32

Syntax:      include image.e
             i = get_active_page()

Description: Some graphics modes on most video cards have multiple pages of
             memory. This lets you write screen output to one page while
             displaying a different page. get_active_page() returns the
             current page number that screen output is being sent to.

Comments:    The active and display pages are both 0 by default.

             video_config() will tell you how many pages are available in the
             current graphics mode.

See Also:    set_active_page, get_display_page, video_config


--------------------------------<get_all_palette>--------------------------------

Platform:    DOS32

Syntax:      include image.e
             s = get_all_palette()

Description: Retrieve color intensities for the entire set of colors in the
             current graphics mode. s is a sequence of the form:

             {{r,g,b}, {r,g,b}, ..., {r,g,b}}

             Each element specifies a color intensity {red, green, blue} for
             the corresponding color number, starting with color number 0. The
             values for red, green and blue will be in the range 0 to 63.

Comments:    This function might be used to get the palette values needed by
             save_bitmap(). Remember to multiply these values by 4 before
             calling save_bitmap(), since save_bitmap() expects values in the
             range 0 to 255.

See Also:    palette, all_palette, read_bitmap, save_bitmap, save_screen


---------------------------------<get_bytes>---------------------------------
```

Syntax:
```
include get.e
s = get_bytes(fn, i)
```

Description: Read the next i bytes from file number fn. Return the bytes as a sequence. The sequence will be of length i, except when there are fewer than i bytes remaining to be read in the file.

Comments: When i > 0 and length(s) < i you know you've reached the end of file. Eventually, an empty sequence will be returned for s.

This function is normally used with files opened in binary mode, "rb". This avoids the confusing situation in text mode where DOS will convert CR LF pairs to LF.

Example:

```
include get.e

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_display_page>

Platform: DOS32

Syntax:
```
include image.e
i = get_display_page()
```

Description: Some graphics modes on most video cards have multiple pages of memory. This lets you write screen output to one page while displaying another. get_display_page() returns the current page number that is being displayed on the monitor.

Comments: The active and display pages are both 0 by default.

video_config() will tell you how many pages are available in the

current graphics mode.

See Also: set_display_page, get_active_page, video_config

----------<get_key>----------

Syntax: i = get_key()

Description: Return the key that was pressed by the user, without waiting. Return -1 if no key was pressed. Special codes are returned for the function keys, arrow keys etc.

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

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

See Also: wait_key, getc

----------<get_mouse>----------

Platform: DOS32, Linux

Syntax: include mouse.e
x1 = get_mouse()

Description: Return the last mouse event in the form: {event, x, y} or return -1 if there has not been a mouse event since the last time get_mouse() was called.

Constants have been defined in mouse.e for the possible mouse events:

```
global 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. get_mouse() returns immediately with either a -1 or a mouse event. It does not wait for an event to occur. 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.

Comments: In pixel-graphics modes that are 320 pixels wide, you need to divide the x value by 2 to get the correct position on the screen. (A strange feature of DOS.)

In DOS32 text modes you need to scale the x and y coordinates to get line and column positions. 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 DOS32, you need a DOS mouse driver to use this routine. In Linux, GPM Server must be running.

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.

You can use get_mouse() in most text and pixel-graphics modes.

The first call that you make to get_mouse() will turn on a mouse pointer, or a highlighted character.

DOS generally does not support the use of a mouse in SVGA graphics modes (beyond 640x480 pixels). This restriction has been removed in Windows 95 (DOS 7.0). Graeme Burke, Peter Blue and others have contributed mouse routines that get around the problems with using a mouse in SVGA. See the Euphoria Archive Web page.

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. Test this if you are trying to read the pixel color using get_pixel(). You may have to read x-1,y-1 instead.

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:

```
object event

while 1 do
    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, and_bits
```

----------<get_pixel>----------

```
Platform:   DOS32

Syntax:     x = get_pixel(s)

Description: When s is a 2-element screen coordinate {x, y}, get_pixel()
            returns the color (a small integer) of the pixel on the
            pixel-graphics screen at that point.

            When s is a 3-element sequence of the form: {x, y, n} get_pixel()
            returns a sequence of n colors for the points starting at {x, y}
            and moving to the right {x+1, y}, {x+2, y} etc.

            Points off the screen have unpredictable color values.

Comments:   When n is specified, a very fast algorithm is used to read the
            pixel colors on the screen. It is much faster to call get_pixel()
            once, specifying a large value of n, than it is to call it many
            times, reading one pixel color at a time.

Example:

            object x

            x = get_pixel({30,40})
            -- x is set to the color value of point x=30, y=40

            x = get_pixel({30,40,100})
            -- x is set to a sequence of 100 integer values, representing
            -- the colors starting at {30,40} and going to the right


See Also:   pixel, graphics_mode, get_position
```

----------<get_position>----------

```
Syntax:     include graphics.e
            s = get_position()

Description: Return the current line and column position of the cursor as a
            2-element sequence {line, column}.

Comments:   get_position() works in both text and pixel-graphics modes. In
            pixel-graphics modes no cursor will be displayed, but
            get_position() will return the line and column where the next
            character will be displayed.

            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.

See Also: position, get_pixel

<get_screen_char>

Platform: DOS32, Linux, FreeBSD

Syntax:
```
include image.e
s = get_screen_char(i1, i2)
```

Description: Return a 2-element sequence s, of the form {ascii-code, attributes} for the character on the screen at line i1, column i2. s consists of two atoms. The first is the ASCII code for the character. The second is an atom that contains the foreground and background color of the character, and possibly other information describing the appearance of the character on the screen.

Comments: With get_screen_char() and put_screen_char() you can save and restore a character on the screen along with its attributes.

Example:

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

See Also: put_screen_char, save_text_image

<get_vector>

Platform: DOS32

Syntax:
```
include machine.e
s = get_vector(i)
```

Description: Return the current protected mode far address of the handler for interrupt number i. s will be a 2-element sequence: {16-bit segment, 32-bit offset}.

Example:

```
s = get_vector(#1C)
-- s will be set to the far address of the clock tick
-- interrupt handler, for example: {59, 808}
```

Example Program: demo\dos32\hardint.ex

See Also: set_vector, lock_memory

----------<getc>----------

Syntax: i = getc(fn)

Description: Get the next character (byte) from file or device fn. The character will have a value from 0 to 255. -1 is returned at end of file.

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 control-Z, which the operating system treats as "end of file". -1 will be returned.

See Also: gets, get_key, wait_key, open

----------<getenv>----------

Syntax: x = getenv(s)

Description: Return the value of an environment variable. If the variable is undefined, return -1.

Comments: Because either a sequence or an atom (-1) might be returned, you should probably assign the result to a variable declared as object.

Example:

```
e = getenv("EUDIR")
-- e will be "C:\EUPHORIA" -- or perhaps D:, E: etc.
```

See Also: command_line

----------<gets>----------

Syntax: x = gets(fn)

Description: Get the next sequence (one line, including '\n') of characters from file or device fn. The characters will have values from 0 to 255. The atom -1 is returned on end of file.

Comments: Because either a sequence or an atom (-1) might be returned, you should probably assign the result to a variable declared as object.

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.

The last line in a file might not end with a new-line '\n' character.

When your program reads from the keyboard, the user can type control-Z, which the operating system treats as "end of file". -1 will be returned.

In SVGA modes, DOS might set the wrong cursor position, after a call to gets(0) to read the keyboard. You should set it yourself using position().

Example 1:

```
sequence buffer
object line
integer fn

-- read a text file into a sequence
fn = open("myfile.txt", "r")
if fn = -1 then
    puts(1, "Couldn't open myfile.txt\n")
    abort(1)
end if

buffer = {}
while 1 do
    line = gets(fn)
    if atom(line) then
        exit   -- -1 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..length(line)-1] -- get rid of \n character at end
puts(1, '\n')   -- necessary
puts(1, line & " is a nice name.\n")
```

See Also: getc, puts, open

----------<graphics_mode>----------

Platform: DOS32

Syntax: include graphics.e
i1 = graphics_mode(i2)

Description: Select graphics mode i2. See graphics.e for a list of valid graphics modes. If successful, i1 is set to 0, otherwise i1 is set to 1.

Comments: Some modes are referred to as text modes because they only let you display text. Other modes are referred to as pixel-graphics modes because you can display pixels, lines, ellipses etc., as well as text.

As a convenience to your users, it is usually a good idea to switch back from a pixel-graphics mode to the standard text mode before your program terminates. You can do this with graphics_mode(-1). If a pixel-graphics program leaves your screen in a mess, you can clear it up with the DOS CLS command, or by running ex or ed.

Some graphics cards will be unable to enter some SVGA modes, under some conditions. You can't always tell from the i1 value, whether the graphics mode was set up successfully.

On the WIN32 and Linux/FreeBSD platforms, graphics_mode() will allocate a plain, text mode console if one does not exist yet. It will then return 0, no matter what value is passed as i2.

Example:

```
if graphics_mode(18) then
    puts(SCREEN, "need VGA graphics!\n")
    abort(1)
end if
draw_line(BLUE, {{0,0}, {50,50}})
```

See Also: text_rows, video_config

<instance>

Platform: WIN32

Syntax: include misc.e
i = instance()

Description: Return a handle to the current program.

Comments: This handle value can be passed to various Windows routines to get information about the current program that is running, i.e. your program. Each time a user starts up your program, a different instance will be created.

In C, this is the first parameter to WinMain().

On DOS32 and Linux/FreeBSD, instance() always returns 0.

See Also: platform.doc

<int_to_bits>

Syntax:
```
include machine.e
s = int_to_bits(a, i)
```

Description: Return the low-order i bits of a, as a sequence of 1's and 0's. The least significant bits come first. For negative numbers the two's complement bit pattern is returned.

Comments: 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:

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

See Also: bits_to_int, and_bits, or_bits, xor_bits, not_bits, operations on sequences

---

<int_to_bytes>

Syntax:
```
include machine.e
s = int_to_bytes(a)
```

Description: Convert an integer into a sequence of 4 bytes. These bytes are in the order expected on the 386+, i.e. least-significant byte first.

Comments: You might use this routine prior to poking the 4 bytes into memory for use by a machine language program.

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+.

This function will correctly convert integer values up to 32-bits. For larger values, only the low-order 32-bits are converted. Euphoria's integer type only allows values up to 31-bits, so declare your variables as atom if you need a larger range.

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, bits_to_int, peek, poke, poke4

----------<integer>----------

Syntax: i = integer(x)

Description: Return 1 if x is an integer in the range -1073741824 to +1073741823. Otherwise return 0.

Comments: This serves to define the integer type. You can also call it like an ordinary function to determine if an object is an integer.

Example 1:

```
integer z
z = -1
```

Example 2:

```
if integer(y/x) then
    puts(SCREEN, "y is an exact multiple of x")
end if
```

See Also: atom, sequence, floor

----------<length>----------

Syntax: i = length(s)

Description: Return the length of s. s must be a sequence. An error will occur if s is an atom.

Comments: The length of each sequence is stored internally by the interpreter for quick access. (In other languages this operation requires a search through memory for an end marker.)

Example 1:

```
length({{1,2}, {3,4}, {5,6}})   -- 3
```

Example 2:

```
length("")    -- 0
```

Example 3:

```
            length({})    -- 0
```

See Also: sequence

----------<lock_file>----------

```
Syntax:      include file.e
             i1 = lock_file(fn, i2, s)
```

Description: 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.

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 Linux/FreeBSD, there are two types of locks that you can request using the i2 parameter. (Under DOS32 and WIN32 the i2 parameter 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. file.e contains the following declaration:

```
global constant LOCK_SHARED = 1,
         LOCK_EXCLUSIVE = 2
```

On DOS32 and WIN32 you can lock a specified portion of a file using the s parameter. s 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. In the current release for Linux/FreeBSD, locks always apply to the whole file, and you should specify {} for this parameter.

If it is successful in obtaining the desired lock, lock_file() will return 1. If unsuccessful, it will return 0. 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.

Comments: On Linux/FreeBSD, 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 and DOS32, locks are enforced by the operating system.

On DOS32, lock_file() is more useful when file sharing is

enabled. It will typically return 0 (unsuccessful) under plain MS-DOS, outside of Windows.

Example:

```
include misc.e
include file.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(1, "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, {})
close(v)
```

See Also: unlock_file, flush, sleep

----------<lock_memory>----------

Platform: DOS32

Syntax:
```
include machine.e
lock_memory(a, i)
```

Description: Prevent the block of virtual memory starting at address a, of length i, from ever being swapped out to disk.

Comments:
lock_memory() should only be used in the highly-specialized situation where you have set up your own DOS hardware interrupt handler using machine code. When a hardware interrupt occurs, it is not possible for the operating system to retrieve any code or data that has been swapped out, so you need to protect any blocks of machine code or data that will be needed in servicing the interrupt.

Example Program: demo\dos32\hardint.ex

See Also: get_vector, set_vector

----------<log>----------

Syntax:
```
x2 = log(x1)
```

Description: Return the natural logarithm of x1.

Comments: This function may be applied to an atom or to all elements of a sequence. Note that log is only defined for positive numbers.

Your program will abort with a message if you try to take the log of a negative number or zero.

Example:

```
a = log(100)
-- a is 4.60517
```

See Also: sin, cos, tan, sqrt

----------<lower>----------

Syntax:
```
include wildcard.e
x2 = lower(x1)
```

Description: Convert an atom or sequence to lower case.

Example:

```
s = lower("Euphoria")
-- s is "euphoria"

a = lower('B')
-- a is 'b'

s = lower({"Euphoria", "Programming"})
-- s is {"euphoria", "programming"}
```

See Also: upper

----------<machine_func>----------

Syntax:
```
x1 = machine_func(a, x)
```

Description: see machine_proc() below

----------<machine_proc>----------

Syntax:
```
machine_proc(a, x)
```

Description: Perform a machine-specific operation such as graphics and sound effects. This routine should normally be called indirectly via one of the library routines in a Euphoria include file. A direct call might cause a machine exception if done incorrectly.

See Also: machine_func

----------<match>----------

Syntax:
```
i = match(s1, s2)
```

Description: Try to match s1 against some slice of s2. If successful, return the element number of s2 where the (first) matching slice begins, else return 0.

Example:

```
location = match("pho", "Euphoria")
-- location is set to 3
```

See Also: find, compare, wildcard_match

----------<mem_copy>----------

Syntax: mem_copy(a1, a2, i)

Description: Copy a block of i bytes of memory from address a2 to address a1.

Comments: The bytes of memory will be copied correctly even if the block of memory at a2 overlaps with the block of memory at a1.

mem_copy(a1, a2, i) is equivalent to: poke(a1, peek({a2, i})) but is much faster.

Example:

```
dest = allocate(50)
src = allocate(100)
poke(src, {1,2,3,4,5,6,7,8,9})
mem_copy(dest, src, 9)
```

See Also: mem_set, peek, poke, allocate, allocate_low

----------<mem_set>----------

Syntax: mem_set(a1, i1, i2)

Description: Set i2 bytes of memory, starting at address a1, to the value of i1.

Comments: The low order 8 bits of i1 are actually stored in each byte.

mem_set(a1, i1, i2) is equivalent to: poke(a1, repeat(i1, i2)) but is much faster.

Example:

```
destination = allocate(1000)
mem_set(destination, ' ', 1000)
-- 1000 consecutive bytes in memory will be set to 32
-- (the ASCII code for ' ')
```

See Also: mem_copy, peek, poke, allocate, allocate_low

---------------------------------<message_box>---------------------------------

Platform: WIN32

Syntax: include msgbox.e
             i = message_box(s1, s2, x)

Description: Display a window with title s2, containing the message string s1.
             x determines the combination of buttons that will be available
             for the user to press, plus some other characteristics. x can be
             an atom or a sequence. A return value of 0 indicates a failure to
             set up the window.

Comments: See msgbox.e for a complete list of possible values for x and i.

Example:

```
             response = message_box("Do you wish to proceed?",
                                    "My Application",
                                    MB_YESNOCANCEL)
             if response = IDCANCEL or response = IDNO then
                 abort(1)
             end if
```

Example Program: demo\win32\email.exw

---------------------------------<mouse_events>---------------------------------

Platform: DOS32, Linux

Syntax: include mouse.e
             mouse_events(i)

Description: Use this procedure to select the mouse events that you want
             get_mouse() to report. 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 Linux, mouse_events() currently has no effect.

Comments: 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:

```
             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

---------------------------------<mouse_pointer>---------------------------------

Platform: DOS32, Linux

Syntax: include mouse.e
mouse_pointer(i)

Description: If i is 0 hide the mouse pointer, otherwise turn on the mouse pointer. 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

Comments: 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_events

---------------------------------<not_bits>---------------------------------

Syntax: x2 = not_bits(x1)

Description: Perform the logical NOT operation on each bit in x1. A bit in x2 will be 1 when the corresponding bit in x1 is 0, and will be 0 when the corresponding bit in x1 is 1.

Comments: The argument to this function may be an atom or a sequence. The rules for operations on sequences apply.

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.

Example:

```
a = not_bits(#000000F7)
-- a is -248 (i.e. FFFFFF08 interpreted as a negative number)
```

See Also: and_bits, or_bits, xor_bits, int_to_bits

---------------------------------<object>---------------------------------

Syntax: i = object(x)

Description: Test if x is of type object. This will always be true, so object() will always return 1.

Comments: All predefined and user-defined types can also be used as functions to test if a value belongs to the type. object() is included just for completeness. It always returns 1.

Example:

```
? object({1,2,3})  -- always prints 1
```

See Also: integer, atom, sequence

----------<open>----------

Syntax: fn = open(st1, st2)

Description: Open a file or device, to get the file number. -1 is returned if the open fails. st1 is the path name of the file or device. st2 is the mode in which the file is to be opened. 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.

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.

Some typical devices that you can open 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

Comments: DOS32: When running under Windows 95 or later, you can open any existing file that has a long file or directory name in its path (i.e. greater than the standard DOS 8.3 format) using any open mode - read, write etc. However, if you try to create a new file (open with "w" or "a" and the file does not already exist) then the name will be truncated if necessary to an 8.3 style name. We hope to support creation of new long-filename files in a future release.

WIN32, Linux and FreeBSD: Long filenames are fully supported for reading and writing and creating.

Example:

```
integer file_num, file_num95
sequence first_line
constant ERROR = 2

file_num = open("myfile", "r")
if file_num = -1 then
    puts(ERROR, "couldn't open myfile\n")
else
    first_line = gets(file_num)
end if

file_num = open("PRN", "w") -- open printer for output

-- on Windows 95:
file_num95 = open("bigdirectoryname\\verylongfilename.abcdefg",
                  "r")
if file_num95 != -1 then
    puts(1, "it worked!\n")
end if
```

See Also: close

<open_dll>

Platform: WIN32, Linux, FreeBSD

Syntax:
```
include dll.e
a = open_dll(st)
```

Description: Open a Windows dynamic link library (.dll) file, or a Linux or FreeBSD shared library (.so) file. A 32-bit address will be returned, or 0 if the .dll can't be found. st can be a relative or an absolute file name. Windows will use the normal search path for locating .dll files.

Comments: 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 for you automatically at the end of execution.

Example:

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

See Also: define_c_func, define_c_proc, define_c_var, c_func, c_proc, platform.doc

----------<or_bits>----------

Syntax: x3 = or_bits(x1, x2)

Description: Perform the logical OR operation on corresponding bits in x1 and x2. A bit in x3 will be 1 when a corresponding bit in either x1 or x2 is 1.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply.

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

----------<palette>----------

Platform: DOS32

Syntax:
```
include graphics.e
x = palette(i, s)
```

Description: Change the color for color number i to s, where s is a sequence of color intensities: {red, green, blue}. Each value in s can be from 0 to 63. If successful, a 3-element sequence containing the previous color for i will be returned, and all pixels on the screen with value i will be set to the new color. If unsuccessful, the atom -1 will be returned.

Example:

```
x = palette(0, {15, 40, 10})
-- color number 0 (normally black) is changed to a shade
-- of mainly green.
```

See Also: all_palette

----------<peek>----------

Syntax:
```
i = peek(a)
or ...
s = peek({a, i})
```

Description: Return a single byte value in the range 0 to 255 from machine address a, or return a sequence containing i consecutive byte values starting at address a in memory.

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.

Remember that peek takes just one argument, which in the second form is actually a 2-element sequence.

Example: The following are equivalent:

```
-- method 1
s = {peek(100), peek(101), peek(102), peek(103)}

-- method 2
s = peek({100, 4})
```

See Also: poke, peek4s, peek4u, allocate, free, allocate_low, free_low, call

----------<peek4s>----------

Syntax: a2 = peek4s(a1)
or ...
s = peek4s({a1, i})

Description: Return a 4-byte (32-bit) signed value in the range -2147483648 to +2147483647 from machine address a1, or return a sequence containing i consecutive 4-byte signed values starting at address a1 in memory.

Comments: The 32-bit values returned by peek4s() may be too large for the Euphoria integer type (31-bits), so you should use atom variables.

Since machine addresses are 32-bit numbers, they can also be too large for Euphoria's integer type. Variables that hold an address should therefore be declared as atoms.

It is faster to read several 4-byte values at once using the second form of peek4s() than it is to read one 4-byte value at a time in a loop.

Remember that peek4s() takes just one argument, which in the second form is actually a 2-element sequence.

Example: The following are equivalent:

```
-- method 1
s = {peek4s(100), peek4s(104), peek4s(108), peek4s(112)}

-- method 2
s = peek4s({100, 4})
```

See Also: peek4u, peek, poke4, allocate, free, allocate_low, free_low, call

----------<peek4u>----------

Syntax: a2 = peek4u(a1)
or ...
s = peek4u({a1, i})

Description: Return a 4-byte (32-bit) unsigned value in the range 0 to 4294967295 from machine address a1, or return a sequence containing i consecutive 4-byte unsigned values starting at address a1 in memory.

Comments: The 32-bit values returned by peek4u() may be too large for the Euphoria integer type (31-bits), so you should use atom variables.

Since machine addresses are 32-bit numbers, they can also be too

large for Euphoria's integer type. Variables that hold an address should therefore be declared as atoms.

It is faster to read several 4-byte values at once using the second form of peek4u() than it is to read one 4-byte value at a time in a loop.

Remember that peek4u() takes just one argument, which in the second form is actually a 2-element sequence.

Example: The following are equivalent:

```
-- method 1
s = {peek4u(100), peek4u(104), peek4u(108), peek4u(112)}

-- method 2
s = peek4u({100, 4})
```

See Also: peek4s, peek, poke4, allocate, free, allocate_low, free_low, call

----------<PI>----------

Syntax: include misc.e
PI

Description: PI (3.14159...) has been defined as a global constant.

Comments: Enough digits have been used to attain the maximum accuracy possible for a Euphoria atom.

Example:

```
x = PI  -- x is 3.14159...
```

See Also: sin, cos, tan

----------<pixel>----------

Platform: DOS32

Syntax: pixel(x1, s)

Description: Set one or more pixels on a pixel-graphics screen starting at point s, where s is a 2-element screen coordinate {x, y}. If x1 is an atom, one pixel will be set to the color indicated by x1. If x1 is a sequence then a number of pixels will be set, starting at s and moving to the right (increasing x value, same y value).

Comments: When x1 is a sequence, a very fast algorithm is used to put the pixels on the screen. It is much faster to call pixel() once, with a sequence of pixel colors, than it is to call it many times, plotting one pixel color at a time.

In graphics mode 19, pixel() is highly optimized.

Any off-screen pixels will be safely clipped.

Example 1:

```
pixel(BLUE, {50, 60})
-- the point {50,60} is set to the color BLUE
```

Example 2:

```
pixel({BLUE, GREEN, WHITE, RED}, {50,60})
-- {50,60} set to BLUE
-- {51,60} set to GREEN
-- {52,60} set to WHITE
-- {53,60} set to RED
```

See Also: get_pixel, graphics_mode

----------<platform>----------

Syntax: i = platform()

Description: platform() is a function built-in to the interpreter. It indicates the platform that the program is being executed on: DOS32, WIN32, Linux or FreeBSD.

Comments: When ex.exe is running, the platform is DOS32. When exw.exe is running the platform is WIN32. When exu is running the platform is LINUX (or FREEBSD).

The include file misc.e contains the following constants:

```
global constant DOS32 = 1,
                WIN32 = 2,
                LINUX = 3,
                FREEBSD = 3
```

Use platform() when you want 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.

The call to platform() costs nothing. It is optimized at compile-time into the appropriate integer value: 1, 2 or 3.

Example 1:

```
if platform() = WIN32 then
    -- call system Beep routine
    err = c_func(Beep, {0,0})
elsif platform() = DOS32 then
```

```
            -- make beep
            sound(500)
            t = time()
            while time() < t + 0.5 do
            end while
            sound(0)
        else
            -- do nothing (Linux/FreeBSD)
        end if
```

See Also: platform.doc

----------<poke>----------

Syntax: poke(a, x)

Description: If x is an atom, write a single byte value to memory address a.

If x is a sequence, write a sequence of byte values to consecutive memory locations starting at location a.

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:

```
a = allocate(100)   -- allocate 100 bytes in memory

-- poke one byte at a time:
poke(a, 97)
poke(a+1, 98)
poke(a+2, 99)

-- poke 3 bytes at once:
poke(a, {97, 98, 99})
```

Example Program: demo\callmach.ex

See Also: peek, poke4, allocate, free, allocate_low, free_low, call, safe.e

----------<poke4>----------

Syntax: poke4(a, x)

Description: If x is an atom, write a 4-byte (32-bit) value to memory address

a.

If x is a sequence, write a sequence of 4-byte values to consecutive memory locations starting at location a.

Comments: The value or values to be stored must not exceed 32-bits in size.

It is faster to write several 4-byte values at once by poking a sequence of values, than it is to write one 4-byte value at a time in a loop.

The 4-byte values to be stored can be negative or positive. You can read them back with either peek4s() or peek4u().

Example:

```
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: peek4u, peek4s, poke, allocate, allocate_low, call

----------<polygon>----------

Platform: DOS32

Syntax:
```
include graphics.e
polygon(i1, i2, s)
```

Description: Draw a polygon with 3 or more vertices given in s, on a pixel-graphics screen using a certain color i1. Fill the area if i2 is 1. Don't fill if i2 is 0.

Example:

```
polygon(GREEN, 1, {{100, 100}, {200, 200}, {900, 700}})
-- makes a solid green triangle.
```

See Also: draw_line, ellipse

----------<position>----------

Syntax:
```
position(i1, i2)
```

Description: Set the cursor to line i1, column i2, 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.

Comments: position() works in both text and pixel-graphics modes.

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. position() only sets the line and column for the text that you display, not the pixels that you plot. There is no corresponding routine for setting the next pixel position.

Example:

```
position(2,1)
-- the cursor moves to the beginning of the second line from
-- the top
```

See Also: get_position, puts, print, printf

----------<power>----------

Syntax: x3 = power(x1, x2)

Description: Raise x1 to the power x2

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.

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, sqrt


-----------------------------------<prepend>-----------------------------------

Syntax:      s2 = prepend(s1, x)

Description: Prepend x to the start of sequence s1. The length of s2 will be
             length(s1) + 1.

Comments:    If x is an atom this is the same as s2 = x & s1. If x is a
             sequence it is definitely not the same.

             The case where s1 and s2 are the same variable 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, concatenation operator &, sequence-formation operator


--------------------------------<pretty_print>--------------------------------

Syntax:      include misc.e
             pretty_print(fn, x, s)

Description: Print, to file or device fn, an object x, using braces { , , , },
             indentation, and multiple lines to show the structure.

             Several options may be supplied in s to control the presentation.
             Pass {} 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
             [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

If the length of s is less than 8, 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.
```

Comments: 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 \n 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.

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, puts, printf

----------<print>----------

Syntax: print(fn, x)

Description: Print, to file or device fn, an object x with braces { , , , } to show the structure.

Example 1:

```
print(1, "ABC")  -- output is:  {65, 66, 67}
puts(1, "ABC")   -- output is:  ABC
```

Example 2:

```
print(1, repeat({10,20}, 3))
-- output is: {{10,20},{10,20},{10,20}}
```

See Also: ?, pretty_print, puts, printf, get

----------<printf>----------

Syntax: printf(fn, st, x)

Description: Print x, to file or device fn, using format string st. If x is an atom then a single value will be printed. If x is a sequence, then formats from st are applied to successive elements of x. Thus printf() always takes exactly 3 arguments. Only the length of the last argument, containing the values to be printed, will vary. The basic formats are:

```
%d - print an atom as a decimal integer
%x - print an atom as a hexadecimal integer
%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 either the %f or %e format, whichever seems more appropriate
%% - print the '%' character itself

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(1, "%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(1, "%s", {name})   -- correct
```

Now, the third argument of printf() is a one-element sequence containing the item to be formatted.

Example 1:

```
rate = 7.875
printf(myfile, "The interest rate is: %8.2f\n", rate)
```

```
      The interest rate is:     7.88
```

Example 2:

```
name="John Smith"
score=97
printf(1, "%15s, %5d\n", {name, score})
```

```
      John Smith,    97
```

Example 3:

```
printf(1, "%-10.4s $ %s", {"ABCDEFGHIJKLMNOP", "XXX"})
```

```
                ABCD       $ XXX
```

See Also: sprintf, puts, open

----------<profile>----------

Syntax: profile(i)

Description: Enable or disable profiling at run-time. This works for both execution-count and time-profiling. If i is 1 then profiling will be enabled, and samples/counts will be recorded. If i is 0 then profiling will be disabled and samples/counts will not be recorded.

Comments: After a "with profile" or "with profile_time" statement, profiling is turned on automatically. Use profile(0) to turn it off. Use profile(1) to turn it back on when execution reaches the code that you wish to focus the profile on.

Example 1:

```
with profile_time
profile(0)
  ...
procedure slow_routine()
profile(1)
  ...
profile(0)
end procedure
```

See Also: trace, profiling, special top-level statements

----------<prompt_number>----------

Syntax:
```
include get.e
a = prompt_number(st, s)
```

Description: Prompt the user to enter a number. st is a string of text that will be displayed on the screen. s is a sequence of two values {lower, upper} which determine the range of values that the user may enter. If the user enters a number that is less than lower or greater than upper, he will be prompted again. s can be empty, {}, if there are no restrictions.

Comments: 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: get, prompt_string

<prompt_string>

Syntax:
```
include get.e
s = prompt_string(st)
```

Description: Prompt the user to enter a string of text. st is a string that will be displayed on the screen. The string that the user types will be returned as a sequence, minus any new-line character.

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

Example:

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

See Also: gets, prompt_number

<put_screen_char>

Platform: DOS32, Linux, FreeBSD

Syntax:
```
include image.e
put_screen_char(i1, i2, s)
```

Description: Write zero or more characters onto the screen along with their attributes. i1 specifies the line, and i2 specifies the column where the first character should be written. The sequence s looks like: {ascii-code1, attribute1, ascii-code2, attribute2, ...}. Each pair of elements in s describes one character. The ascii-code atom contains the ASCII code of the character. The attributes atom contains the foreground color, background color, and possibly other platform-dependent information controlling how the character is displayed on the screen.

Comments: The length of s must be a multiple of 2. If s has 0 length, nothing will be written to the screen.

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:

```
-- 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

----------<puts>----------

Syntax: puts(fn, x)

Description: Output, to file or device fn, a single byte (atom) or sequence of bytes. The low order 8-bits of each value is actually sent out. If fn is the screen you will see text characters displayed.

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 may get truncated.

Example 1:

```
puts(SCREEN, "Enter your first name: ")
```

Example 2:

```
puts(output, 'A')  -- the single byte 65 will be sent to output
```

See Also: printf, gets, open

----------<rand>----------

Syntax: x2 = rand(x1)

Description: Return a random integer from 1 to x1, where x1 may be from 1 to the largest positive value of type integer (1073741823).

Comments: This function may be applied to an atom or to all elements of a sequence.

Example:

```
s = rand({10, 20, 30})
-- s might be: {5, 17, 23} or {9, 3, 12} etc.
```

See Also: set_rand

----------<read_bitmap>----------

Syntax:

```
include image.e
x = read_bitmap(st)
```

Description: st is the name of a .bmp "bitmap" file. The file should be in the

bitmap format. The most common variations of the format are supported. If the file is read successfully the result will be a 2-element sequence. The first element is the palette, containing intensity values in the range 0 to 255. The second element is a 2-d sequence of sequences containing a pixel-graphics image. You can pass the palette to all_palette() (after dividing it by 4 to scale it). The image can be passed to display_image().

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:

```
global constant BMP_OPEN_FAILED = 1,
                BMP_UNEXPECTED_EOF = 2,
          BMP_UNSUPPORTED_FORMAT = 3
```

Comments: 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:

```
x = read_bitmap("c:\\windows\\arcade.bmp")
-- note: double backslash needed to get single backslash in
-- a string
```

Example Program: demo\dos32\bitmap.ex

See Also: palette, all_palette, display_image, save_bitmap

<register_block>

Syntax: include machine.e (or safe.e)
register_block(a, i)

Description: Add a block of memory to the list of safe blocks maintained by safe.e (the debug version of machine.e). The block starts at address a. The length of the block is i bytes.

Comments: 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() or allocate_low() routines, and which you have not yet freed using Euphoria's free() or free_low(). 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().

When you include machine.e, you'll get different versions of register_block() and unregister_block() that do nothing. This makes it easy to switch back and forth between debug and non-debug runs of your program.

Example 1:

```
atom addr

addr = c_func(x, {})
register_block(addr, 5)
poke(addr, "ABCDE")
unregister_block(addr)
```

See Also: unregister_block, safe.e

----------<remainder>----------

Syntax: x3 = remainder(x1, x2)

Description: Compute the remainder after dividing x1 by x2. The result will have the same sign as x1, and the magnitude of the result will be less than the magnitude of x2.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply.

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: floor

----------<repeat>----------

Syntax: s = repeat(x, a)

Description: Create a sequence of length a where each element is x.

```
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}


Example 2:

             repeat("JOHN", 4)  -- {"JOHN", "JOHN", "JOHN", "JOHN"}
             -- The interpreter will create only one copy of "JOHN"
             -- in memory


See Also:    append, prepend, sequence-formation operator


-----------------------------------<reverse>-----------------------------------

Syntax:      include misc.e
             s2 = reverse(s1)

Description: Reverse the order of elements in a sequence.

Comments:    A new sequence is created where the top-level elements appear in
             reverse order compared to the original sequence.

Example 1:

             reverse({1,3,5,7})          -- {7,5,3,1}


Example 2:

             reverse({{1,2,3}, {4,5,6}}) -- {{4,5,6}, {1,2,3}}


Example 3:

             reverse({99})               -- {99}


Example 4:

             reverse({})                 -- {}


See Also:    append, prepend, repeat


---------------------------------<routine_id>---------------------------------

Syntax:      i = routine_id(st)

Description: Return an integer id number, known as a routine id, for a
```

user-defined Euphoria procedure or function. The name of the procedure or function is given by the string sequence st. -1 is returned if the named routine can't be found.

Comments: The id number can be passed to call_proc() or call_func(), to indirectly call the routine named by st.

The routine named by st must be visible, i.e. callable, at the place where routine_id() is used to get the id number. 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().

Some typical uses of routine_id() are:

1. Calling a routine that is defined later in a program.
2. Creating a subroutine that takes another routine as a parameter. (See Example 2 below)
3. Using a sequence of routine id's to make a case (switch) statement.
4. Setting up an Object-Oriented system.
5. Getting a routine id so you can pass it to call_back(). (See platform.doc)

Note that C routines, callable by Euphoria, also have routine id's. 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, platform.doc

---

<save_bitmap>

Syntax:

```
include image.e
i = save_bitmap(s, st)
```

Description: Create a bitmap (.bmp) file from a 2-element sequence s. st is the name of a .bmp "bitmap" file. s[1] is the palette:

```
{{r,g,b}, {r,g,b}, ..., {r,g,b}}
```

Each red, green, or blue value is in the range 0 to 255. s[2] is a 2-d sequence of sequences containing a pixel-graphics image. The sequences contained in s[2] must all have the same length. s is in the same format as the value returned by read_bitmap().

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
                BMP_OPEN_FAILED = 1,
               BMP_INVALID_MODE = 4 -- invalid graphics mode
                                    -- or invalid argument
```

Comments: If you use get_all_palette() to get the palette before calling this function, you must multiply the returned intensity values by 4 before calling save_bitmap().

You might use save_image() to get the 2-d image for s[2].

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:

```
paletteData = get_all_palette() * 4
code = save_bitmap({paletteData, imageData},
                   "c:\\example\\a1.bmp")
```

See Also: save_image, read_bitmap, save_screen, get_all_palette

---------------------------------<save_image>---------------------------------

Platform: DOS32

Syntax: include image.e
s3 = save_image(s1, s2)

Description: Save a rectangular image from a pixel-graphics screen. The result is a 2-d sequence of sequences containing all the pixels in the image. You can redisplay the image using display_image(). s1 is a 2-element sequence {x1,y1} specifying the top-left pixel in the image. s2 is a sequence {x2,y2} specifying the bottom-right pixel.

Example:

```
s = save_image({0,0}, {50,50})
display_image({100,200}, s)
display_image({300,400}, s)
-- saves a 51x51 square image, then redisplays it at {100,200}
-- and at {300,400}
```

See Also: display_image, save_text_image

---------------------------------<save_screen>---------------------------------

Platform: DOS32

Syntax: include image.e
i = save_screen(x1, st)

Description: Save the whole screen or a rectangular region of the screen as a Windows bitmap (.bmp) file. To save the whole screen, pass the integer 0 for x1. To save a rectangular region of the screen, x1 should be a sequence of 2 sequences: {{topLeftXPixel, topLeftYPixel}, {bottomRightXPixel, bottomRightYPixel}}

st is the name of a .bmp "bitmap" file.

The result will be one of the following codes:

```
global constant BMP_SUCCESS = 0,
                BMP_OPEN_FAILED = 1,
               BMP_INVALID_MODE = 4 -- invalid graphics mode
                                    -- or invalid argument
```

Comments: save_screen() 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.

save_screen() only works in pixel-graphics modes, not text modes.

Example 1:

```
            -- save whole screen:
            code = save_screen(0, "c:\\example\\a1.bmp")
```

Example 2:

```
            -- save part of screen:
            err = save_screen({{0,0},{200, 15}}, "b1.bmp")
```

See Also: save_image, read_bitmap, save_bitmap

<save_text_image>

Platform: DOS32, Linux, FreeBSD

Syntax:
```
include image.e
s3 = save_text_image(s1, s2)
```

Description: Save a rectangular region of text from a text-mode screen. The result is a sequence of sequences containing ASCII characters and attributes from the screen. You can redisplay this text using display_text_image(). s1 is a 2-element sequence {line1, column1} specifying the top-left character. s2 is a sequence {line2, column2} specifying the bottom right character.

Comments: Because the character attributes are also saved, you will get the correct foreground color, background color and other properties for each character when you redisplay the text.

On DOS32, an attribute byte is made up of two 4-bit fields that encode the foreground and background color of a character. The high-order 4 bits determine the background color, while the low-order 4 bits determine the foreground color.

This routine 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.

On DOS32, if you are flipping video pages, note that this function reads from the current active page.

Example: If the top 2 lines of the screen have:

```
   Hello
   World
```

And you execute:

```
s = save_text_image({1,1}, {2,5})
```

Then s is something like:

```
                {"H-e-l-l-o-",
                 "W-o-r-l-d-"}
            where '-' indicates the attribute bytes
```

See Also: display_text_image, save_image, set_active_page, get_screen_char

----------<scroll>----------

Syntax:
```
include graphics.e
scroll(i1, i2, i3)
```

Description: Scroll a region of text on the screen either up (i1 positive) or down (i1 negative) by i1 lines. The region is the series of lines on the screen from i2 (top line) to i3 (bottom line), inclusive. New blank lines will appear at the top or bottom.

Comments: 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 Program: bin\ed.ex

See Also: clear_screen, text_rows

----------<seek>----------

Syntax:
```
include file.e
i1 = seek(fn, i2)
```

Description: Seek (move) to any byte position in the file fn or to the end of file if i2 is -1. 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. The value returned by seek() is 0 if the seek was successful, and non-zero if it was unsuccessful. 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.

Comments: 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, DOS 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.

Example:

```
include file.e

integer fn
fn = open("mydata", "rb")
-- read and display first line of file 3 times:
for i = 1 to 3 do
    puts(1, gets(fn))
    if seek(fn, 0) then
        puts(1, "rewind failed!\n")
    end if
end for
```

See Also: where, open

----------<sequence>----------

Syntax: i = sequence(x)

Description: Return 1 if x is a sequence else return 0.

Comments: This serves to define the sequence type. You can also call it like an ordinary function to determine if an object is a sequence.

Example 1:

```
sequence s
s = {1,2,3}
```

Example 2:

```
if sequence(x) then
    sum = 0
    for i = 1 to length(x) do
        sum = sum + x[i]
    end for
else
    -- x must be an atom
    sum = x
end if
```

See Also: atom, object, integer, atoms and sequences

----------<set_active_page>----------

Platform: DOS32

Syntax:
```
include image.e
set_active_page(i)
```

Description: Select video page i to send all screen output to.

Comments: With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

Example:

```
include image.e

-- active & display pages are initially both 0
puts(1, "\nThis is page 0\n")
set_active_page(1)     -- screen output will now go to page 1
clear_screen()
puts(1, "\nNow we've flipped to page 1\n")
if getc(0) then        -- wait for key-press
end if
set_display_page(1)    -- "Now we've ..." becomes visible
if getc(0) then        -- wait for key-press
end if
set_display_page(0)    -- "This is ..." becomes visible again
set_active_page(0)
```

See Also: get_active_page, set_display_page, video_config

----------<set_display_page>----------

Platform: DOS32

Syntax:
```
include image.e
set_display_page(i)
```

Description: Set video page i to be mapped to the visible screen.

Comments: With multiple pages you can instantaneously change the entire screen without causing any visible "flicker". You can also save the screen and bring it back quickly.

video_config() will tell you how many pages are available in the current graphics mode.

By default, the active page and the display page are both 0.

This works under DOS, or in a full-screen DOS window. In a partial-screen window you cannot change the active page.

Example: See set_active_page() example.

See Also: get_display_page, set_active_page, video_config

----------<set_rand>----------

Syntax: include machine.e
set_rand(i1)

Description: Set the random number generator to a certain state, i1, so that you will get a known series of random numbers on subsequent calls to rand().

Comments: 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.

Example:

```
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
```

See Also: rand

----------<set_vector>----------

Platform: DOS32

Syntax: include machine.e
set_vector(i, s)

Description: Set s as the new address for handling interrupt number i. s must be a protected mode far address in the form: {16-bit segment, 32-bit offset}.

Comments: Before calling set_vector() you must store a machine-code interrupt handling routine at location s in memory.

The 16-bit segment can be the code segment used by Euphoria. To get the value of this segment see demo\dos32\hardint.ex. The offset can be the 32-bit value returned by allocate(). Euphoria runs in protected mode with the code segment and data segment

pointing to the same physical memory, but with different access modes.

Interrupts occurring in either real mode or protected mode will be passed to your handler. Your interrupt handler should immediately load the correct data segment before it tries to reference memory.

Your handler might return from the interrupt using the iretd instruction, or jump to the original interrupt handler. It should save and restore any registers that it modifies.

You should lock the memory used by your handler to ensure that it will never be swapped out. See lock_memory().

It is highly recommended that you study demo\dos32\hardint.ex before trying to set up your own interrupt handler.

You should have a good knowledge of machine-level programming before attempting to write your own handler.

You can call set_vector() with the far address returned by get_vector(), when you want to restore the original handler.

Example:

```
set_vector(#1C, {code_segment, my_handler_address})
```

Example Program: demo\dos32\hardint.ex

See Also: get_vector, lock_memory, allocate

<sin>

Syntax: x2 = sin(x1)

Description: Return the sine of x1, where x1 is in radians.

Comments: This function may be applied to an atom or to all elements of a sequence.

Example:

```
sin_x = sin({.5, .9, .11})
-- sin_x is {.479, .783, .110}
```

See Also: cos, tan

<sleep>

Syntax:
```
include misc.e
sleep(i)
```

Description: Suspend execution for i seconds.

Comments: On WIN32 and Linux/FreeBSD, the operating system will suspend your process and schedule other processes. On DOS32, your program will go into a busy loop for i seconds, during which time other processes may run, but they will compete with your process for the CPU.

Example:

```
puts(1, "Waiting 15 seconds...\n")
sleep(15)
puts(1, "Done.\n")
```

See Also: lock_file, abort, time

<sort>

Syntax:
```
include sort.e
s2 = sort(s1)
```

Description: Sort s1 into ascending order using a fast sorting algorithm. The elements of s1 can be any mix of atoms or sequences. Atoms come before sequences, and sequences are sorted "alphabetically" where the first elements are more significant than the later elements.

Example 1:

```
x = 0 & sort({7,5,3,8}) & 0
-- x is set to {0, 3, 5, 7, 8, 0}
```

Example 2:

```
y = sort({"Smith", "Jones", "Doe", 5.5, 4, 6})
-- y is {4, 5.5, 6, "Doe", "Jones", "Smith"}
```

Example 3:

```
database = sort({{"Smith",   95.0, 29},
                 {"Jones",   77.2, 31},
                 {"Clinton", 88.7, 44}})

-- The 3 database "records" will be sorted by the first "field"
-- i.e. by name. Where the first field (element) is equal it
-- will be sorted by the second field etc.

-- after sorting, database is:
               {{"Clinton", 88.7, 44},
                {"Jones",   77.2, 31},
                {"Smith",   95.0, 29}}
```

See Also: custom_sort, compare, match, find

----------<sound>----------

Platform: DOS32

Syntax: include graphics.e
sound(i)

Description: Turn on the PC speaker at frequency i. If i is 0 the speaker will be turned off.

Comments: On WIN32 and Linux/FreeBSD no sound will be made.

Example:

sound(1000) -- starts a fairly high pitched sound

----------<sprint>----------

Syntax: include misc.e
s = sprint(x)

Description: The representation of x as a string of characters is returned. 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.

Comments: 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: print, sprintf, value, get

----------<sprintf>----------

Syntax: s = sprintf(st, x)

Description: 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. st is a format string, x is the value or sequence of values to be formatted. printf(fn, st, x) is equivalent to puts(fn, sprintf(st, x)).

Comments: 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:

```
s = sprintf("%08d", 12345)
-- s is "00012345"
```

See Also: printf, value, sprint, get, system

----------<sqrt>----------

Syntax: x2 = sqrt(x1)

Description: Calculate the square root of x1.

Comments: This function may be applied to an atom or to all elements of a sequence.

Taking the square root of a negative number will abort your program with a run-time error message.

Example:

```
r = sqrt(16)
-- r is 4
```

See Also: log, power

----------<system>----------

Syntax: system(st, i)

Description: Pass a command string st to the operating system command interpreter. The argument i indicates the manner in which to return from the call to system():

When i is 0, the previous graphics mode is restored and the screen is cleared.

When i is 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.

When i is 2, the graphics mode is not restored and the screen is not cleared.

Comments: i = 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 DOS or Linux/FreeBSD shell.

system() allows you to use command-line redirection of standard input and output in the command string st.

Under DOS32, a Euphoria program will start off using extended memory. If extended memory runs out the program will consume conventional memory. If conventional memory runs out it will use virtual memory, i.e. swap space on disk. The DOS command run by system() will fail if there is not enough conventional memory available. To avoid this situation you can reserve some conventional (low) memory by typing:

```
SET CAUSEWAY=LOWMEM:xxx
```

where xxx is the number of K of conventional memory to reserve. Type this before running your program. You can also put this in autoexec.bat, or in a .bat file that runs your program. For example:

```
SET CAUSEWAY=LOWMEM:80
ex myprog.ex
```

This will reserve 80K of conventional memory, which should be enough to run simple DOS commands like COPY, MOVE, MKDIR etc.

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("ex \\test\\myprog.ex < indata > outdata", 2)
-- executes myprog by redirecting standard input and
-- standard output
```

Example Program: bin\install.ex

See Also: system_exec, dir, current_dir, getenv, command_line

----------<system_exec>----------

Syntax: i1 = system_exec(st, i2)

Description: Try to run the command given by st. st must be a command to run an executable program, possibly with some command-line arguments. If the program can be run, i1 will be the exit code from the program. If it is not possible to run the program, system_exec()

will return -1. i2 is a code that indicates what to do about the graphics mode when system_exec() is finished. These codes are the same as for system():

When i2 is 0, the previous graphics mode is restored and the screen is cleared.

When i2 is 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.

When i2 is 2, the graphics mode is not restored and the screen is not cleared.

Comments: On DOS32 or WIN32, system_exec() will only run .exe and .com programs. To run .bat files, or built-in DOS commands, you need system(). Some commands, such as DEL, are not programs, they are actually built-in to the command interpreter.

On DOS32 and WIN32, system_exec() does not allow the use of command-line redirection in the command string st.

exit codes from DOS or 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 DOS shell.

Example 1:

```
integer exit_code
exit_code = system_exec("xcopy temp1.dat temp2.dat", 2)

if exit_code = -1 then
    puts(2, "\n couldn't run xcopy.exe\n")
elsif exit_code = 0 then
    puts(2, "\n xcopy succeeded\n")
else
    printf(2, "\n xcopy failed with code %d\n", exit_code)
end if
```

Example 2:

```
-- executes myprog with two file names as arguments
if system_exec("ex \\test\\myprog.ex indata outdata", 2) then
    puts(2, "failure!\n")
end if
```

See Also: system, abort

----------<tan>----------

Syntax: x2 = tan(x1)

Description: Return the tangent of x1, where x1 is in radians.

Comments: This function may be applied to an atom or to all elements of a sequence.

Example:

```
t = tan(1.0)
-- t is 1.55741
```

See Also: sin, cos, arctan

----------<text_color>----------

Syntax:
```
include graphics.e
text_color(i)
```

Description: Set the foreground text color. Add 16 to get blinking text in some modes. See graphics.e for a list of possible colors.

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

----------<text_rows>----------

Platform: DOS32, WIN32

Syntax:
```
include graphics.e
i2 = text_rows(i1)
```

Description: Set the number of lines on a text-mode screen to i1 if possible. i2 will be set to the actual new number of lines.

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

See Also: graphics_mode

----------<tick_rate>----------

Platform: DOS32

Syntax: include machine.e
tick_rate(a)

Description: Specify the number of clock-tick interrupts per second. This determines the precision of the time() library routine. It also affects the sampling rate for time profiling.

Comments: tick_rate() is ignored on WIN32 and Linux/FreeBSD. The time resolution on WIN32 is always 100 ticks/second.

On a PC the clock-tick interrupt normally occurs at 18.2 interrupts per second. tick_rate() lets you increase that rate, but not decrease it.

tick_rate(0) will restore the rate to the normal 18.2 rate. Euphoria will also restore the rate automatically when it exits, even when it finds an error in your program.

If a program runs in a DOS window with a tick rate other than 18.2, the time() function will not advance unless the window is the active window.

While ex.exe is running, the system will maintain the correct time of day. However if ex.exe should crash (e.g. you see a "CauseWay..." error) while the tick rate is high, you (or your user) may need to reboot the machine to restore the proper rate. If you don't, the system time may advance too quickly. This problem does not occur on Windows 95/98/NT, only on DOS or Windows 3.1. You will always get back the correct time of day from the battery-operated clock in your system when you boot up again.

Example:

```
tick_rate(100)
-- time() will now advance in steps of .01 seconds
-- instead of the usual .055 seconds
```

See Also: time, time profiling

----------<time>----------

Syntax: a = time()

Description: Return the number of seconds since some fixed point in the past.

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

The resolution with DOS32 is normally about 0.05 seconds. On WIN32 and Linux/FreeBSD it's about 0.01 seconds.

Under DOS32 you can improve the resolution by calling tick_rate().

Under DOS32 the period of time that you can measure is limited to 24 hours. After that, the value returned by time() will reset and start over.

Example:

```
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, tick_rate

----------<trace>----------

Syntax:
```
with trace
trace(i)
```

Description: If i is 1 or 2, turn on full-screen interactive statement tracing/debugging. If i is 3, turn on tracing of statements to a file called ctrace.out. If i is 0, turn off tracing. When i is 1 a color display appears. When i is 2 a monochrome trace display appears. Tracing can only occur in routines that were compiled "with trace", and trace() has no effect unless it is executed in a "with trace" section of your program.

See Part I - 3.1 Debugging for a full discussion of tracing / debugging.

Comments: Use trace(2) if the color display is hard to view on your system.

trace(3) is supported by the Complete Edition (only) of the Euphoria To C Translator. Interactive tracing is not supported with the Translator.

All forms of trace() are supported in the Public Domain interpreter, but only for programs up to 300 statements in size. For larger programs you'll need the Complete Edition interpreter.

Example:

```
if x < 0 then
    -- ok, here's the case I want to debug...
    trace(1)
    -- etc.
    ...
end if
```

See Also: profile, debugging and profiling

<unlock_file>

Syntax: include file.e
unlock_file(fn, s)

Description: Unlock an open file fn, or a portion of file fn. You must have previously locked the file using lock_file(). On DOS32 and WIN32 you can unlock a range of bytes within a file by specifying the s parameter as {first_byte, last_byte}. The same range of bytes must have been locked by a previous call to lock_file(). On Linux/FreeBSD you can currently only lock or unlock an entire file. The s parameter should be {} when you want to unlock an entire file. On Linux/FreeBSD, s must always be {}.

Comments: 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 lock_file() for further comments and an example.

See Also: lock_file

<unregister_block>

Syntax: include machine.e (or safe.e)
unregister_block(a)

Description: Remove a block of memory from the list of safe blocks maintained by safe.e (the debug version of machine.e). The block starts at address a.

Comments: 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


-----------------------------------<upper>-----------------------------------

Syntax:      include wildcard.e
             x2 = upper(x1)

Description: Convert an atom or sequence to upper case.

Example:

             s = upper("Euphoria")
             -- s is "EUPHORIA"

             a = upper('g')
             -- a is 'G'

             s = upper({"Euphoria", "Programming"})
             -- s is {"EUPHORIA", "PROGRAMMING"}


See Also:    lower


---------------------------------<use_vesa>----------------------------------

Platform:    DOS32

Syntax:      include machine.e
             use_vesa(i)

Description: use_vesa(1) will force Euphoria to use the VESA graphics
             standard. This may cause Euphoria programs to work better in SVGA
             graphics modes with certain video cards. use_vesa(0) will restore
             Euphoria's original method of using the video card.

Comments:    Most people can ignore this. However if you experience difficulty
             in SVGA graphics modes you should try calling use_vesa(1) at the
             start of your program before any calls to graphics_mode().

             Arguments to use_vesa() other than 0 or 1 should not be used.

Example:

             use_vesa(1)
             fail = graphics_mode(261)


See Also:    graphics_mode


-----------------------------------<value>-----------------------------------
```

Syntax: include get.e
s = value(st)

Description: Read the string representation of a Euphoria object, and compute the value of that object. A 2-element sequence, {error_status, value} is actually returned, where error_status can be one of:

```
GET_SUCCESS -- a valid object representation was found
GET_EOF     -- end of string reached too soon
GET_FAIL    -- syntax is wrong
```

Comments: 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}.

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, sprintf, print

<video_config>

Platform: DOS32, Linux, FreeBSD

Syntax: include graphics.e
s = video_config()

Description: Return a sequence of values describing the current video configuration:
{color monitor?, graphics mode, text rows, text columns, xpixels, ypixels, number of colors, number of pages}

The following constants are defined in graphics.e:

```
global constant VC_COLOR   = 1,
                VC_MODE    = 2,
                VC_LINES   = 3,
```

```
                                VC_COLUMNS = 4,
                                VC_XPIXELS = 5,
                                VC_YPIXELS = 6,
                                VC_NCOLORS = 7,
                                VC_PAGES   = 8
```

Comments: This routine makes it easy for you to parameterize a program so it will work in many different graphics modes.

On the PC there are two types of graphics mode. The first type, text mode, lets you print text only. The second type, pixel-graphics mode, lets you plot pixels, or points, in various colors, as well as text. You can tell that you are in a text mode, because the VC_XPIXELS and VC_YPIXELS fields will be 0. Library routines such as polygon(), draw_line(), and ellipse() only work in a pixel-graphics mode.

Example:

```
vc = video_config()  -- in mode 3 with 25-lines of text:
-- vc is {1, 3, 25, 80, 0, 0, 32, 8}
```

See Also: graphics_mode

----------<wait_key>----------

Syntax:
```
include get.e
i = wait_key()
```

Description: Return the next key pressed by the user. Don't return until a key is pressed.

Comments: You could achieve the same result using get_key() as follows:

```
    while 1 do
        k = get_key()
        if k != -1 then
            exit
        end if
    end while
```

However, on multi-tasking systems like Windows or Linux/FreeBSD, this "busy waiting" would tend to slow the system down. wait_key() lets the operating system do other useful work while your program is waiting for the user to press a key.

You could also use getc(0), assuming file number 0 was input from the keyboard, except that you wouldn't pick up the special codes for function keys, arrow keys etc.

See Also: get_key, getc

----------<walk_dir>----------

Syntax: include file.e
i1 = walk_dir(st, i2, i3)

Description: This routine will "walk" through a directory with path name given by st. i2 is the routine id of a routine that you supply. walk_dir() will call your routine once for each file and subdirectory in st. If i3 is non-zero (TRUE), then the subdirectories in st will be walked through recursively.

The routine that you supply should accept 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().

Comments: 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, set the global integer my_dir to the routine id of your own modified dir() function that sorts the directory entries differently. See the default dir() function in file.e.

Example:

```
function look_at(sequence path_name, sequence entry)
-- this function accepts two sequences as arguments
    printf(1, "%s\\%s: %d\n",
           {path_name, entry[D_NAME], entry[D_SIZE]})
    return 0 -- keep going
end function

exit_code = walk_dir("C:\\MYFILES", routine_id("look_at"), TRUE)
```

Example Program: euphoria\bin\search.ex

See Also: dir, current_dir

----------<where>----------

Syntax: include file.e
i = where(fn)

Description: This function returns the current byte position in the file fn. This position is updated by reads, writes and seeks on the file. It is the place in the file where the next byte will be read from, or written to.

See Also: seek, open

----------<wildcard_file>----------

Syntax: include wildcard.e
i = wildcard_file(st1, st2)

Description: Return 1 (true) if the filename st2 matches the wild card pattern st1. Return 0 (false) otherwise. This is similar to DOS wildcard matching, but better in some cases. * matches any 0 or more characters, ? matches any single character. On Linux and FreeBSD the character comparisons are case sensitive. On DOS and Windows they are not.

Comments: 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.

In DOS "*ABC.*" will match all files. wildcard_file("*ABC.*", s) will only match when the file name part has "ABC" at the end (as you would expect).

Example 1:

```
i = wildcard_file("AB*CD.?", "aB123cD.e")
-- i is set to 1 on DOS or 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 Program: bin\search.ex

See Also: wildcard_match, dir

<wildcard_match>

Syntax:
```
include wildcard.e
i = wildcard_match(st1, st2)
```

Description: This function performs a general matching of a string against a pattern containing * and ? wildcards. It returns 1 (true) if string st2 matches pattern st1. It returns 0 (false) otherwise. * matches any 0 or more characters. ? matches any single character. Character comparisons are case sensitive.

Comments: If you want case insensitive comparisons, pass both st1 and st2 through upper(), or both through lower() before calling wildcard_match().

If you want to detect a pattern anywhere within a string, add * to each end of the pattern:

```
i = wildcard_match('*' & pattern & '*', string)
```

There is currently no way to treat * or ? literally in a pattern.

Example 1:

```
            i = wildcard_match("A?B*", "AQBXXYY")
            -- i is 1 (TRUE)
```

Example 2:

```
            i = wildcard_match("*xyz*", "AAAbbbxyz")
            -- i is 1 (TRUE)
```

Example 3:

```
            i = wildcard_match("A*B*C", "a111b222c")
            -- i is 0 (FALSE) because upper/lower case doesn't match
```

Example Program: bin\search.ex

See Also: wildcard_file, match, upper, lower, compare

----------<wrap>----------

Syntax: include graphics.e
wrap(i)

Description: Allow text to wrap at the right margin (i = 1) or get truncated (i = 0).

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

----------<xor_bits>----------

Syntax: x3 = xor_bits(x1, x2)

Description: Perform the logical XOR (exclusive OR) operation on corresponding bits in x1 and x2. A bit in x3 will be 1 when one of the two corresponding bits in x1 or x2 is 1, and the other is 0.

Comments: The arguments to this function may be atoms or sequences. The rules for operations on sequences apply.

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, int_to_bytes